

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xiii.

First Edition (November 1996)

This edition applies to Version 3 Release 7 Modification Level 0 of IBM VisualAge for C++ for AS/400 (5716-CX5) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. Please see "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1995, 1996. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xiii
Trademarks and Service Marks	xiii
About This Book	xv
Who Should Use This Book	xv
How to Use This Book	xv
Highlighting Conventions	xvi
How to Read the Syntax Diagrams	xvi
How to Get Help	xix
Getting Help Inside VisualAge for C++ for AS/400	xix
Getting Help from the Command Line	xx
Getting Help for a Keyword or Construct	xx
Online Documents Available in VisualAge for C++ for AS/400	xx
Chapter 1. The C++ Language	1
Overview of the C++ Language	1
C++ Support for Object-Oriented Programming	2
Data Abstraction	2
Encapsulation	2
Inheritance	3
Dynamic Binding and Polymorphism	3
Other Features of C++	3
C++ Programs	4
Declarations and Definitions	5
Scope	7
Local Scope	7
Function Scope	7
File Scope	7
Class Scope	8
Program Linkage	8
Internal Linkage	8
External Linkage	9
No Linkage	9
Simple C++ Input and Output	10
Output (cout, cerr, and clog)	10
Input (cin)	11
Linkage Specifications — Linking to non-C++ Programs	12
Chapter 2. Lexical Elements of C++	15
Tokens	15
Source Program Character Set	15
Trigraph Sequences	16
Comments	16
C++ Comments	18
Identifiers	18

Significant Characters in Identifiers	18
Case Sensitivity and Special Characters in Identifiers	18
Keywords	19
Constants	20
Integer Constants	20
Floating-Point Constants	22
Character Constants	23
String Literals	24
Escape Sequences	27
Chapter 3. Declarations	29
Declarations Overview	29
Objects	30
Storage Class Specifiers	31
auto Storage Class Specifier	32
extern Storage Class Specifier	35
register Storage Class Specifier	39
static Storage Class Specifier	40
typedef	44
Type Specifiers	45
Characters	45
Floating-Point Variables	48
Integer Variables	49
Enumerations	50
Pointers	55
void Type	60
Arrays	61
Structures	70
Unions	78
Incomplete Types	83
Declarators	83
volatile and const Qualifiers	84
_Packed Qualifier	86
Example Declarators	87
Initializers	88
Function Specifiers	90
References	91
Chapter 4. Expressions and Operators	93
Operator Precedence and Associativity	93
Operands	96
lvalues	96
Primary Expressions	97
C++ Scope Resolution Operator ::	97
Parenthesized Expressions ()	98
Constant Expressions	99
Function Calls ()	100
Array Subscript [] (Array Element Specification)	102

Dot Operator .	102
Arrow Operator ->	103
Unary Expressions	103
Increment ++	103
Decrement --	104
Unary Plus +	104
Unary Minus -	105
Logical Negation !	105
Bitwise Negation ~	105
Address &	106
Indirection *	106
Cast Expressions	106
sizeof (Size of an Object)	107
C++ new Operator	108
C++ delete Operator	113
C++ throw Expressions	114
Binary Expressions	114
Multiplication *	115
Division /	115
Remainder %	115
Addition +	116
Subtraction -	116
Bitwise Left and Right Shift << >>	117
Relational < > <= >=	118
Equality == !=	119
Bitwise AND &	120
Bitwise Exclusive OR ^	120
Bitwise Inclusive OR	121
Logical AND &&	121
Logical OR	122
C++ Pointer to Member Operators .* ->*	123
Conditional Expressions	123
Assignment Expressions	126
Simple Assignment =	126
Compound Assignment	128
Comma Expression ,	129
Chapter 5. Implicit Type Conversions	131
Integral Promotions	131
Standard Type Conversions	132
Signed-Integer Conversions	132
Unsigned-Integer Conversions	132
Floating-Point Conversions	133
Pointer Conversions	133
Reference Conversions	134
Pointer-to-Member Conversions	134
Function Argument Conversions	134
Other Conversions	135

Arithmetic Conversions	135
Chapter 6. Functions	137
Functions Overview	137
C++ Enhancements to C Functions	138
Function Declarations	138
C++ Function Declarations	139
Examples of Function Declarations	140
Related Information	142
Function Definitions	143
Function Declarator	145
Ellipsis and void	146
Function Body	146
Examples of Function Declarators	147
The main() Function	148
Arguments to main	149
Example of Arguments to main	149
Calling Functions and Passing Arguments	150
Passing Arguments by Reference	154
Default Arguments in C++ Functions	156
Restrictions on Default Arguments	157
Evaluating Default Arguments	158
Function Return Values	159
Using References as Return Types	160
Pointers to Functions	161
Inline Functions	163
Chapter 7. Statements	165
Labels	165
Block	166
break	168
continue	171
do	173
Expression	175
for	177
goto	179
if	180
Null Statement	182
return	182
switch	184
while	189
Chapter 8. Preprocessor Directives	191
Preprocessor Overview	191
Preprocessor Directive Format	192
Macro Definition and Expansion (#define)	192
Object-Like Macros	193
Function-Like Macros	194

Scope of Macro Names (#undef)	197
# Operator	198
Macro Concatenation with the ## Operator	199
Preprocessor Error Directive (#error)	201
File Inclusion (#include)	201
Predefined Macro Names	203
ANSI/ISO Standard Predefined Macro Names	203
VisualAge for C++ for AS/400 Predefined Macro Names	204
Additional VisualAge for C++ for AS/400 Predefined Macros	205
Examples of Predefined Macros	206
Conditional Compilation Directives	207
#if, #elif	208
#ifdef	209
#ifndef	209
#else	210
#endif	210
Examples of Conditional Compilation Directives	210
Line Control (#line)	211
Null Directive (#)	213
Pragma Directives (#pragma)	213
cancel_handler	214
chars	215
comment	215
define	216
disable_handler	216
enumsize	217
exception_handler	218
implementation	220
info	221
langlvl	221
map	222
mapinc	222
operational descriptor	225
pack	227
page	235
pagesize	235
pointer	235
priority	236
skip	237
strings	237
subtitle	238
title	238
Examples of #pragma Directives	238
Chapter 9. Classes	239
C++ Classes Overview	239
Classes and Polymorphic Functions	240
Classes and Structures	240

Aggregate Classes	241
Declaring Class Objects	241
Class Names	242
Using Class Objects	244
Scope of Class Names	246
Incomplete Class Declarations	247
Nested Classes	248
Local Classes	249
Local Type Names	250
Chapter 10. Class Members and Friends	253
Class Member Lists	253
Data Members	254
Class-Type Class Members	255
Member Functions	256
const and volatile Member Functions	256
Virtual Member Functions	256
Special Member Functions	256
Inline Member Functions	257
Member Function Templates	257
Member Scope	258
Pointers to Members	260
The this Pointer	262
Static Members	265
Using the Class Access Operators with Static Members	266
Static Data Members	267
Static Member Functions	269
Member Access	270
Classes and Access Control	270
Access Specifiers	271
Friends	273
Friend Scope	275
Friend Access	276
Chapter 11. Overloading	277
Overloading Functions	277
Declaration Matching	278
Restrictions on Overloaded Functions	278
Argument Matching in Overloaded Functions	279
Sequence of Argument Conversions	280
Trivial Conversions	281
Overloading Operators	281
General Rules for Overloading Operators	282
Operands of Overloaded Operators	283
Restrictions on Overloaded Operators	284
Overloading Unary Operators	285
Overloading Binary Operators	286
Special Overloaded Operators	287

Overloaded Assignment	287
Overloaded Function Calls	288
Overloaded Subscripting	288
Overloaded Class Member Access	288
Overloaded Increment and Decrement	290
Overloaded new and delete	291
Chapter 12. Special Member Functions	293
Constructors and Destructors Overview	293
Constructors	294
Default Constructors	294
Construction Order of Class Objects	296
Explicitly Constructing Objects	297
Destructors	297
Free Store	299
Temporary Objects	302
Related Information	303
User-Defined Conversions	304
Conversion by Constructor	304
Conversion Functions	305
Initialization by Constructor	306
Explicit Initialization	306
Initializing Base Classes and Members	308
Construction Order of Derived Class Objects	311
Copying Class Objects	311
Copy Restrictions	312
Copy by Assignment	312
Copy by Initialization	313
Chapter 13. Inheritance	315
Inheritance Overview	315
Multiple Inheritance	317
The Inheritance Design Process	318
Direct and Indirect Base Classes	318
Polymorphism	318
Derivation	319
Syntax of a Derived Class Declaration	323
Inherited Member Access	324
Protected Members	324
Derivation Access of Base Classes	325
Access Declarations	326
Access Resolution	329
Access Summary	331
Multiple Inheritance	332
Virtual Base Classes	333
Multiple Access	335
Accessible Base Classes	335
Ambiguous Base Classes	335

Virtual Functions	338
Ambiguous Virtual Function Calls	340
Virtual Function Access	342
Abstract Classes	343
Chapter 14. Templates	345
Templates Overview	346
Structuring Your Program Using Templates	349
Class Templates	351
Class Template Declarations and Definitions	353
Reference and Uniqueness	354
Nontype Template Arguments	354
Explicitly Defined Template Classes	356
Function Templates	356
Example of a Function Template	356
Overloading Resolution for Template Functions	357
Defining Template Functions	358
Explicitly Defined Template Functions	358
Function Template Declarations and Definitions	359
Differences between Class and Function Templates	360
Member Function Templates	361
Friends and Templates	364
Static Data Members and Templates	365
Chapter 15. Exception Handling	367
C++ Exception Handling Overview	367
Formal and Informal Exception Handling	368
Using Exception Handling	368
Transferring Control	370
Catching Exceptions	372
Matching between Exceptions Thrown and Caught	372
Order of Catching	372
Nested Try Blocks	373
Rethrowing an Exception	373
Using a Conditional Expression in a Throw Expression	375
Constructors and Destructors in Exception Handling	376
Exception Specifications	379
Exception Specification Syntax	379
Empty Exception Specifications	380
Functions without an Exception Specification	380
Other Exception Specifications	380
Special Exception Handling Functions	381
unexpected()	381
terminate()	381
set_unexpected() and set_terminate()	381
Example of Using the Exception Handling Functions	382
Appendix A. C and C++ Compatibility	385

C++ Constructs Not Found in ANSI/ISO C	385
Constructs Found in Both C++ and ANSI/ISO C	385
Character Array Initialization	385
Character Constants	386
Class and typedef Names	386
Class and Scope Declarations	386
const Object Initialization	386
Definitions	387
Definitions within Return or Argument Types	387
Enumerator Type	387
Enumeration Type	387
Function Declarations	387
Functions with an Empty Argument List	388
Global Constant Linkage	388
Jump Statements	388
Keywords	388
main() Recursion	388
Names of Nested Classes	388
Pointers to void	389
Prototype Declarations	389
Return without Declared Value	389
__STDC__ Macro	389
typedefs in Class Declarations	389
Bibliography	391
The IBM VisualAge for C++ for AS/400 Library	391
C and C++ Related Publications	391
Other Books You Might Need	391
Non-IBM Publications	391
Index	393

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks and Service Marks

The following terms are trademarks of International Business Machines Corporation in the United States or other countries or both:

AS/400	C/400
COBOL/400	Common User Access
CUA	IBM
IBMLink	ILE
Integrated Language Environment	Open Class
PROFS	RPG/400
VisualAge	

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

IBM's VisualAge products and services are not associated with or sponsored by VisualEdge Software Ltd.

About This Book

This book describes the C++ language implemented by the VisualAge for C++ for AS/400 product and contains language reference material to accompany the *IBM VisualAge for C++ for AS/400 C++ User's Guide*. It describes in detail the constructs that make up the C++ language, emphasizing the elements of C++ that are not part of the C language.

Use this book with the other publications described in the "Bibliography" on page 391.

For information about other AS/400 publications, see either of the following:

- The *Publications Reference* book, SC09-4003, in the AS/400 Softcopy library
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database containing descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with your system at no charge.

Who Should Use This Book

This book is intended for programmers who want to write C++ applications under the OS/400 operating system. This book is a reference rather than a tutorial. It assumes you have some experience with writing C or C++ programs and are familiar with your operating system.

How to Use This Book

You can either read this book sequentially or you can go directly to a topic. The following list describes the content of each chapter:

- Chapter 1, "The C++ Language," gives a brief overview of the features of C++, including a description of the C++ constructs that support object-oriented programming.
- Chapter 2, "Lexical Elements of C++," describes the basic elements of C++.
- Chapter 3, "Declarations," describes declarations and declarators. It also describes program linkage, storage classes, fundamental data types, and initialization of the fundamental data types.
- Chapter 4, "Expressions and Operators," describes the expressions and standard C and C++ operators used in computation.
- Chapter 5, "Implicit Type Conversions," describes the standard conversions performed on the fundamental C and C++ data types.
- Chapter 6, "Functions," describes the form and use of functions, including function declarations and definitions.
- Chapter 7, "Statements," describes the statements used to control the execution sequence of programs.

Reading the Syntax Diagrams

- Chapter 8, “Preprocessor Directives,” discusses preprocessor directives.
- Chapter 9, “Classes,” describes the concept of classes in C++, including a description of the different types of classes, how to declare class objects, and the scoping rules for class objects.
- Chapter 10, “Class Members and Friends,” describes the scoping rules for class members and member access rules.
- Chapter 11, “Overloading,” describes the form and use of overloaded functions and overloaded operators.
- Chapter 12, “Special Member Functions,” describes the member functions that are used to create, destroy, convert, initialize, and copy class objects.
- Chapter 13, “Inheritance,” describes the concept of inheritance, including a description of access control for derived and base classes.
- Chapter 14, “Templates,” describes class templates and function templates.
- Chapter 15, “Exception Handling,” describes the facilities C++ provides for handling errors and other exceptions.
- Appendix A, “C and C++ Compatibility,” summarizes the main differences between International Standards Organization C language definition (ANSI/ISO C) and C++.

Implementation dependencies are described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

Highlighting Conventions

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
<i>Italics</i>	Identify parameters whose actual names or values are to be supplied by the programmer. <i>Italics</i> are also used for the first mention of new terms that are defined in the glossary.
Example	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Examples illustrating the use of the VisualAge for C++ for AS/400 compiler are written in a simple style. They are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of C++ language constructs. Some examples are only code fragments and will not compile without additional code.

How to Read the Syntax Diagrams

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
- The ►— symbol indicates the beginning of a command, directive, or statement.

Reading the Syntax Diagrams

The \longrightarrow symbol indicates that the command, directive, or statement syntax is continued on the next line.

The \blacktriangleright symbol indicates that a command, directive, or statement is continued from the previous line.

The $\longrightarrow\blacktriangleleft$ symbol indicates the end of a command, directive, or statement.

Diagrams of syntactical units other than complete commands, directives, or statements start with the \blacktriangleright symbol and end with the \longrightarrow symbol.

Note: In the following diagrams, *statement* represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

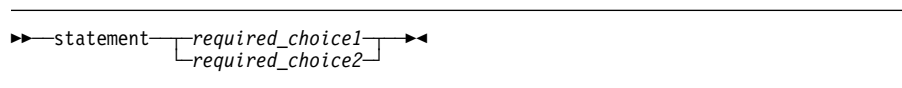


- Optional items appear below the main path.

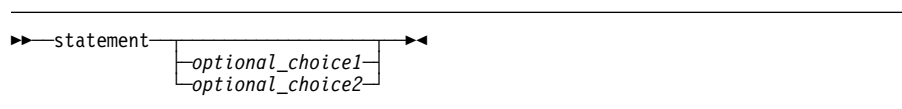


- If you can choose from two or more items, they appear vertically, in a stack.

If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.

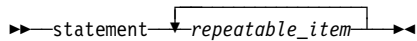


The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.

Reading the Syntax Diagrams



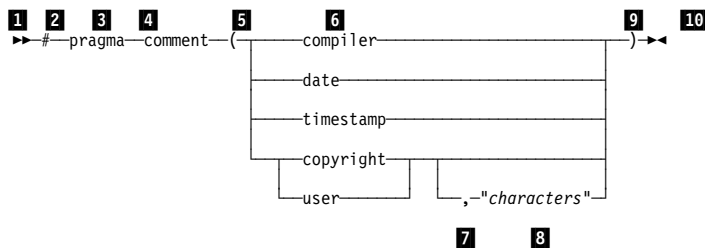
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

The following syntax diagram example shows the syntax for the **#pragma comment** directive. See “Pragma Directives (#pragma)” on page 213 for information on the **#pragma** directive.



- 1** This is the start of the syntax diagram.
- 2** The symbol # must appear first.
- 3** The keyword `pragma` must appear following the # symbol.
- 4** The keyword `comment` must appear following the keyword `pragma`.
- 5** An opening parenthesis must be present.
- 6** The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.
- 7** If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.
- 8** A character string must follow the comma. The character string must be enclosed in double quotation marks.
- 9** A closing parenthesis is required.
- 10** This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++ for AS/400:

Online documents

These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++ for AS/400. For your convenience, the online documents are presented in standard format (.INF files). See "Getting Help Inside VisualAge for C++ for AS/400" for instructions on opening standard format documents from inside VisualAge for C++ for AS/400. See "Getting Help from the Command Line" on page xx for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ for AS/400 documents that are available in standard format, see "Online Documents Available in VisualAge for C++ for AS/400" on page xx.

Contextual help

Contextual help is available throughout VisualAge for C++ for AS/400. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I help

Many of the common tasks that you want to perform with VisualAge for C++ for AS/400 are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++ for AS/400, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++ for AS/400

All three kinds of help are available directly within the VisualAge for C++ for AS/400 interface:

- To get general contextual help for the component of VisualAge for C++ for AS/400 that you are using, press **F1** anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.
- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
 - **Help Index**, an alphabetical list of all of the help topics that are available from this window
 - **General Help**, overall help for the window
 - **Using Help**, general information about the help facility

- **How Do I...**, the How Do I help for the component
- **Product Information**, a dialog that shows the level of VisualAge for C++ for AS/400 being used

In addition, there are selections that let you open all of online documents that are available in VisualAge for C++ for AS/400.

- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ for AS/400 folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++ for AS/400. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `iview` command. The installation routine stores the online document files in the `\ICCASW\HELP` directory. To view the *Language Reference*, for example, make `C:\ICCASW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for AS/400 for `C:`) and enter this command:

```
IVIEW CTTLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the *Language Reference*, you can enter the following command:

```
IVIEW CTTLNG.INF OPERATOR PRECEDENCE
```

Getting Help for a Keyword or Construct

If you are editing a file using the VisualAge Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing **Ctrl+H**. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing **Ctrl+H**.

Online Documents Available in VisualAge for C++ for AS/400

These documents are available in standard format:

- *VisualAge for C++ for AS/400 C++ User's Guide*
- *VisualAge for C++ for AS/400 C++ Programming Guide*
- *VisualAge for C++ for AS/400 IBM Open Class Library Reference*
- *VisualAge for C++ for AS/400 C Library Reference*
- *VisualAge for C++ for AS/400 C++ Language Reference*
- *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*
- *IBM Access Class Library User's Guide*
- *IBM Access Class Library for OS/400 Reference*
- *IBM Access Class Library for Windows Reference*

Overview of the C++ Language

Chapter 1. The C++ Language

This chapter introduces the C++ language implemented by the VisualAge for C++ for AS/400 compiler. It briefly summarizes the differences between C and C++, and discusses the principles of object-oriented programming.

This chapter discusses:

Overview of the C++ Language	1
C++ Support for Object-Oriented Programming	2
C++ Programs	4
Declarations and Definitions	5
Scope	7
Program Linkage	8
Simple C++ Input and Output	10
Linkage Specifications — Linking to non-C++ Programs	12

Overview of the C++ Language

C++ is an object-oriented language based on the C programming language. It can be viewed as a superset of C. Almost all of the features and constructs available in C are also available in C++. However, C++ is more than just an extension of C. Its additional features support the programming style known as *object-oriented programming*. Several features that are already available in C, such as input and output may be implemented differently in C++. In C++ you may use the conventional C input and output routines or you may use object oriented input and output by using the I/O Stream class library.

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories. It was originally based on the definition of the C language stated in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. This C language definition is commonly called *K&R C*. Since then, the International Standards Organization C language definition (referred to here as ANSI/ISO C) has been approved. It specifies many of the features that K&R left unspecified. Some features of ANSI/ISO C have been incorporated into the current definition of C++, and some parts of the ANSI/ISO C definition have been motivated by C++.

While there is currently no C++ standard comparable to the ANSI/ISO C definition, an ISO committee is working on such a definition. The draft of the Working Paper for Draft Proposed American National Standard for Information Systems—Programming Language C++, X3J16/95-0087, is the base document for the ongoing standardization of C++. The VisualAge for C++ for AS/400 compiler adheres to most aspects of the language specified in the ANSI/ISO working paper dated April 28, 1995.

Object-Oriented Programming

C++ Support for Object-Oriented Programming

Object-oriented programming is based on the concepts of *data abstraction*, *inheritance*, and *polymorphism*. Unlike procedural programming, it concentrates on the data objects that are involved in a problem and how they are manipulated, not on how something is accomplished. Based on the foundation of data abstraction, object-oriented programming allows you to reuse existing code more efficiently and increase your productivity.

Data Abstraction

Data abstraction provides the foundation for object-oriented programming. In addition to providing fundamental data types, object-oriented programming languages allow you to define your own data types, called *user-defined* or *abstract* data types. In the C programming language, related data items can be organized into structures. These structures can then be manipulated as units of data. In addition to providing this type of data structure, object-oriented programming languages allow you to implement a set of operations that can be applied to the data elements. The data elements and the set of operations applicable to the data elements together form the abstract data type.

To support data abstraction, a programming language must provide a construct that can be used to encapsulate the data elements and operations that make up an abstract data type. In C++, this construct is called a *class*. An instance of a class is called an *object*. Classes are composed of data elements called *data members* and *member functions* that define the operations that can be carried out on the data members.

Encapsulation

Another key feature of object-oriented programming is *encapsulation*. Encapsulation means a class can hide the details of:

- The representation of its data members
- The implementation of the operations that can be performed on these data members

Application programs manipulate objects of a class using a clearly defined interface. As long as this interface does not change, you can change the implementation of a class without having to change the application programs that use the class. Encapsulation provides the following advantages:

- Users of a class do not have to deal with unnecessary implementation details.
- Programs are easier to debug and maintain.
- Permitted alterations are clearly specified.

In C++, encapsulation is accomplished by specifying the level of access for each member of a class. Both the data members and member functions of a class can be declared **public**, **protected**, or **private** depending on the kind of access required.

Note: C++ encapsulation is *not* a true security mechanism. It is possible to circumvent the class access controls that make encapsulation possible. The language is not designed to prevent such misuse.

Object-Oriented Programming

Inheritance

Inheritance lets you reuse existing code and data structures in new applications. In C++, inheritance is implemented through class derivation. You can extend a library of existing classes by adding data elements and operations to existing classes to form *derived* classes. A derived class has all the members of its parent or *base* class, as well as extensions that can provide additional features. When you create a new derived class, you only have to write the code for the additional features. The existing features of the base class are already available.

A base class can have more than one class derived from it. In addition, a derived class can serve as a base class for other derived classes in a hierarchy. Typically, a derived class is more specialized than its base class.

A derived class can inherit data members and member functions from more than one base class. Inheritance from more than one base class is called *multiple inheritance*.

Dynamic Binding and Polymorphism

Another key concept that allows you to write generic programs is *dynamic* or *late binding*. Dynamic binding allows a member function call to be resolved at run time, according to the run-time type of an object reference. This permits each user-defined class in an inheritance hierarchy to have a different implementation of a particular function. Application programs can then apply that function to an object without needing to know the specifics of the class that the object belongs to.

In C++, dynamic binding hides the differences between a group of classes in an inheritance hierarchy from the application program. At run time, the system determines the specific class of the object and invokes the appropriate function implementation for that class.

Dynamic binding is distinguished from *static* or *compile-time* binding, which involves compile-time member function resolution according to the static type of an object reference.

Other Features of C++

C++ provides several other powerful extensions to the C programming language. Among these are:

- Constructors and destructors, which are used to create, initialize and destroy class objects
- Overloaded functions and operators, which let you extend the operations a function or operator can perform on different data types
- Inline functions, which make programs more efficient
- References, which allow a function to modify its arguments in the calling function
- Template functions and classes, which allow the definition of generic classes and functions
- Object-Oriented Exception handling, which provides transfer of control and recovery from errors and other exceptional circumstances

C++ Programs

C++ Programs

C++ programs contain many of the same programming statements and constructs as C programs:

- C++ has the same fundamental types (built-in) data types as C, as well as some types that are not built-in to C.
- Like ANSI/ISO C, C++ allows you to declare new type names by using the **typedef** construct. These new type names are not new types.
- In general, the scope and storage class rules for C also apply in C++.
- C and C++ have the same set of arithmetic and logical operators.

A C++ name can identify any of the following:

- an object
- a function
- a set of functions
- an enumerator
- a type
- a class member
- a template
- a value
- a label

A declaration introduces a name into a program and can define an area of storage associated with that name.

An expression can be evaluated and is composed of operations and operands. An expression ending with a ; (semicolon) is called a statement. A statement is the smallest independent computational unit. Functions are composed of groups of one or more statements.

A C++ program is composed of one or more functions. These functions can all reside in a single file or can be placed in different files that are linked to each other. In C++, a program must have one and only one non-member function called **main()**.

Declarations and Definitions

The following is a simple C++ program containing declarations, expressions, statements, and two functions:

```
/**
 ** A simple C++ program containing declarations,
 ** expressions, statements, and two functions:
 **/

#include <math.h>                // contains definition of abs()
double multiplier, common_ratio; // variable declarations
double geo_series(double a, double r) // function definition
{
    if (r == 1)                 // if statement
        return -1.0;           // return statement
    else if (abs(r) < 1.0)      // else if statement
        return (a / (1 - r));  // statement containing
                                // expression
    else return -2.0;
}
int main()                      // program execution begins here
{
    double sum;                // variable declaration
    multiplier = 2.2;          // initialization of external variable
    common_ratio = 3.1;        // initialization of external variable
    sum = geo_series(multiplier, common_ratio); // function call
    // ..
    return 0;
}
```

Declarations and Definitions

Declarations introduce identifiers into the program. Most objects in C++ must be declared before they are used. A declaration is a *definition* unless it:

- Declares a function without including the function body
- Contains the **extern** storage class specifier and has no initializer or function body
- Declares a static data member in a class declaration
- Declares a class name
- Declares a **typedef**

The following are only declarations:

```
extern double d;
extern int i;
int bar(int);
class C;
typedef float FP;
```

The following declarations are also definitions:

Declarations and Definitions

```
double d;
extern int i = 1;
int bar(int b) {return b;}
class C { int a; };
enum direction {vertical, horizontal};
```

A given function, object, or type can have only one definition. It can have more than one declaration as long as all of the declarations match. If a function is never called and its address is never taken, then you do not have to define it. If an object is declared but never used, or is only used as the operand of `sizeof`, you do not have to define it. You can declare a given class or enumerator more than once.

One of the fundamental differences between C++ and C is the placement of variable declarations. Although variables are declared in the same way, in C++, variable declarations can be placed anywhere in the program. In C, variable declarations must come before any other statements in a block. In the following C++ example, the variable `d` is declared in the middle of the `main()` function:

```
/**
 ** The variable d is declared
 ** in the middle of the main() function:
 **/

#include <iostream.h>
void main()
{
    int a, b;
    cout << "Please enter two integers" << endl;
    cin >> a >> b;
    int d = a + b;
    cout << "Here is the sum of your two integers:" << d << endl;
}
```

In C++ code, objects are represented by variables. A variable also represents the location in storage that contains the value of an object.

In C++ function declarations, you must declare the number and type of arguments that the function takes. This style of function declaration is the same as function prototyping in ANSI/ISO C. You must declare the type of each argument separately. In the following example, the function `yonge()` takes two **int** arguments and returns an **int** value:

```
int yonge(int, int);
```

You can include optional argument identifiers in a function declaration. For example:

```
int yonge(int a, int b);
```

Scope

The area of the code where an identifier is visible is referred to as the *scope* of the identifier. The four kinds of scope are:

- Local
- Function
- File
- Class

The scope of a name is determined by the location of the name's declaration.

A type name first declared in a function return type has file scope. A type name first declared in a function argument list has local scope.

A function name that is first declared as a friend of a class is in the first nonclass scope that encloses the class.

If the friend function is a member of another class, it has the scope of that class. The scope of a class name first declared as a friend of a class is the first nonclass enclosing scope. See “Friend Scope” on page 275 for more information.

Local Scope

A name has *local scope* if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block, but the name must be declared before it is used. When the block is exited, the names declared in the block are no longer available.

Formal argument names for a function have the scope of the outermost block of that function.

If a local variable is a class object with a destructor, the destructor is called when control passes out of the block in which the class object was constructed.

When one block is nested inside another, the variables from the outer block are usually visible in the nested block. However, if an outer block variable is redefined in a nested block, the new declaration is in effect in the inner block. The original declaration is restored when program control returns to the outer block. This is called *block visibility*.

Function Scope

The only type of identifier with *function scope* is a label name. A label is implicitly declared by its appearance in the program text and is visible throughout the function that declares it.

File Scope

A name has *file scope* if its declaration appears outside of all blocks and classes. A name with file scope is visible from the point where it is declared to the end of the source file. The name is also made accessible for the initialization of global variables.

Program Linkage

If a name is declared **extern**, it is also visible, at link time, in all object files being linked. Global names are names declared with file scope.

Class Scope

The name of a class member has *class scope* and can only be used in the following cases:

- In a member function of that class
- In a member function of a class derived from that class
- After the `.` (dot) operator applied to an instance of that class
- After the `.` (dot) operator applied to an instance of a class derived from that class
- After the `->` (arrow) operator applied to a pointer to an instance of that class
- After the `->` (arrow) operator applied to a pointer to an instance of a class derived from that class
- After the `::` (scope resolution) operator applied to the name of a class
- After the `::` (scope resolution) operator applied to a class derived from that class.

For more information on class scope, see “Scope of Class Names” on page 246.

Program Linkage

The association, or lack of association, between two identical identifiers is known as *linkage*. The kind of linkage that an identifier has depends on the way that it is declared.

A file scope identifier has one of the following kinds of *linkage*:

<i>Internal</i>	Identical identifiers within a single source file refer to the same data object or function.
<i>External</i>	Identical identifiers in separately compiled files refer to the same data object or function.
<i>No linkage</i>	Each identical identifier refers to a unique object.

Internal Linkage

The following kinds of identifiers have internal linkage:

- All identifiers with file or block scope that have the keyword **static** in their declarations. Functions with **static** storage class are visible only in the source file in which you define them.
- C++ identifiers declared at file scope with the specifier **const** and not explicitly declared **extern**.

A variable that has **static** storage class can be defined within a block or outside a function. If the definition occurs within a block, the variable has internal linkage and is only visible within the block after its declaration is seen. If the definition occurs outside a function, the variable has internal linkage and is available from the point where it is defined to the end of the current source file.

Program Linkage

A class that has no static members or noninline member functions, and that has not been used in the declaration of an object or function or class is local to its compilation unit.

If the declaration of an identifier has the keyword **extern** and if a previous declaration of the identifier is visible at file scope, the identifier has the same linkage as the first declaration.

External Linkage

The following kinds of identifiers have external linkage:

- Identifiers with file or block scope that have the keyword **extern** in their declarations.

If a previous declaration of the identifier is visible at file scope, the identifier has the same linkage as the first declaration. For example, a variable or function that is first declared with the keyword **static** and later declared with the keyword **extern** has internal linkage.

- Function identifiers declared without storage-class specifiers.
- Object identifiers that have file scope declarations without a storage-class specifier. Storage is allocated for such object identifiers.
- Static class members and noninline member functions.

Identifiers declared with the keyword **extern** can be defined in other compilation units.

No Linkage

The following kinds of identifiers have no linkage:

- Identifiers that do not represent an object or a function, including labels, enumerators, **typedef** names, type names, and template names
- Identifiers that represent a function argument
- Identifiers declared inside a block without the keyword **extern**

Note: During compilation, the compiler encodes all function names and certain other identifiers to include type and scope information. This encoding process is called *mangling*, and the mangled names are used in the object files and final executable file.

Input and Output

Simple C++ Input and Output

Like C, the C++ language has no built-in input and output facilities. Instead, input and output facilities for C++ are provided by the I/O Stream Library. For compatibility with C, C++ also supports the standard I/O functions of C. The I/O Stream Library supports a set of I/O operations, written in the C++ language, for the built-in types. You can extend these facilities to provide input and output functions for user-defined data types.

For a complete description of the I/O Stream Library, see the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*. C++ I/O is also summarized in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*.

There are four predefined I/O stream objects that you can use to perform standard I/O:

- **cout**
- **cin**
- **cerr**
- **clog**

You can use these in conjunction with the overloaded << (insertion or *output*) and >> (extraction or *input*) operators. To use these streams and operators, you must include the header file `iostream.h`. The following example prints Hello World! to standard output:

```
/**
** Hello World
**/

#include <iostream.h>
void main()
{
    cout << "Hello World!" << endl;
}
```

The manipulator **endl** acts as a newline character, causing any output following it to be directed to the next line. Because it also causes any buffered output to be flushed, **endl** is preferred over `\n` to end lines.

Output (cout, cerr, and clog)

The **cout** stream is associated with standard output. You can use the output operator in conjunction with **cout** to direct a value to standard output. Successive output operators are concatenated when applied to **cout**. The following example prints out three strings in a row and produces the same result as the previous example, printing Hello World! to standard output.

Input and Output

```
/**
 ** Another Hello World, illustrating concatenation with cout
 **/

#include <iostream.h>
void main()
{
    cout << "Hello "
         << "World"
         << "!"
         << endl;
}
```

Output operators are defined to accept arguments of any of the fundamental data types, as well as pointers, references, and array types. You can also overload the output operator to define output for your own class types.

The **cerr** and **clog** streams direct output to standard error. **cerr** provides unbuffered output, while **clog** provides buffered output. The following example checks for a division by zero condition. If one occurs, a message is sent to standard error.

```
/**
 ** Check for a division by zero condition.
 ** If one occurs, a message is sent to standard error.
 **/

#include <iostream.h>

main()
{
    double val1, val2;
    cout << "Divide Two Values" << endl;
    cout << "Enter two numeric values: " << endl;
    cin >> val1 >> val2;
    if (val2 == 0 )
    {
        cerr << "The second value must be non-zero" << endl;
        return;
    }
    cout << "The answer is " << val1 / val2 << endl;
}
```

Input (cin)

The **cin** class object is associated with standard input. You can use the input operator in conjunction with **cin** to read a value from standard input. By default, white space (including blanks, tabs, and new lines) is disregarded by the input operator. For example:

Linkage Specifications

```
/**
** This example illustrates linkage specifications
**/

extern "C" int printf(const char*,...);
void main()
{
    printf("hello\n");
}
```

Here the *string-literal*, "C", tells the compiler that the routine `printf(const char*,...)` is a C library function. Note that string literals used in linkage specifications are not case sensitive.

Some valid values for *string-literal* are:

"C++"	Default
"C"	ILE C procedure
"C nowiden"	ILE C procedure without widened parameters
"RPG"	ILE RPG procedure
"COBOL"	ILE COBOL procedure
"CL"	ILE CL procedure
"OS"	OS linkage call
"OS nowiden"	OS linkage call without widened parameters
"ILE"	General ILE function call
"ILE nowiden"	ILE function call without widened parameters
"VREF"	ILE function call with pointers in temporary storage. (Behaves the same as a regular call although parameters are passed to the function as if they were by reference.)
"VREF nowiden"	Same as "VREF" without widened parameters

If the value of *string-literal* is not recognized, ILE C type linkage is used.

Linkage Specifications

Chapter 2. Lexical Elements of C++

This chapter describes the following lexical elements of C++:

Tokens	15
Source Program Character Set	15
Comments	16
Identifiers	18
Constants	20

Tokens

Source code is treated during preprocessing and compilation as a sequence of tokens. There are five different types of tokens:

- Identifiers
- Keywords
- Literals
- Operators
- Other separators

Adjacent identifiers, keywords and literals must be separated with white space. Other tokens should be separated by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, new lines, form feeds and comments.

Source Program Character Set

The following lists the basic character set that must be available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

- The decimal digits 0 through 9

```
0 1 2 3 4 5 6 7 8 9
```

- The following graphic characters:

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] _ { } `
```

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (~) character in EBCDIC
- The split vertical bar (|) character in ASCII, which may be represented by the vertical bar (|) character on EBCDIC systems
- The space character

Comments

- The control characters representing new-line, horizontal tab, vertical tab, and form feed, and end of string (NULL character).

Trigraph Sequences

Some characters from the C character set are not available in all environments. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*. The trigraph sequences are:

??=	#	pound sign
??{	[left bracket
??}]	right bracket
??<	{	left brace
??>	}	right brace
??/	\	backslash
??'	^	caret
??!		vertical bar
??-	-	tilde

The preprocessor replaces trigraph sequences with the corresponding single-character representation.

Comments

Comments begin with the `/*` characters, end with the `*/` characters, and can span more than one line. You can put comments anywhere the language allows white space.

Comments are replaced during preprocessing by a single space character.

Multibyte characters can also be included within a comment.

Note: The `/*` or `*/` characters found in a character constant or string literal do not start or end comments.

In the following program, line 6 is a comment:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("This program has a comment.\n");
6      /* printf("This is a comment line and will not print.\n"); */
7      return 0;
8  }
```

Because the comment on line 6 is equivalent to a space, the output of this program is:

This program has a comment.

Because the comment delimiters are inside a string literal, line 5 in the following program is not a comment.

Comments

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("This program does not have \
6  /* NOT A COMMENT */ a comment.\n");
7  return 0;
8  }
```

The output of the program is:

This program does not have /* NOT A COMMENT */ a comment.

You cannot nest comments. Each comment ends at the first occurrence of */.

In the following example, the comments are highlighted:

```
1  /* A program with nested comments. */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      test_function();
8      return 0;
9  }
10
11 int test_function(void)
12 {
13     int number;
14     char letter;
15     /*
16     number = 55;
17     letter = 'A';
18     /* number = 44; */
19     */
20     return 999;
21 }
```

In `test_function`, the compiler reads the `/*` in line 15 through the `*/` in line 18 as a comment and line 19 as C language code, causing errors at line 19. To avoid commenting over comments already in the source code, you should use conditional compilation preprocessor directives to cause the compiler to bypass sections of a program. For example, instead of commenting out the above statements, change line 2 and lines 15 to 19 in the following way:

```
2  #define TEST_FUNCTION 0
   ...
16 #if TEST_FUNCTION
17     number = 55;
18     letter = 'A';
19     /*number = 44;*/
20 #endif /*TEST_FUNCTION */
```

Identifiers

Conditional compilation preprocessor directives are described in Chapter 8, “Preprocessor Directives” on page 191. Multibyte characters can also be included with a comment.

C++ Comments

C++ permits double-slash comments as part of the language definition.

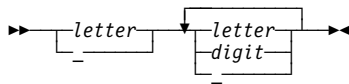
A C++ comment can span more than one physical source line if it is joined into one logical source line with line-continuation (\) characters. The backslash character can also be represented by a trigraph.

Identifiers

Identifiers consist of an arbitrary number of letters or digits. They provide names for the following language elements:

- Functions
- Data objects
- Labels
- Tags
- Parameters
- Macros
- Typedefs
- Structure and union members.

An identifier has the form:



Significant Characters in Identifiers

There is no limit for the number of characters in an identifier. However, only the first several characters of identifiers may be significant. The following table shows the number of significant characters for several kinds of identifiers.

Identifier	Maximum Number of Significant Characters
Static data objects	100 characters
Static function names	100 characters
External data objects	100 characters
External function names	100 characters

Case Sensitivity and Special Characters in Identifiers

The compiler distinguishes between uppercase and lowercase letters in identifiers. For example, PROFIT and profit represent different data objects. For complete portability, never use different case representations to refer to the same object.

Identifiers

Avoid creating identifiers that begin with an underscore (`_`) for function names and variable names.

The first character in an identifier must be a letter. The `_` (underscore) character is considered a letter; however, identifiers beginning with an underscore are reserved by the compiler for identifiers at file scope.

Identifiers that begin with two underscores or an underscore followed by a capital letter, are reserved in all contexts.

Although the names of system calls and library functions are not reserved words if you do not include the appropriate headers, avoid using them as identifiers. Duplication of a predefined name can lead to confusion for the maintainers of your code and can cause errors at link time or run time. If you include a library in a program, be aware of the function names in that library to avoid name duplications.

You should always include the appropriate headers when using standard library functions.

Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved but `AUTO` is not. The following lists the keywords common to both the C and C++ languages. These keywords are also included in the ANSI/ISO C language definition:

Table 1. Keywords Common to C and C++

_Packed	double	int	struct
auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do			

Note: `_Packed` is an extension to the ANSI/ISO standard.

The C++ language also reserves the following keywords:

Table 2. C++ Keywords

asm	inline	public	virtual
catch	new	template	wchar_t
class	operator	this	
delete	private	throw	
friend	protected	try	

Constants

Constants

A *constant* does not change its value while the program is running. The value of any constant must be in the range of representable values for its type.

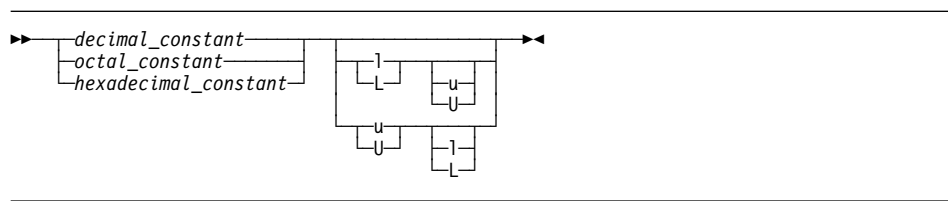
The C language contains the following types of constants (also called *literals*):

- Integer
- Floating-Point
- Character
- String
- Enumeration

Enumeration constants, which belong to the lexical class of identifiers, are discussed in “Enumerations” on page 50. For more information on data types, see “Type Specifiers” on page 45.

Integer Constants

Integer constants can represent decimal, octal, or hexadecimal values.



Data Types for Integer Constants

The data type of an integer constant is determined by the form, value, and suffix of the constant. The following lists the integer constants and shows the possible data types for each constant. The smallest data type that can represent the constant value is used to store the constant.

Table 3. Data Types for Integer Constants

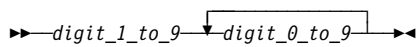
Constant	Data Type
unsuffixed decimal	int, long int, unsigned long int
unsuffixed octal	int, unsigned int, long int, unsigned long int
unsuffixed hexadecimal	int, unsigned int, long int, unsigned long int
suffixed by u or U	unsigned int, unsigned long int
suffixed by l or L	long int, unsigned long int
suffixed by both u or U , and l or L	unsigned long int

A plus (+) or minus (-) symbol can precede the constant. It is treated as a unary operator rather than as part of the constant value.

Constants

Decimal Constants

A *decimal constant* contains any of the digits 0 through 9. The first digit cannot be 0.



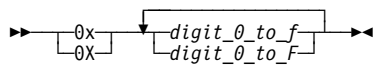
Integer constants beginning with the digit 0 are interpreted as an octal constant, rather than as a decimal constant.

The following are examples of decimal constants:

```
485976
-433132211
+20
5
```

Hexadecimal Constants

A *hexadecimal constant* begins with the 0 digit followed by either an x or X, followed by any combination of the digits 0 through 9 and the letters a through f or A through F. The letters A (or a) through F (or f) represent the values 10 through 15, respectively.



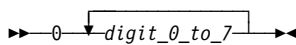
The following are examples of hexadecimal constants:

```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```


Constants

Octal Constants

An *octal constant* begins with the digit 0 and contains any of the digits 0 through 7.



The following are examples of octal constants:

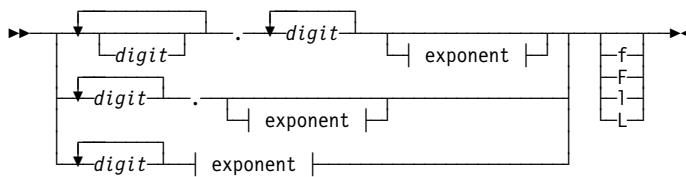
```
0
0125
034673
03245
```

Floating-Point Constants

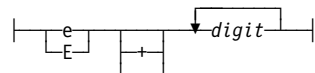
A *floating-point constant* consists of:

- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix.

Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part, but not both.



Exponent:



The representation of a floating-point number on a system is unspecified. If a floating-point constant is too large or too small, the result is undefined by the language.

The suffix **f** or **F** indicates a type of **float**, and the suffix **l** or **L** indicates a type of **long double**. If a suffix is not specified, the floating-point constant has a type **double**.

A plus (+) or minus (-) symbol can precede a floating-point constant. However, it is not part of the constant; it is interpreted as a unary operator.

Constants

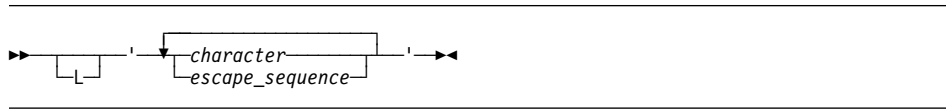
The following are examples of floating-point constants:

Floating-Point Constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Note: When you use the **printf** function to display a floating-point constant value, make certain that the **printf** conversion code modifiers that you specify are large enough for the floating-point constant value.

Character Constants

A *character constant* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols.



At least one character or escape sequence must appear in the character constant. The characters can be any from the source program character set, excluding the single quotation mark, backslash and new-line symbols. The prefix **L** indicates a wide character constant. A character constant must appear on a single logical source line.

The value of a character constant containing a single character is the numeric representation of the character in the character set used at run time. The value of a wide character constant containing a single multibyte character is the code for that character, as defined by the **mbtowc** function. If the character constant contains more than one character, the last 4 bytes represent the character constant. In C++, a character constant can contain only one character.

In C, a character constant has type **int**. In C++, a character constant has type **char**.

A wide character constant is represented by a double-byte character of type **wchar_t**. Multibyte characters represent character sets that use more than one byte in their representation. Each multibyte character can contain up to 4 bytes.

You can represent the double quotation mark symbol by itself, but you must use the backslash symbol followed by a single quotation mark symbol (`\'` escape sequence) to represent the single quotation mark symbol.

You can represent the new-line character by the `\n` new-line escape sequence.

You can represent the backslash character by the `\\` backslash escape sequence.

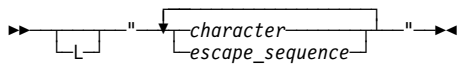
Constants

The following are examples of character constants:

```
'a'    '\''
'0'    '('
'x'    '\n'
'7'    '\\117'
'C'
```

String Literals

A *string constant* or *literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols.



The prefix **L** indicates a wide-character string literal.

A null (`'\0'`) character is appended to each string. For a wide character string (a string prefixed by the letter **L**), the value `'\0'` of type **wchar_t** is appended. By convention, programs recognize the end of a string by finding the null character.

Multiple spaces contained within a string constant are retained.

To continue a string on the next line, use the line continuation sequence (`\` symbol immediately followed by a new-line character). A carriage return must immediately follow the backslash. In the following example, the string literal second causes a compile-time error.

```
char *first = "This string continues onto the next\  
line, where it ends.";          /* compiles successfully. */  
char *second = "The comment makes the \  
invisible to the compiler.";    /* compilation error. */
```

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals are concatenated to produce a single string. You cannot concatenate a wide string constant with a character string constant. For example:

```
"hello " "there"    /* is equivalent to "hello there" */  
"hello " L"there"  /* is not valid */  
"hello" "there"    /* is equivalent to "hellothere" */
```

Characters in concatenated strings remain distinct. For example, the strings `"\xab"` and `"3"` are concatenated to form `"\xab3"`. However, the characters `\xab` and `3` remain distinct and are not merged to form the hexadecimal character `\xab3`.

Following any concatenation, `'\0'` of type **char** is appended at the end of each string. C++ programs find the end of a string by scanning for this value. For a wide-character string literal, `'\0'` of type **wchar_t** is appended. For example:

Constants

```
char *first = "Hello ";           /* stored as "Hello \0"      */
char *second = "there";          /* stored as "there\0"     */
char *third = "Hello " "there";  /* stored as "Hello there\0" */
```

A character string constant has type *array of char* and static storage duration. A wide character constant has type *array of wchar_t* and static storage duration.

Use the escape sequence `\n` to represent a new-line character as part of the string. Use the escape sequence `\\` to represent a backslash character as part of the string. You can represent the single quotation mark symbol by itself `'`, but you use the escape sequence `\"` to represent the double quotation mark symbol.

For example:

```
/**
 ** This example illustrates escape sequences in string literals
 **/

#include <iostream.h>
void main ()
{
    char *s = "Hi there! \n";
    cout << s;
    char *p = "The backslash character \\\.";
    cout << p << endl;
    char *q = "The double quotation mark \".\n";
    cout << q ;
}
```

This program produces the following output:

```
Hi there!
The backslash character \.
The double quotation mark ".
```

You should be careful when modifying string literals because the resulting behavior depends on whether your strings are stored in read/write static memory. C strings are read/write by default. C++ strings are readonly by default.

Use the **#pragma strings** preprocessor directive to change the default storage for string literals. The **#pragma strings** directive is described in “strings” on page 237.

In C++ string literals are stored in static storage and can be modified like any other storage location. C++ has the concept of *readonly* and *writable* strings. This deals with how multiple occurrences of strings are stored rather than whether or not the strings can be modified.

When a string literal appears more than once in the program source, how that string is stored depends on whether strings are *readonly* or *writable*. If strings are *readonly*, then only one location will be allocated for that string and all occurrences will refer to that one location. If strings are *writable*, then each occurrence of the string will have a separate, distinct storage location.

Constants

By default, the compiler will consider strings to be *writable*. You can change this using the **#pragma strings** preprocessor directive. Caution should be used with *readonly* strings since the single instance of the string could be modified inadvertently as is shown in the following simple example.

```
#pragma strings(readonly)
#include <stdio.h>

/*Since "readonly" strings are being used, the string literal */
/*"ABC" will only be allocated in storage once. The address of */
/*that location will be stored in both 'p1' and 'p2'. */

char *p1 = "ABC";
char *p2 = "ABC";

main() {
    /* change the first character */
    *p1 = 'a'; /* pointed to by 'p1' */

    /* output the string pointed to */
    printf("p2: %3.3s \n", p2); /* by 'p2' to see that it has */
    /* also changed. */
}
```

The output from this example is:

```
p2: aBC
```

If the **#pragma strings** directive had not been used at all or it was used to specify *writable* strings, then `p2` would be pointing to a different copy of `ABC` which would not have been affected by the change made using the `p1` pointer. Therefore, the output in this case would be:

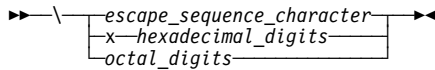
```
p2: ABC
```

The following are examples of string literals:

```
char titles[ ] = "Handel's \"Water Music\"";
char *mail_addr = "Last Name   First Name   MI   Street Address   \
    City   Province   Postal code ";
char *temp_string = "abc" "def" "ghi"; /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";
```

Escape Sequences

You can represent any member of the execution character set by an *escape sequence*. They are primarily used to put nonprintable characters in character and string literals. For example, you can use escape sequences to put such characters as tab, carriage return, and backspace into an output stream.



An escape sequence contains a backslash (\) symbol followed by one of the escape sequence characters or an octal or hexadecimal number. A hexadecimal escape sequence contains an x followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence uses up to three octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

Note: The line continuation sequence (\ followed by a new-line character) is not an escape sequence. It is used in character strings to indicate that the current line continues on the next line.

The escape sequences and the characters they represent are:

Escape Sequence	Character Represented
\a	Alert (bell, alarm)
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\\	Backslash

The value of an escape sequence represents the member of the character set used at run time. Escape sequences are translated during preprocessing. For example, on a system using the ASCII character codes, the value of the escape sequence \x56 is the letter V. On a system using EBCDIC character codes, the value of the escape sequence \xE5 is the letter V.

Use escape sequences only in character constants or in string literals. An error message is issued if an escape sequence is not recognized.

In string and character sequences, when you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a \\ backslash escape sequence. For example:

```
cout << "The escape sequence \\n." << endl;
```

Constants

This statement results in the following output:

The escape sequence `\n`.

Chapter 3. Declarations

A *declaration* establishes the names and characteristics of data objects and functions used in a program. A *definition* allocates storage for data objects or specifies the body for a function. When you define a type, no storage is allocated.

This chapter discusses:

Declarations Overview	29
Objects	30
Storage Class Specifiers	31
Type Specifiers	45
Declarators	83
Initializers	88
Function Specifiers	90
References	91

Declarations Overview

Declarations determine the following properties of data objects and their identifiers:

- Scope, which describes the visibility of an identifier in a block or source file. For a complete description of scope, see “Scope” on page 7.
- Linkage, which describes the association between two identical identifiers. See “Program Linkage” on page 8 for more information.
- Type, which describes the kind of data the object is to represent.

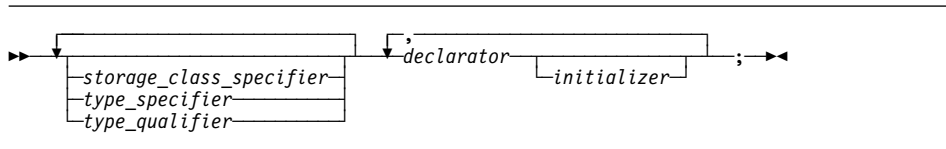
The lexical order of elements of declaration for a data object is as follows:

- Storage duration and linkage specification, described in “Storage Class Specifiers” on page 31
- Type specification, described in “Type Specifiers” on page 45
- Declarators, which introduce identifiers and make use of type qualifiers and storage qualifiers, described in “Declarators” on page 83
- Initializers, which initialize storage with initial values, described in “Initializers” on page 88.

Function declarations are described in Chapter 6, “Functions” on page 137.

Objects

All data declarations have the form:



The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. In the case of a function, the corresponding definition is the code or body of the function. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

Table 4. Examples of Declarations and Definitions

Declarations	Declarations and Definitions
<code>extern double pi;</code>	<code>double pi = 3.14159265;</code>
<code>float square(float x);</code>	<code>float square(float x) { return x*x; }</code>
<code>struct payroll;</code>	<code>struct payroll { char *name; float salary; } employee;</code>

Objects

An *object* is a region of storage that contains a value or group of values. Each value can be accessed using its identifier or a more complex expression that refers to the object. In addition, each object has a unique *data type*. Both the identifier and data type of an object are established in the object *declaration*.

The data type of an object determines the initial storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type-checking operations.

C++ has built-in, or *standard*, data types and user-defined data types. Standard data types include signed and unsigned integers, floating-point numbers, and characters. User-defined types include enumerations, structures, unions, and classes.

An instance of a class type is commonly called a *class object*. The individual class members are also called objects. The set of all member objects comprises a class object.

Storage Class Specifiers

Storage Class Specifiers

The storage class specifier used within the declaration determines whether:

- The object has internal, external, or no linkage.
- The object is to be stored in memory or in a register, if available.
- The object receives the default initial value 0 or an indeterminate default initial value.
- The object can be referenced throughout a program or only within the function, block, or source file where the variable is defined.
- The storage duration for the object is static (storage is maintained throughout program run time) or automatic (storage is maintained only during the execution of the block where the object is defined).

For a function, the storage class specifier determines the linkage of the function.

Declarations with the **auto** or **register** storage-class specifier result in automatic storage. Those with the **extern** or **static** storage-class specifier result in static storage.

Most local declarations that do not include the **extern** storage-class specifier allocate storage; however, function declarations and type declarations do not allocate storage.

The only storage-class specifiers allowed in a global or file scope declaration are **static** and **extern**.

This section describes the following storage class specifiers:

- **auto**
- **extern**
- **register**
- **static**
- **typedef**

Storage Class Specifiers

auto Storage Class Specifier

The **auto** storage class specifier lets you define a variable with automatic storage; its use and storage is restricted to the current block. The storage class keyword **auto** is optional in a data declaration. It is not permitted in a parameter declaration. A variable having the **auto** storage class specifier must be declared within a block. It cannot be used for file scope declarations.

Because automatic variables require storage only while they are actually being used, defining variables with the **auto** storage class can decrease the amount of memory required to run a program. However, having many large automatic objects may cause you to run out of stack space.

Declaring variables with the **auto** storage class can also make code easier to maintain, because a change to an **auto** variable in one function never affects another function (unless it is passed as an argument).

Initialization

You can initialize any **auto** variable except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Note: If you use the **goto** statement to jump into the middle of a block, automatic variables within that block are not initialized.

Storage

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered, storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Storage Class Specifiers

Examples of auto Storage Class

The following program shows the scope and initialization of **auto** variables. The function `main` defines two variables, each named `auto_var`. The first definition occurs on line 10. The second definition occurs in a nested block on line 13. While the nested block is running, only the `auto_var` created by the second definition is available. During the rest of the program, only the `auto_var` created by the first definition is available.

```
1  /*****
2  ** Example illustrating the use of auto variables **
3  *****/
4
5  #include <stdio.h>
6
7  int main(void)
8  {
9      void call_func(int passed_var);
10     auto int auto_var = 1; /* first definition of auto_var */
11
12     {
13         int auto_var = 2; /* second definition of auto_var */
14         printf("inner auto_var = %d\n", auto_var);
15     }
16     call_func(auto_var);
17     printf("outer auto_var = %d\n", auto_var);
18     return 0;
19 }
20
21 void call_func(int passed_var)
22 {
23     printf("passed_var = %d\n", passed_var);
24     passed_var = 3;
25     printf("passed_var = %d\n", passed_var);
26 }
```

This program produces the following output:

```
inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1
```

Storage Class Specifiers

The following example uses an array that has the storage class **auto** to pass a character string to the function `sort`. The function `sort` receives the address of the character string, rather than the contents of the array. The address enables `sort` to change the values of the elements in the array.

```

/*****
** Sorted string program -- this example passes an array name **
** to a function **
*****/

#include <stdio.h>
#include <string.h>

int main(void)
{
    void sort(char *array, int n);
    char string[75];
    int length;

    printf("Enter letters:\n");
    scanf("%74s", string);
    length = strlen(string);
    sort(string,length);
    printf("The sorted string is: %s\n", string);

    return(0);
}

void sort(char *array, int n)
{
    int gap, i, j, temp;

    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && array[j] > array[j + gap];
                j -= gap)
            {
                temp = array[j];
                array[j] = array[j + gap];
                array[j + gap] = temp;
            }
}

```

Storage Class Specifiers

When the program is run, interaction with the program could produce:

Output Enter letters:

Input zyfab

Output The sorted string is: abfyz

Related Information

- “register Storage Class Specifier” on page 39
- “Address &” on page 106
- “Function Declarator” on page 145

extern Storage Class Specifier

The **extern** storage class specifier lets you declare objects and functions that several source files can use. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

You can distinguish an **extern** declaration from an **extern** definition by the presence of the keyword **extern** and the absence of an initial value. If the keyword **extern** is absent or if there is an initial value, the declaration is also a definition; otherwise, it is just a declaration. An **extern** definition can appear only at file scope.

An **extern** variable, function definition, or declaration also makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any **extern** declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword **extern** is optional.

If you do not specify a storage class specifier, the function has external linkage. It is an error to include a declaration for the same function with the storage class specifier **static** before the declaration with no storage class specifier because of the incompatible declarations. Including the **extern** storage class specifier on the original declaration is valid and the function has internal linkage.

C++ Note:

Storage Class Specifiers

In C++, an **extern** declaration cannot appear in class scope.

Initialization

You can initialize any object with the **extern** storage class specifier at file scope. You can initialize an **extern** object with an initializer that must either:

- Appear as part of the definition and the initial value must be described by a constant expression.
- OR
- Reduce to the address of a previously declared object with static storage duration. This object may be modified by adding or subtracting an integral constant expression.

If you do not explicitly initialize an **extern** variable, its initial value is zero of the appropriate type. Initialization of an **extern** object is completed by the time the program starts running.

Storage

extern objects have static storage duration. Memory is allocated for **extern** objects before the **main** function begins running. When the program finishes running, the storage is freed.

Storage Class Specifiers

Examples of extern Storage Class

The following program shows the linkage of **extern** objects and functions. The **extern** object `total` is declared on line 12 of File 1 and on line 11 of File 2. The definition of the external object `total` appears in File 3. The **extern** function `tally` is defined in File 2. The function `tally` can be in the same file as `main` or in a different file. Because `main` precedes these definitions and `main` uses both `total` and `tally`, `main` declares `tally` on line 11 and `total` on line 12.

File 1

```
1  /*****
2  ** The program receives the price of an item, adds the      **
3  ** tax, and prints the total cost of the item.              **
4  *****/
5  *****/
6
7  #include <stdio.h>
8
9  int main(void)
10 {
11     void tally(void);          /* declaration of function tally */
12     extern float total;       /* first declaration of total   */
13
14     printf("Enter the purchase amount: \n");
15     tally();
16     printf("\nWith tax, the total is: %.2f\n", total);
17
18     return(0);
19 }
```

File 2

```
1  /*****
2  ** This file defines the function tally                      **
3  *****/
4  *****/
5  *****/
6  #include <stdio.h>
7
8  #define tax_rate 0.05
9
10 void tally(void)
11 {
12     /* begin tally */
13     float tax;
14     extern float total;    /* second declaration of total */
15
16     scanf("%f", &total);
17     tax = tax_rate * total;
18     total += tax;
19 }
```


Storage Class Specifiers

File 3

```
1 float total;
```

When this program is run, interaction with it could produce:

Output Enter the purchase amount:

Input 99.95

Output With tax, the total is: 104.95

The following program shows **extern** variables used by two functions. Because both functions `main` and `sort` can access and change the values of the **extern** variables `string` and `length`, `main` does not have to pass parameters to `sort`.

```
/* ****  
** Sorted string program -- this example shows extern          **  
** used by two functions                                     **  
**** */  
  
#include <stdio.h>  
#include <string.h>  
  
char string[75];  
int length;  
  
int main(void)  
{  
    void sort(void);  
  
    printf("Enter letters:\n");  
    scanf("%s", string);  
    length = strlen(string);  
    sort();  
    printf("The sorted string is: %s\n", string);  
  
    return(0);  
}
```

Storage Class Specifiers

```
void sort(void)
{
    int gap, i, j, temp;

    for (gap = length / 2; gap > 0; gap /= 2)
        for (i = gap; i < length; i++)
            for (j = i - gap;
                 j >= 0 && string[j] > string[j + gap];
                 j -= gap)
                {
                    temp = string[j];
                    string[j] = string[j + gap];
                    string[j + gap] = temp;
                }
}
```

When this program is run, interaction with it could produce:

```
Output    Enter letters:
Input     zyfab
Output    The sorted string is: abfyz
```

The following program shows a **static** variable `var1`, which is defined at file scope and then declared with the storage class specifier **extern**. The second declaration refers to the first definition of `var1` and so it has internal linkage.

```
static int var1;
:
extern int var1;
```

Related Information

- “Constant Expressions” on page 99
- “Function Definitions” on page 143
- “Function Declarator” on page 145

register Storage Class Specifier

The **register** storage class specifier indicates to the compiler that a heavily used variable (such as a loop control variable) within a block scope data definition or a parameter declaration should be allocated a register to minimize access time.

It is equivalent to the **auto** storage class except that the compiler places the object, if possible, into a machine register for faster access. The **register** storage class keyword is required in a data definition and in a parameter declaration that describes an object having the **register** storage class. An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.

Initialization

You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the

Storage Class Specifiers

expression representing the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Storage

Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block are made available. When the block is exited, the objects are no longer available for use.

If a **register** object is defined within a function that is recursively invoked, the memory is allocated for the variable at each invocation of the block.

The **register** storage class specifier indicates that the object is heavily used and indicates to the compiler that the value of the object should reside in a machine register. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers.

If the compiler does not allocate a machine register for a **register** object, the object is treated as having the storage class specifier **auto**.

Restrictions

You cannot use the **register** storage class specifier in file scope data declarations. In C programs, you cannot apply the address (&) operator to **register** variables. However, C++ lets you take the address of an object with the **register** storage class. For example:

```
register i;
int* b = &i;    // valid in C++, but not in C
```

Related Information

- “auto Storage Class Specifier” on page 32
- “Address &” on page 106
- “Parameter Declaration List Syntax” on page 145

static Storage Class Specifier

The **static** storage class specifier lets you define objects with static storage duration and internal linkage, or to define functions with internal linkage.

An object having the **static** storage class specifier can be defined within a block or at file scope. If the definition occurs within a block, the object has no linkage. If the definition occurs at file scope, the object has internal linkage.

Initialization

You can initialize any **static** object with a constant expression or an expression that reduces to the address of a previously declared **extern** or static object, possibly modified by a constant expression. If you do not provide an initial value, the object receives the value of zero of the appropriate type.

Storage Class Specifiers

Storage

Objects with the **static** storage class specifier have static storage duration. The storage for a **static** variable is made available when the program begins running. When the program finishes running, the memory is freed.

Usage

You can use **static** variables when you need an object that retains its value from one execution of a block to the next execution of that block. Using the **static** storage class specifier keeps the system from reinitializing the object each time the block where the object is defined runs.

If a local **static** variable is a class object with constructors and destructors, the object is constructed when control passes through its definition for the first time. If a local class object is created by a constructor, its destructor is called immediately before or as part of the calls of the **atexit** function.

Restrictions

You cannot declare a **static** function at block scope.

Examples of Static Storage Class

The following program shows the linkage of **static** identifiers at file scope. This program uses two different external **static** identifiers named `stat_var`. The first definition occurs in `file 1`. The second definition occurs in `file 2`. The `main` function references the object defined in `file 1`. The `var_print` function references the object defined in `file 2`:

Storage Class Specifiers

File 1

```
/******  
** Program to illustrate file scope static variables **  
*****/  
  
#include <stdio.h>  
  
extern void var_print(void);  
static stat_var = 1;  
  
int main(void)  
{  
    printf("file1 stat_var = %d\n", stat_var);  
    var_print();  
    printf("FILE1 stat_var = %d\n", stat_var);  
  
    return(0);  
}
```

File 2

```
/******  
** This file contains the second definition of stat_var **  
*****/  
  
#include <stdio.h>  
  
static int stat_var = 2;  
  
void var_print(void)  
{  
    printf("file2 stat_var = %d\n", stat_var);  
}
```

This program produces the following output:

```
file1 stat_var = 1  
file2 stat_var = 2  
FILE1 stat_var = 1
```

Storage Class Specifiers

The following program shows the linkage of **static** identifiers with block scope. The function **test** defines the static variable `stat_var`, which retains its storage throughout the program, even though `test` is the only function that can refer to `stat_var`.

```
/******  
** Program to illustrate block scope static variables          **  
*****/  
  
#include <stdio.h>  
  
int main(void)  
{  
    void test(void);  
    int counter;  
    for (counter = 1; counter <= 4; ++counter)  
        test();  
  
    return(0);  
}  
  
void test(void)  
{  
    static int stat_var = 0;  
    auto int auto_var = 0;  
    stat_var++;  
    auto_var++;  
    printf("stat_var = %d auto_var = %d\n", stat_var, auto_var);  
}
```

This program produces the following output:

```
stat_var = 1 auto_var = 1  
stat_var = 2 auto_var = 1  
stat_var = 3 auto_var = 1  
stat_var = 4 auto_var = 1
```

Related Information

- “Function Definitions” on page 143
- “Function Declarator” on page 145

Storage Class Specifiers

typedef

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. A **typedef** declaration does not reserve storage. The names you define using **typedef** are not new data types. They are synonyms for the data types or combinations of data types they represent.

The syntax of a **typedef** declaration is:

```
▶—typedef—type_specifier—identifier—;—◀
```

When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

C++ Note:

A C++ class defined in a **typedef** without being named is given a dummy name and the **typedef** name for linkage. Such a class cannot have constructors or destructors. For example:

```
typedef class {  
    Trees();  
} Trees;
```

Here the function `Trees()` is an ordinary member function of a class whose type name is unspecified. In the above example, `Trees` is an alias for the unnamed class, not the class type name itself, so `Trees()` cannot be a constructor for that class.

Examples of typedef Declarations

The following statements declare `LENGTH` as a synonym for **int** and then use this **typedef** to declare `length`, `width`, and `height` as integral variables:

```
typedef int LENGTH;  
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, **typedef** can be used to define a class type (structure, union, or C++ class). For example:

```
typedef struct {  
    int scruples;  
    int drams;  
    int grains;  
} WEIGHT;
```

The structure `WEIGHT` can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

Type Specifiers

Related Information

- “Characters”
- “Floating-Point Variables” on page 48
- “Integer Variables” on page 49
- “Enumerations” on page 50
- “Pointers” on page 55
- “void Type” on page 60
- “Arrays” on page 61
- “Structures” on page 70
- “Unions” on page 78
- Chapter 9, “Classes” on page 239
- “Constructors and Destructors Overview” on page 293

Type Specifiers

Type specifiers indicate the type of the object or function being declared. The fundamental data types are:

- Characters
- Floating-Point Numbers
- Integers
- Enumerations
- Void

From these types, you can derive:

- Pointers
- Arrays
- Structures
- Unions
- Functions

The integral types are **char**, **wchar_t** and **int** of all sizes. Floating-point numbers can have types **float**, **double**, or **long double**. Integral and floating-point types are collectively called *arithmetic* types. In *C++ only*, you can also derive the following:

- References
- Classes
- Pointers to Members

In C++, enumerations are not an integral type, but they can be subject to *integral promotion*, as described in “Integral Promotions” on page 131.

You can give names to both fundamental and derived types by using the **typedef** specifier.

Characters

There are three character data types: **char**, **signed char**, and **unsigned char**. These three data types are not compatible.

Type Specifiers

The character data types provide enough storage to hold any member of the character set used at run time. The amount of storage allocated for a **char** is implementation-dependent. The VisualAge for C++ for AS/400 compiler represents a character by 8 bits, as defined in the `CHAR_BIT` macro in the `<limits.h>` header.

The default character type behaves like an **unsigned char**. To change this default, use **#pragma chars**, described on page 215.

If it does not matter whether a **char** data object is **signed** or **unsigned**, you can declare the object as having the data type **char**; otherwise, explicitly declare **signed char** or **unsigned char**. When a **char** (**signed** or **unsigned**) is widened to an **int**, its value is preserved.

To declare a data object having a character type, use a **char** type specifier. The **char** specifier has the form:



The declarator for a simple character declaration is an identifier. You can initialize a simple character with a character constant or with an expression that evaluates to an integer.

Use the **char** specifier in variable definitions to define such variables as: arrays of characters, pointers to characters, and arrays of pointers to characters. Use **signed char** or **unsigned char** to declare numeric variables that occupy a single byte.

Type Specifiers

C++ Note: For the purposes of distinguishing overloaded functions, a C++ **char** is a distinct type from **signed char** and **unsigned char**.

Examples of Character Data Types

The following example defines the identifier `end_of_string` as a constant object of type **char** having the initial value `\0` (the null character):

```
const char end_of_string = '\0';
```

The following example defines the **unsigned char** variable `switches` as having the initial value 3:

```
unsigned char switches = 3;
```

The following example defines `string_pointer` as a pointer to a character:

```
char *string_pointer;
```

The following example defines `name` as a pointer to a character. After initialization, `name` points to the first letter in the character string "Johnny":

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are a pointer to the string "Venus", a pointer to "Jupiter", and a pointer to "Saturn":

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

Related Information

- "Character Constants" on page 23
- "Pointers" on page 55
- "Arrays" on page 61
- "Assignment Expressions" on page 126

Type Specifiers

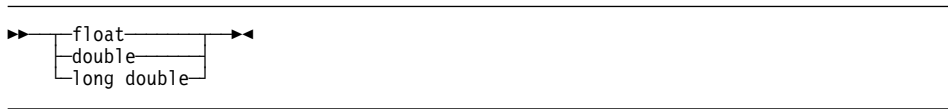
Floating-Point Variables

There are three types of floating-point variables: **float**, **double**, and **long double**.

The amount of storage allocated for a **float**, a **double**, or a **long double** is implementation-dependent. On all compilers, the storage size of a **float** variable is less than or equal to the storage size of a **double** variable.

To declare a data object having a floating-point type, use the *float specifier*.

The **float** specifier has the form:



The declarator for a simple floating-point declaration is an identifier. Initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

Examples of Floating-Point Data Types

The following example defines the identifier `pi` as an object of type **double**:

```
double pi;
```

The following example defines the **float** variable `real_number` with the initial value 100.55:

```
static float real_number = 100.55f;
```

The following example defines the **float** variable `float_var` with the initial value 0.0143:

```
float float_var = 1.43e-2f;
```

The following example declares the **long double** variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type **double**:

```
double table[20];
```

Related Information

- “Floating-Point Constants” on page 22
- “Assignment Expressions” on page 126

Type Specifiers

Integer Variables

Integer variables fall into the following categories:

- **short int** or **short** or **signed short int** or **signed short**
- **signed int** or **int**
- **long int** or **long** or **signed long int** or **signed long**
- **unsigned short int** or **unsigned short**
- **unsigned** or **unsigned int**
- **unsigned long int** or **unsigned long**

The default integer type for a bit field is **unsigned**.

Use the **bitfields=signed** option to change this default.

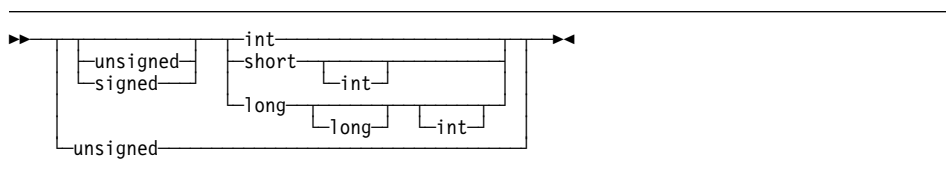
The amount of storage allocated for integer data is implementation-dependent.

Two sizes of integer data types are provided. Objects having type **short** are 2 bytes of storage long. Objects having type **long** are 4 bytes of storage long. An **int** represents the most efficient data storage size on the system (the word-size of the machine) and receives 4 bytes of storage.

The **unsigned** prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, **int** reserves the same storage as **unsigned int**. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

To declare a data object having an integer data type, use an **int** type specifier.

The **int** specifier has the form:



The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

C++ Note: When the arguments in overloaded functions and overloaded operators are integer types, two integer types that both come from the same group are not treated as distinct types. For example, you cannot overload an **int** argument against a **signed int** argument. Overloading and argument matching is described in Chapter 11, “Overloading” on page 277.

Type Specifiers

Examples of Integer Data Types

The following example defines the **short int** variable `flag`:

```
short int flag;
```

The following example defines the **int** variable `result`:

```
int result;
```

The following example defines the **unsigned long int** variable `ss_number` as having the initial value 438888834:

```
unsigned long ss_number = 438888834ul;
```

The following example defines the identifier `sum` as an object of type **int**. The initial value of `sum` is the result of the expression `a + b`:

```
extern int a, b;  
auto sum = a + b;
```

Related Information

- “Integer Constants” on page 20
- “Decimal Constants” on page 21
- “Octal Constants” on page 22
- “Hexadecimal Constants” on page 21

Enumerations

An *enumeration* data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can declare an enumeration type separately from the definition of variables of that type. The identifier associated with the data type (not an object) is called an *enumeration tag*.

C++ Note: In C, an enumeration has an implementation-defined integral type. This restriction does not apply to C++. In C++, an enumeration has a distinct type that does not have to be integral.

Declaring an Enumeration Data Type

An enumeration type declaration contains the **enum** keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. Commas separate each enumerator in the enumerator list. In extended mode, a trailing comma is permitted at the end of the enumerator list.

```
enum [identifier] { [enumerator] };
```

The keyword **enum**, followed by the identifier, names the data type (like the tag on a **struct** data type). The list of enumerators provides the data type with a set of values.

Type Specifiers

C++ Note: In C, each enumerator represents an integer value. In C++, each enumerator represents a value that can be converted to an integral value.

An enumerator has the form:

►—*identifier*—┐
└—*integral_constant_expression*—◄

To conserve space, enumerations may be stored in spaces smaller than that of an **int**.

Enumeration Constants

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an *enumeration constant*.

The value of the constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

In C, enumeration constants have type **int**.

In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. Use an enumeration constant anywhere an integer constant is allowed, or for C++, anywhere a value of the enumeration type is allowed.

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the declarations of `average` on line 4 and of `poor` on line 5 cause compiler error messages:

```
1 func()
2 {
3     enum score { poor, average, good };
4     enum rating { below, average, above };
5     int poor;
6 }
```

Type Specifiers

The following data type declarations list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };
/*      0      1      2      3      4      */
```

```
enum grain { oats=1, wheat, barley, corn, rice };
/*      1      2      3      4      5      */
```

```
enum grain { oats, wheat=10, barley, corn=20, rice };
/*      0      10      11      20      21 */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers `suspend` and `hold` have the same integer value.

```
enum status { run, clear=5, suspend, resume, hold=6 };
/*      0      5      6      7      6      */
```

The following example is a different declaration of the enumeration tag `status`:

```
enum status { run, create, clear=5, suspend };
/*      0      1      5      6      */
```

Defining Enumeration Variables

An enumeration variable definition contains an optional storage class specifier, a type specifier, a declarator, and an optional initializer. The type specifier contains the keyword **enum** followed by the name of the enumeration data type. You must declare the enumeration data type before you can define a variable having that type.

The initializer for an enumeration variable contains the = symbol followed by an expression. In C++, the initializer must be have the same type as the associated enumeration type.

The first line of the following example declares the enumeration tag `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (2).

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

In C, the type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`. In C++, the value of `g_food` has the enumerated data type `grain`.

C++ also makes the **enum** keyword optional in an initialization expression like the one in the second line of the preceding example. For example, both of the following statements are valid C++ code:

```
enum grain g_food = barley;
    grain cob_food = corn;
```

Type Specifiers

Defining an Enumeration Type and Enumeration Objects

You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

C++ also lets you put the storage class immediately before the declarator. For example:

```
enum score { poor=1, average, good } register rating = good;
```

Either of these examples is equivalent to the following two declarations:

```
enum score { poor=1, average, good };  
register enum score rating = good;
```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier **register**, the data type `enum score`, and the initial value `good`.

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,  
      Saturday } weekday;
```

defines the variable `weekday`, which can be assigned any of the specified enumeration constants.

Example Program Using Enumerations

The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints "C'est le mauvais jour."

```
/**  
 ** Example program using enumerations  
 **/  
  
#include <stdio.h>  
  
enum days {  
    Monday=1, Tuesday, Wednesday,  
    Thursday, Friday, Saturday, Sunday  
} weekday;  
  
void french(enum days);  
  
int main(void)  
{  
    int num;
```


Type Specifiers

```
printf("Enter an integer for the day of the week. "
      "Mon=1,...,Sun=7\n");
scanf("%d", &num);
weekday=num;
french(weekday);
return(0);
}

void french(enum days weekday)
{
    switch (weekday)
    {
        case Monday:
            printf("Le jour de la semaine est lundi.\n");
            break;
        case Tuesday:
            printf("Le jour de la semaine est mardi.\n");
            break;
        case Wednesday:
            printf("Le jour de la semaine est mercredi.\n");
            break;
        case Thursday:
            printf("Le jour de la semaine est jeudi.\n");
            break;
        case Friday:
            printf("Le jour de la semaine est vendredi.\n");
            break;
        case Saturday:
            printf("Le jour de la semaine est samedi.\n");
            break;
        case Sunday:
            printf("Le jour de la semaine est dimanche.\n");
            break;
        default:
            printf("C'est le mauvais jour.\n");
    }
}
```

Related Information

- "Identifiers" on page 18
- "Enumeration Constants" on page 51
- "Constant Expressions" on page 99

Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type except to a bit field or a reference. Some common uses for pointers are:

- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure or C++ class.
- To access an array of characters as a string.
- To pass the address of a variable to a function. (In C++, you can also use a reference to do this.) By referencing a variable through its address, a function can change the contents of that variable. “Calling Functions and Passing Arguments” on page 150 describes passing arguments by reference.

Declaring Pointers

The following example declares `pcoat` as a pointer to an object having type **long**:

```
extern long *pcoat;
```

If the keyword **volatile** appears before the `*`, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** comes between the `*` and the identifier, the declarator describes a **volatile** pointer. The keyword **const** operates in the same manner as the **volatile** keyword described. In the following example, `pvolt` is a constant pointer to an object having type **short**:

```
short * const pvolt;
```

The following example declares `pnut` as a pointer to an **int** object having the **volatile** qualifier:

```
extern int volatile *pnut;
```

The following example defines `psoup` as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines `pfowl` as a pointer to an enumeration object of type `bird`:

```
enum bird *pfowl;
```

The next example declares `pvish` as a pointer to a function that takes no parameters and returns a **char** object:

```
char (*pvish)(void);
```

Type Specifiers

Assigning Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;
float * sub_ptr;
.
.
.
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;
int * minor;
.
.
.
minor = &league;    /* error */
```

Initializing Pointers

The initializer is an = (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables `time` and `speed` as having type **double** and `amount` as having type pointer to a **double**. The pointer `amount` is initialized to point to `total`:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in section:

```
int section[80];
int *student = section;
```

is equivalent to:

```
int section[80];
int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable `string` and the string constant "abcd". The pointer `string` is initialized to point to the character `a` in the string "abcd".

```
char *string = "abcd";
```

Type Specifiers

The following example defines `weekdays` as an array of pointers to string constants. Each element points to a different string. The pointer `weekdays[2]`, for example, points to the string "Tuesday".

```
static char *weekdays[ ] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

A pointer can also be initialized to `NULL` using any integer constant expression that evaluates to `0`, for example `char * a=0;`. Such a pointer is a *NULL pointer*. It does not point to any object.

Restrictions on Pointers

You cannot use pointers to reference bit fields or objects having the **register** storage class specifier.

Using Pointers

Two operators are commonly used in working with pointers, the address (&) operator and the indirection (*) operator. You can use the & operator to refer to the address of an object. For example, the following statement assigns the address of `x` to the variable `p_to_x`. The variable `p_to_x` has been defined as a pointer.

```
int x, *p_to_x;

p_to_x = &x;
```

The * (indirection) operator lets you access the value of the object a pointer refers to. The following statement assigns to `y` the value of the object that `p_to_x` points to:

```
float y, *p_to_x;
.
.
.
y = *p_to_x;
```

The following statement assigns the value of `y` to the variable that `*p_to_x` references:

```
char y ,
    *p_to_x,
.
.
.
*p_to_x = y;
```

Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and decrement

Type Specifiers

- Addition and subtraction
- Comparison
- Assignment

The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add a pointer to an integer, but you cannot add a pointer to a pointer.

If the pointer `p` points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`. See Chapter 4, “Expressions and Operators” on page 93 for more information on these operators.

Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the `==` and `!=` operators can be performed even when the pointers point to elements of different arrays.

You can assign to a pointer the address of a data object, the value of another compatible pointer or the NULL pointer.

Example Program Using Pointers

The following program contains pointer arrays:

```
/* *****  
** Program to search for the first occurrence of a specified **  
** character string in an array of character strings. **  
/* *****/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
#define SIZE 20  
#define EXIT_FAILURE 999  
  
int main(void)  
{
```

Type Specifiers

```
static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
char * find_name(char **, char *);
char new_name[SIZE], *name_pointer;

printf("Enter name to be searched.\n");
scanf("%s", new_name);
name_pointer = find_name(names, new_name);
printf("name %s%sfound\n", new_name,
       (name_pointer == NULL) ? " not " : " ");
exit(EXIT_FAILURE);
} /* End of main */

/*****
**      Function find_name. This function searches an array of
**      names to see if a given name already exists in the array.
**      It returns a pointer to the name or NULL if the name is
**      not found.
**
** char **array is a pointer to arrays of pointers (existing names)
** char *strng is a pointer to character array entered (new name)
*****/

char * find_name(char **array, char *strng)
{
    for (; *array != NULL; array++)          /* for each name      */
    {
        if (strcmp(*array, strng) == 0)     /* if strings match     */
            return(*array);                 /* found it!            */
    }
    return(*array);                          /* return the pointer   */
} /* End of find_name */
```

Interaction with this program could produce the following sessions:

Output Enter name to be searched.

Input Mark

Output name Mark found

OR:

Output Enter name to be searched.

Input Deborah

Output name Deborah not found

Related Information

- "Declarators" on page 83
- "volatile and const Qualifiers" on page 84
- "Initializers" on page 88
- "Address &" on page 106

Type Specifiers

- “Indirection *****” on page 106

void Type

The **void** data type always represents an empty set of values. The only object that can be declared with the type specifier **void** is a pointer.

When a function does not return a value, you should use **void** as the type specifier in the function definition and declaration. An argument list for a function taking no arguments is **void**.

You cannot declare a variable of type **void**, but you can explicitly convert any expression to type **void**, but the resulting expression can only be used as one of the following:

- An expression statement
- The left operand of a comma expression
- The second or third operand in a conditional expression.

Example of void Type

On line 7 of the following example, the function `find_max` is declared as having type **void**. Lines 15 through 26 contain the complete definition of `find_max`.

Note: The use of the `sizeof` operator in line 13 is a standard method of determining the number of elements in an array.

Type Specifiers

```
1  /**
2  ** Example of void type
3  **/
4  #include <stdio.h>
5
6  /* declaration of function find_max */
7  extern void find_max(int x[ ], int j);
8
9  int main(void)
10 {
11     static int numbers[ ] = { 99, 54, -102, 89 };
12
13     find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));
14
15     return(0);
16 }
17
18 void find_max(int x[ ], int j)
19 { /* begin definition of function find_max */
20     int i, temp = x[0];
21
22     for (i = 1; i < j; i++)
23     {
24         if (x[i] > temp)
25             temp = x[i];
26     }
27     printf("max number = %d\n", temp);
28 } /* end definition of function find_max */
```

Arrays

An *array* is an ordered group of data objects. Each object is called an *element*. All elements within an array have the same data type.

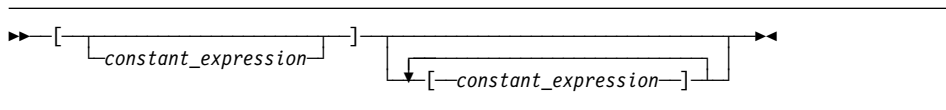
Use any type specifier in an array definition or declaration. Array elements can be of any data type, except function or, in C++, a reference. You can, however, declare an array of pointers to functions.

Type Specifiers

Declaring Arrays

The array declarator contains an identifier followed by an optional *subscript declarator*. An identifier preceded by an * (asterisk) is an array of pointers.

A subscript declarator has the form:



The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. Each bracketed expression, or subscript, describes a different dimension and must be a constant expression.

The following example defines a one-dimensional array that contains four elements having type **char**:

```
char list[4];
```

The first subscript of each dimension is 0. The array `list` contains the elements:

```
list[0]  
list[1]  
list[2]  
list[3]
```

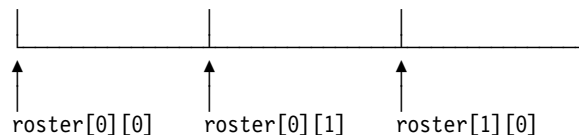
The following example defines a two-dimensional array that contains six elements of type **int**:

```
int roster[3][2];
```

Multidimensional arrays are stored in row-major order. When elements are referred to in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]  
roster[0][1]  
roster[1][0]  
roster[1][1]  
roster[2][0]  
roster[2][1]
```

In storage, the elements of `roster` would be stored as:



You can leave the first (and only the first) set of subscript brackets empty in

Type Specifiers

- Array definitions that contain initializations
- **extern** declarations
- Parameter declarations.

In array definitions that leave the first set of subscript brackets empty, the initializer determines the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the initializer is compared to the subscript declarator to determine the number of elements in the first dimension.

An unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array, provided the array has previously been declared. An unsubscripted array name with square brackets (for example, `region[]`) is allowed in the following contexts:

- In arrays declared at file scope
- In the argument list of a function declaration

In declarations, only the first dimension can be left empty; you must specify the sizes of additional dimensions.

In extended modes, they are also permitted in the following contexts:

- In union members
- As the last member of a structure

Whenever an array is used in a context (such as a parameter) where it cannot be used as an array, the identifier is treated as a pointer. The two exceptions are when an array is used as an operand of the **sizeof** or the address (&) operator.

Initializing Arrays

The initializer for an array contains the = symbol followed by a comma-separated list of constant expressions enclosed in braces ({ }). You do not need to initialize all elements in an array. Elements that are not initialized (in **extern** and **static** definitions only) receive the value 0 of the appropriate type.

Note: Array initializations can be either *fully braced* (with braces around each dimension) or *unbraced* (with only one set of braces enclosing the entire set of initializers). Avoid placing braces around some dimensions and not around others.

Type Specifiers

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values:

Element	Value
<code>number[0]</code>	5
<code>number[1]</code>	7
<code>number[2]</code>	2

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:

Element	Value
<code>number1[0]</code>	5
<code>number1[1]</code>	7
<code>number1[2]</code>	0

Instead of an expression in the subscript declarator defining the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements:

Element	Value
<code>item[0]</code>	1
<code>item[1]</code>	2
<code>item[2]</code>	3
<code>item[3]</code>	4
<code>item[4]</code>	5

You can initialize a one-dimensional character array by specifying:

- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant. (Braces surrounding the constant are optional.)

Initializing a string constant places the null character (`\0`) at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

```
static char name1[ ] = { 'J', 'a', 'n' };  
static char name2[ ] = { "Jan" };  
static char name3[4] = "Jan";
```

These definitions create the following elements:

Type Specifiers

Element	Value	Element	Value	Element	Value
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	\0	name3[3]	\0

Note that the following definition would result in the null character being lost:

```
static char name[3]="Jan";
```

In C, a compiler warning is issued for name3[3]. In C++, the compiler issues a severe error for name3[3].

You can initialize a multidimensional array by:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array month_days:

```
static month_days[2][12] =
{
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension. (You can consider these elements as rows.) The initialization contains braces around each of these two elements:

```
static int month_days[2][12] =
{
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

- Using nested braces to initialize dimensions and elements in a dimension selectively.

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
{
    {1, 2},
    {3, 4},
    {5, 6}
};
```

Type Specifiers

The initial values of matrix are:

Element	Value	Element	Value
matrix[0][0]	1	matrix[1][2]	0
matrix[0][1]	2	matrix[1][3]	0
matrix[0][2]	0	matrix[2][0]	5
matrix[0][3]	0	matrix[2][1]	6
matrix[1][0]	3	matrix[2][2]	0
matrix[1][1]	4	matrix[2][3]	0

You cannot have more initializers than the number of elements in the array.

Zero-Extent Arrays in Structures and Unions

Zero-extent arrays are permitted in structures and unions under **extended** mode.

For example, Struct_A contains the zero-extent array char arr_a[0], and Struct_B contains the zero-extent array char arr_b[[]].

```
struct Struct_A {
    int i;
    char arr_a[0];
} a;
```

```
struct Struct_B {
    int k;
    char arr_b[[]];
} a;
```

Both char arr_a[0]; and char arr_b[[]]; are implicitly viewed by the compiler as if declared as arrays with one element:

```
struct Struct_A {
    int i;
    char arr_a[1];
} a;
```

```
struct Struct_B {
    int k;
    char arr_b[1];
} b;
```

The zero-sized array can be any member of a union. Zero-sized arrays are only allowed as the *last* member of a structure.

The **sizeof** operator assumes 1 for the zero-sized array.

Structures containing zero-sized arrays can be nested within other structures as long as no additional members are subsequently declared in any enclosing structures. For example, the following declaration causes an error:

Type Specifiers

```
struct Struct_S {
    int i;
    char arr_s[];
};

struct Struct_T {
    float f;
    struct Struct_S;
    int k;    // Error - additional structure member following a
             // struct containing a zero-sized array
} t;
```

In C++, the zero-sized array must be non-static. In a class definition, the zero-sized array must be the last non-static data member. Members such as functions, static data members, and typedefs are allowed after the zero-sized array. A class containing a zero-sized array cannot be used as a base class.

Example Programs Using Arrays

The following program defines a floating-point array called `prices`.

The first `for` statement prints the values of the elements of `prices`. The second `for` statement adds five percent to the value of each element of `prices`, and assigns the result to `total`, and prints the value of `total`.

Type Specifiers

```
/**
** Example of one-dimensional arrays
**/

#include <stdio.h>
#define ARR_SIZE 5

int main(void)
{
    static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
    auto float total;
    int i;

    for (i = 0; i < ARR_SIZE; i++)
    {
        printf("price = $%.2f\n", prices[i]);
    }

    printf("\n");

    for (i = 0; i < ARR_SIZE; i++)
    {
        total = prices[i] * 1.05;

        printf("total = $%.2f\n", total);
    }

    return(0);
}
```

This program produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

Type Specifiers

The following program defines the multidimensional array `salary_tbl`. A `for` loop prints the values of `salary_tbl`.

```
/**
 ** Example of a multidimensional array
 **/

#include <stdio.h>
#define ROW_SIZE 3
#define COLUMN_SIZE 5

int main(void)
{
    static int salary_tbl[ROW_SIZE][COLUMN_SIZE] =
    {
        { 500, 550, 600, 650, 700 },
        { 600, 670, 740, 810, 880 },
        { 740, 840, 940, 1040, 1140 }
    };
    int grade , step;

    for (grade = 0; grade < ROW_SIZE; ++grade)
        for (step = 0; step < COLUMN_SIZE; ++step)
        {
            printf("salary_tbl[%d] [%d] = %d\n", grade, step,
                salary_tbl[grade] [step]);
        }

    return(0);
}
```

This program produces the following output:

```
salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

Related Information

- “Pointers” on page 55
- “Array Subscript [] (Array Element Specification)” on page 102

Type Specifiers

- “String Literals” on page 24
- “Declarators” on page 83
- “Initializers” on page 88
- Chapter 5, “Implicit Type Conversions” on page 131

Structures

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a *member* or *field*.

Use structures to group logically related objects. For example, to allocate storage for the components of one address, define the following variables:

```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

To allocate storage for more than one address, group the components of each address by defining a structure data type and as many variables as you need to have the structure data type.

In the following example, lines 1 through 7 declare the structure tag `address`:

```
1 struct address {
2     int street_no;
3     char *street_name;
4     char *city;
5     char *prov;
6     char *postal_code;
7 };
8 struct address perm_address;
9 struct address temp_address;
10 struct address *p_perm_address = &perm_address;
```

The variables `perm_address` and `temp_address` are instances of the structure data type `address`. Both contain the members described in the declaration of `address`. The pointer `p_perm_address` points to a structure of `address` and is initialized to point to `perm_address`.

Refer to a member of a structure by specifying the structure variable name with the dot operator (`.`) or a pointer with the arrow operator (`->`) and the member name. For example, both of the following:

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

assign a pointer to the string "Ontario" to the pointer `prov` that is in the structure `perm_address`.

Type Specifiers

All references to structures must be fully qualified. In the example, you cannot reference the fourth field by `prov` alone. You must reference this field by `perm_address.prov`.

Structures with identical members but different names are not compatible and cannot be assigned to each other.

Structures are not intended to conserve storage. If you need direct control of byte mapping, use pointers.

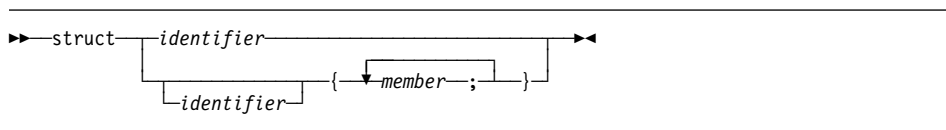
Structure member references are described in “Dot Operator `.`” on page 102 and “Arrow Operator `->`” on page 103

You cannot declare a structure with members of incomplete types. See “Incomplete Types” on page 83 for more information.

Declaring a Structure

A structure type declaration describes the members that are part of the structure. It contains the **struct** keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

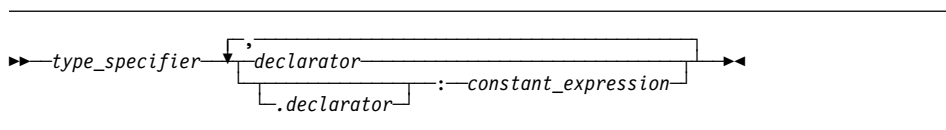
A structure declaration has the form:



The keyword **struct** followed by the identifier (tag) names the data type. If you do not provide a tag name to the data type, you must put all variable definitions that refer to it within the declaration of the data type.

The list of members provides the data type with a description of the values that can be stored in the structure.

A structure member definition has the form:



If a `:` (colon) and a constant expression follow the member declarator, the member represents a *bit field*. A member that does not represent a bit field can be of any data type and can have the **volatile** or **const** qualifier. Bit fields are described in “Declaring and Using Bit Fields in Structures” on page 74.

Type Specifiers

Identifiers used as structure or member names can be redefined to represent different objects in the same scope without conflicting. You cannot use the name of a member more than once in a structure type, but you can use the same member name in another structure type that is defined within the same scope.

You cannot declare a structure type that contains itself as a member, but you can declare a structure type that contains a pointer to itself as a member.

Defining a Structure Variable

A structure variable definition contains an optional storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable. The keyword **struct** is optional in C++.

You can declare structures having any storage class. Most compilers, however, treat structures declared with the **register** storage class specifier as automatic structures.

Initializing Structures

The initializer contains an = (equal sign) followed by a brace-enclosed comma-separated list of values. You do not have to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
static struct address perm_address =
    { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

The values of perm_address are:

Member	Value
perm_address.street_no	3
perm_address.street_name	address of string "Savona Dr."
perm_address.city	address of string "Dundas"
perm_address.prov	address of string "Ontario"
perm_address.postal_code	address of string "L4B 2A1"

The following definition shows a partially initialized structure:

Type Specifiers

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address temp_address =
    { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

The values of temp_address are:

Member	Value
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"
temp_address.postal_code	value depends on the storage class.

Note: The initial value of uninitialized structure members like temp_address.postal_code depends on the storage class associated with the member. See “Storage Class Specifiers” on page 31 for details on the initialization of different storage classes.

Declaring Structure Types and Variables

To define a structure type and a structure variable in one statement, put a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the statement.

For example:

```
static struct {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
} perm_address, temp_address;
```

Because this example does not name the structure data type, perm_address and temp_address are the only structure variables that will have this data type. Putting an identifier after **struct**, lets you make additional variable definitions of this data type later in the program.

The structure type (or tag) cannot have the **volatile** qualifier, but a member or a structure variable can be defined as having the **volatile** qualifier.

Type Specifiers

For example:

```
static struct class1 {
    char    descript[20];
    volatile long code;
    short complete;
} volatile file1, file2;
struct class1 subfile;
```

This example qualifies the structures `file1` and `file2`, and the structure member `subfile.code` as **volatile**.

Declaring and Using Bit Fields in Structures

A structure or a C++ class can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon. The constant expression specifies how many bits the field reserves.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field causes the next field to be aligned on the next container boundary, where the container is the same size as the underlying type as the bit field. The padding to the next container boundary only takes place if the zero-width bit field has the same underlying type as the preceding bit-field member. If the types are different, the zero-width bit field has no effect.

The maximum bit-field length is implementation dependent.

For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

In C, you can declare a bit field as type **int**, **signed int**, or **unsigned int**. Bit fields of the type **int** are equivalent to those of type **unsigned int**.

The default integer type for a bit field is **unsigned**. Use the **bitfields=signed** option to change this default.

In extended mode C, bit fields can be any integral type. For example,

```
struct S {
    short x : 4;
    long y : 10;
    char z : 7;
} s;
```

Type Specifiers

Non-integral bit fields in extended mode C are converted to type **unsigned int** and a warning is issued. In other modes, the use of non-integral bit fields results in an error.

A bit field cannot have the **const** or **volatile** qualifier.

The following structure has three bit-field members `kingdom`, `phylum`, and `genus`, occupying 12, 6, and 2 bits respectively:

```
struct taxonomy {
    int kingdom : 12;
    int phylum : 6;
    int genus : 2;
};
```

Unlike ANSI/ISO C, C++ bit fields can be any integral type or enumeration type. When you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.

If a series of bit fields does not add up to the size of an **int**, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. Bit fields cannot cross word boundaries but are forced to start at the next word boundary.

The following example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {
    unsigned light : 1;
    unsigned toaster : 1;
    int count;          /* 4 bytes */
    unsigned ac : 4;
    unsigned : 4;
    unsigned clock : 1;
    unsigned : 0;
    unsigned flag : 1;
} kitchen ;
```

The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
<code>light</code>	1 bit
<code>toaster</code>	1 bit
(padding — 30 bits)	To next int boundary
<code>count</code>	The size of an int
<code>ac</code>	4 bits
(unnamed field)	4 bits
<code>clock</code>	1 bit
(padding — 23 bits)	To next int boundary (unnamed field)
<code>flag</code>	1 bit
(padding — 31 bits)	To next int boundary

Type Specifiers

All references to structure fields must be fully qualified. For instance, you cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.

The following expression sets the `light` field to 1:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the `toaster` field of the `kitchen` structure to 0 because only the least significant bit is assigned to the `toaster` field:

```
kitchen.toaster = 2;
```

Declaring a Packed Structure

Data elements of a structure are stored in memory on an address boundary specific for that data type. For example, a **double** value is stored in memory on a doubleword (8-byte) boundary. Gaps may be left in memory between elements of a structure to align elements on their natural boundaries.

C++ Note: C++ does support the **_Packed** type qualifier to allow you to move your existing ILE C applications to C++. For new C++ applications you should change the alignment of structures using the **#pragma pack** directive or the `/Sp` compiler option.

Example Program Using Structures

The following program finds the sum of the integer numbers in a linked list:

Type Specifiers

```
/**
** Example program illustrating structures using linked lists
**/

#include <stdio.h>

struct record {
    int number;
    struct record *next_num;
};

int main(void)
{
    struct record name1, name2, name3;
    struct record *recd_pointer = &name1;
    int sum = 0;

    name1.number = 144;
    name2.number = 203;
    name3.number = 488;

    name1.next_num = &name2;
    name2.next_num = &name3;
    name3.next_num = NULL;

    while (recd_pointer != NULL)
    {
        sum += recd_pointer->number;
        recd_pointer = recd_pointer->next_num;
    }
    printf("Sum = %d\n", sum);

    return(0);
}
```

The structure type `record` contains two members: the integer `number` and `next_num`, which is a pointer to a structure variable of type `record`.

Type Specifiers

The record type variables `name1`, `name2`, and `name3` are assigned the following values:

Member Name	Value
<code>name1.number</code>	144
<code>name1.next_num</code>	The address of <code>name2</code>
<code>name2.number</code>	203
<code>name2.next_num</code>	The address of <code>name3</code>
<code>name3.number</code>	488
<code>name3.next_num</code>	NULL (Indicating the end of the linked list.)

The variable `recd_pointer` is a pointer to a structure of type `record`. It is initialized to the address of `name1` (the beginning of the linked list).

The **while** loop causes the linked list to be scanned until `recd_pointer` equals NULL. The statement:

```
recd_pointer = recd_pointer->next_num;
```

advances the pointer to the next object in the list.

Related Information

- “Declarators” on page 83
- “Initializers” on page 88
- “Incomplete Types” on page 83
- “Dot Operator `.`” on page 102
- “Arrow Operator `->`” on page 103

Unions

A *union* is an object that can hold any one of a set of named members. The members of the named set can be of any data type. Members are overlaid in storage.

The storage allocated for a union is the storage required for the largest member of the union (plus any padding that is required so that the union will end at a natural boundary of its strictest member).

In C++, a union can have member functions, including constructors and destructors, but not virtual member functions. A union cannot be used as a base class and cannot be derived from a base class.

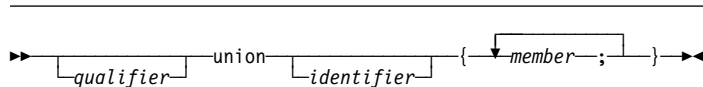
A C++ union member cannot be a class object that has a constructor, destructor, or overloaded copy assignment operator. In C++, a member of a union cannot be declared with the keyword `static`.

Declaring a Union

A union type declaration contains the **union** keyword followed by an identifier (optional) and a brace-enclosed list of members.

Type Specifiers

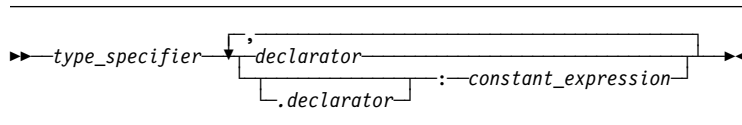
A union declaration has the form:



The *identifier* is a tag given to the union specified by the member list. If you specify a tag, any subsequent declaration of the union (in the same scope) can be made by declaring the tag and omitting the member list. If you do not specify a tag, you must put all variable definitions that refer to that union within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union.

A union member definition has the form:



You can reference one of the possible members of a union the same way as referencing a member of a structure.

For example:

```
union {
    char birthday[9];
    int age;
    float weight;
} people;
```

```
people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`.

A union can represent only one of its members at a time. In the example, the union `people` contains either `age`, `birthday`, or `weight` but never more than one of these. The `printf` statement in the following example does not give the correct result because `people.age` replaces the value assigned to `people.birthday` in the first line:

```
1 people.birthday = "03/06/56";
2 people.age = 38;
3 printf("%s\n", people.birthday);
```

Type Specifiers

Defining a Union Variable

A union variable definition contains an optional storage class keyword, the **union** keyword, a union tag, and a declarator. The union tag indicates the data type of the union variable.

Type Specifier: The type specifier contains the keyword **union** followed by the name of the union type. You must declare the union data type before you can define a union having that type.

You can define a union data type and a union of that type in the same statement by placing the variable declarator after the data type definition.

Declarator: The declarator is an identifier, possibly with the **volatile** or **const** qualifier.

Initializer: You can only initialize the first member of a union.

The following example shows how you would initialize the first union member birthday of the union variable `people`:

```
union {
    char birthday[9];
    int age;
    float weight;
} people = {"23/07/57"};
```

Defining a Union Type and a Union Variable

To define union type and a union variable in one statement, put a declarator after the type definition. The storage class specifier for the variable must go at the beginning of the statement.

Defining Packed Unions

To qualify a C union as packed, use **_Packed**. The **_Packed** qualifier is only supported in C.

Under the **_Packed** qualifier, the memory layout of the union members is not affected; Each member starts at offset zero. The **_Packed** qualifier does affect the total alignment restriction of the whole union.

C++ Note: C++ does not support the **_Packed** qualifier. To change the alignment of C++ unions, use the **#pragma pack** directive or the `/Sp` compiler option.

In the following example, each of the elements in the nonpacked `n_array` is of type union `uu`.

Type Specifiers

```
union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu n_array[2];
```

Because it is not packed, each element in the nonpacked `n_array` has an alignment restriction of 2 bytes (the largest alignment requirement among the union members is that of `short a`), and there is 1 byte of padding at the end of each element to enforce this requirement.

In the packed array `p_array`, each element is of type `_Packed union uu`. Because every element aligned on the byte boundary, each element has a length of only 3 bytes, instead of the 4 bytes in the previous example.

Anonymous Unions in C++

An *anonymous union* is a union without a class name. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object and it cannot have member functions.

The member names of an anonymous union must be distinct from other names within the scope in which the union is declared. You can use member names directly in the union scope without any additional member access syntax.

For example, in the following code fragment, you can access the data members `i` and `cptr` directly because they are in the scope containing the anonymous union. Because `i` and `cptr` are union members and have the same address, you should only use one of them at a time. The assignment to the member `cptr` will change the value of the member `i`.

```
void f()
{
    union { int i; char* cptr ; };
    // .
    // .
    // .
    i = 5;
    cptr = "string_in_union"; // overrides i
}
```

An anonymous union cannot have protected or private members. A global anonymous union must be declared with the keyword **static**.

Type Specifiers

Examples of Unions

The following example defines a union data type (not named) and a union variable (named `length`). The member of `length` can be a **long int**, a **float**, or a **double**.

```
union {
    float meters;
    double centimeters;
    long inches;
} length;
```

The following example defines the union type data as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two data type variables: `input` and `output`.

```
union data {
    char charctr;
    int whole;
    float real;
};
union data input, output;
```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a floating-point number to member `output`:

```
output.real = 9.2;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the union variable `input`. `input` has the union data type defined in the previous example.

```
struct {
    int id_num;
    int type_of_input;
    union data input;
} records[10];
```

The following statement assigns a character to the structure member `input` of the first element of **records**:

```
records[0].input.charctr = 'g';
```

Related Information

- “Declarators” on page 83
- “Initializers” on page 88
- “Structures” on page 70
- “Dot Operator `.`” on page 102
- “Arrow Operator `->`” on page 103

Incomplete Types

Incomplete types are the type **void**, an array of unknown size, or structure, union, or enumeration tags that have no member lists. For example, the following are incomplete types:

```
void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */
```

void is an incomplete type that cannot be completed. Incomplete structure or union and enumeration tags must be completed before being used to declare an object, although you can define a pointer to an incomplete structure or union.

Related Information

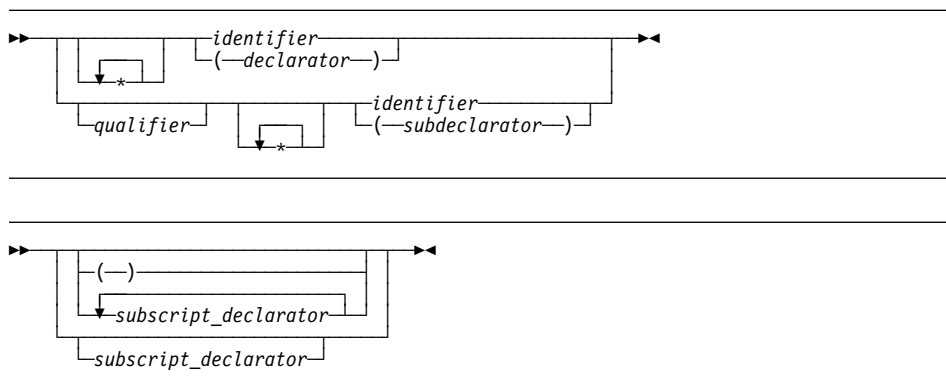
- “void Type” on page 60
- “Arrays” on page 61
- “Structures” on page 70
- “Unions” on page 78

Declarators

A *declarator* designates a data object or function. Declarators appear in all data definitions and declarations and in some type definitions.

In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can specify that the return type of a function is a pointer or a reference. You can also perform initialization in a declarator.

A declarator has the form:



Declarators

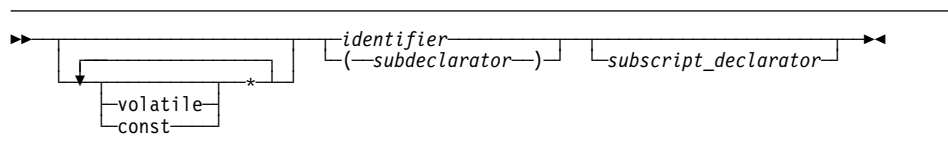
A qualifier is one of:

- **const**
- **volatile**

The VisualAge for C++ for AS/400 compiler also implements the **_Packed** type qualifier.

In C, you cannot declare or define a **volatile** or **const** function. C++ class member functions can be qualified with **const** or **volatile**.

A declarator can contain a *subdeclarator*. A subdeclarator has the form:



A *subscript declarator* describes the number of dimensions in an array and the number of elements in each dimension.

A simple declarator consists of an identifier, which names a data object. For example, the following block scope data declaration uses **initial** as the declarator:

```
auto char initial;
```

The data object **initial** has the storage class **auto** and the data type **char**.

You can define or declare a structure, union, or array. by using a declarator that contains an identifier, which names the data object, and some combination of symbols and identifiers, which describes the type of data that the object represents.

The following declaration uses `compute[5]` as the declarator:

```
extern long int compute[5];
```

volatile and const Qualifiers

The **volatile** qualifier maintains consistency of memory access to data objects. It tells the compiler that the variable should always contain its current value even when optimized, so that the variable can be queried when an exception occurs. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed.

The **volatile** qualifier is useful for data objects having values that may be changed in ways unknown to your program (such as the system clock). Portions of an expression that reference **volatile** objects are not to be changed or removed.

The **const** qualifier explicitly declares a data object as a data item that cannot be changed. Its value is set at initialization. You cannot use **const** data objects in expressions requiring a modifiable lvalue. For example, a **const** data object cannot appear on the left-hand side of an assignment statement.

Declarators

These type qualifiers are only meaningful in expressions that are lvalues.

For a **volatile** or **const** pointer, you must put the keyword between the ***** and the identifier. For example:

```
int * volatile x;      /* x is a volatile pointer to an int */
int * const y = &z;    /* y is a const pointer to the int variable z */
```

For a pointer to a **volatile** or **const** data object, the type specifier, qualifier, and storage class specifier can be in any order. For example:

```
volatile int *x;      /* x is a pointer to a volatile int */
or
int volatile *x;      /* x is a pointer to a volatile int */

const int *y;         /* y is a pointer to a const int */
or
int const *y;         /* y is a pointer to a const int */
```

In the following example, the pointer to *y* is a constant. You can change the value that *y* points to, but you cannot change the value of *y*:

```
int * const y
```

In the following example, the value that *y* points to is a constant integer and cannot be changed. However, you can change the value of *y*:

```
const int * y
```

For other types of **volatile** and **const** variables, the position of the keyword within the definition (or declaration) is less important. For example:

```
volatile struct omega {
    int limit;
    char code;
} group;
```

provides the same storage as:

```
struct omega {
    int limit;
    char code;
} volatile group;
```

In both examples, only the structure variable *group* receives the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the structure variable *group* receives the **const** qualifier. The **const** and **volatile** qualifiers when applied to a structure, union, or class also apply to the members of the structure, union, or class.

Declarators

Although enumeration, structure, and union variables can receive the **volatile** or **const** qualifier, enumeration, structure, and union tags do not carry the **volatile** or **const** qualifier. For example, the `blue` structure does not carry the **volatile** qualifier:

```
volatile struct whale {  
    int weight;  
    char name[8];  
} beluga;  
  
struct whale blue;
```

The keywords **volatile** and **const** cannot separate the keywords **enum**, **struct**, and **union** from their tags.

You can declare or define a **volatile** or **const** function only if it is a C++ member function. You can define or declare any function to return a pointer to a **volatile** or **const** function.

You can put more than one qualifier on a declaration but you cannot specify the same qualifier more than once on a declaration.

Packed Qualifier

Note: The VisualAge for C++ for AS/400 compiler supports the **_Packed** type qualifier.

The **_Packed** qualifier removes padding between members of structures and affects the alignment of unions whenever possible. However, the storage saved using packed structures and unions may come at the expense of runtime performance. Most machines access data more efficiently if it is aligned on appropriate boundaries. With packed structures and unions, members are generally not aligned on natural boundaries, and the result is that member-accessing operations (using the `.` and `->` operators) are slower.

Note: Pointers are always aligned on their natural boundaries, 16 bytes, even in **_Packed** structures and unions.

_Packed can only be used with structures or unions. If you use **_Packed** with other types, an error message is generated and the qualifier has no effect on the declarator it qualifies. Packed and nonpacked structures and unions have different storage layouts. Comparisons between packed and nonpacked structures or unions of the same type are prohibited.

If you specify the **_Packed** qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the contained structure or union. A C structure defined using the **_Packed** qualifier as follows:

Declarators

```
        struct record {
            int number;
            struct record *next_num;
        };

int main(void)
{
    _Packed struct record name1, name2, name3; //C allows packed and
    struct record *recd_pointer = &name1;    //nonpacked structures in
    int sum=0;                                //the same source
    .
    .
    .
}
```

can be rewritten to use the C++ **_Packed** type qualifier:

```
typedef _Packed struct record {
    int number;
    struct record *next_num;
}; record_type;

//C++ objects of a class
//or structure must be all
//packed or all nonpacked

int main(void)
{
    record_type struct record name1, name2, name3;
    record_type struct record *recd_pointer = &name1;
    int sum=0;
    .
    .
    .
}
```

New C++ programs should use the **#pragma pack** directive to remain portable across platforms. C++ does support the **_Packed** type qualifier to allow you to move your existing ILE C applications to C++.

The VisualAge for C++ for AS/400 compiler also lets you pack structures using the **#pragma pack** directive or the `/Sp` option. See "Pragma Directives (`#pragma`)" on page 213 for more information on **#pragma pack**.

Example Declarators

The following table describes some declarators:

Table 5 (Page 1 of 2). Example Declarators

Example	Description
<code>int owner</code>	owner is an int data object.

Initializers

Table 5 (Page 2 of 2). Example Declarators

Example	Description
<code>int *node</code>	node is a pointer to an int data object.
<code>int names[126]</code>	names is an array of 126 int elements.
<code>int *action()</code>	action is a function returning a pointer to an int .
<code>volatile int min</code>	min is an int that has the volatile qualifier.
<code>int * volatile volume</code>	volume is a volatile pointer to an int .
<code>volatile int * next</code>	next is a pointer to a volatile int .
<code>volatile int * sequence[5]</code>	sequence is an array of five pointers to volatile int objects.
<code>extern const volatile int op_system_clock</code>	op_system_clock is a constant and volatile integer with static storage duration and external linkage.

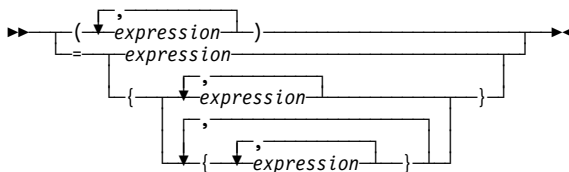
Related Information

- “Enumerations” on page 50
- “Pointers” on page 55
- “Arrays” on page 61
- “Structures” on page 70
- “Unions” on page 78

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object.

An initializer has the form:



C++ Note: The form `(expression)` is allowed in C++ only.

The initializer consists of the `=` symbol followed by an initial *expression* or a braced list of initial expressions separated by commas. The number of initializers must not be more than the number of elements to be initialized. An initializer list with fewer initializers than elements, can end with a comma, indicating that the rest of the uninitialized elements are initialized to zero. The initial expression evaluates to the first value of the data object.

Initializers

To assign a value to a scalar object, use the simple initializer: `= expression`. For example, the following data definition uses the initializer `= 3` to set the initial value of `group` to 3:

```
int group = 3;
```

For unions, structures, and aggregate classes (classes with no constructors, base classes, virtual functions, or private or protected members), the set of initial expressions must be enclosed in `{ }` (braces) unless the initializer is a string literal.

If the initializer of a character string is a string literal, the `{ }` are optional. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas.

In an array, structure, or union initialized using a brace-enclosed initializer list, any members or subscripts that are not initialized are implicitly initialized to zero of the appropriate type.

The initialization properties of each data type are described in the section for that data type.

- An initializer of the form (*expression*) can be used to initialize fundamental types in C++. For example, the following two initializations are identical:

```
int group = 3;
int group(3);
```

- You can also use the (*expression*) form to initialize C++ classes. For more information on initializing classes, see “Initialization by Constructor” on page 306.
- You can initialize variables at file scope with nonconstant expressions. This is not allowed in ANSI/ISO C.
- If your code jumps over declarations that contain initializations, the compiler generates an error. For example, the following code is not valid in C++:

```
goto skiplabel; // error - jumped over declaration
int i = 3;      // and initialization of i
```

```
skiplabel: i = 4;
```

- You can initialize classes in external, static, and automatic definitions. The initializer contains an `=` (equal sign) followed by a brace-enclosed, comma-separated list of values. You do not need to initialize all members of a class.

In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if they were explicitly initialized to zero.

```
static short grid[3][4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

Function Specifiers

The initial values of `grid` are:

Element	Value	Element	Value
<code>grid[0][0]</code>	0	<code>grid[1][2]</code>	1
<code>grid[0][1]</code>	0	<code>grid[1][3]</code>	1
<code>grid[0][2]</code>	0	<code>grid[2][0]</code>	0
<code>grid[0][3]</code>	1	<code>grid[2][1]</code>	0
<code>grid[1][0]</code>	0	<code>grid[2][2]</code>	0
<code>grid[1][1]</code>	0	<code>grid[2][3]</code>	0

Related Information

- “Arrays” on page 61
- “Characters” on page 45
- “Enumerations” on page 50
- “Floating-Point Variables” on page 48
- “Integer Variables” on page 49
- “Pointers” on page 55
- “Structures” on page 70
- “Unions” on page 78

Function Specifiers

The function specifiers **inline** and **virtual** are used only in C++ function declarations, which are described on page 138.

The function specifier **inline** is used to make a suggestion to the compiler to incorporate the code of a function into the code at the point of the call. For more information, see “Inline Functions” on page 163.

The function specifier **virtual** can only be used in nonstatic member function declarations. For more information, see “Virtual Functions” on page 338.

References

A C++ *reference* is an alias or an alternative name for an object. All operations applied to a reference act on the object the reference refers to. The address of a reference is the address of the aliased object.

A reference type is defined by placing the & after the type specifier. You must initialize all references except function parameters when they are defined.

Because arguments of a function are passed by value, a function call does not modify the actual values of the arguments. If a function needs to modify the actual value of an argument, the argument must be passed by *reference* (as opposed to being passed by *value*). Passing arguments by reference can be done using either references or pointers. In C++, this is accomplished transparently. Unlike C, C++ does not force you to use pointers if you want to pass arguments by reference. For example:

```
int f(int&);
void main()
{
    extern int i;
    f(i);
}
```

You cannot tell from the function call `f(i)` that the argument is being passed by reference.

References to NULL are not allowed.

Initializing References

The object that you use to initialize a reference must be of the same type as the reference, or it must be of a type that is convertible to the reference type. If you initialize a reference to a constant using an object that requires conversion, a temporary object is created. In the following example, a temporary object of type **float** is created:

```
int i;
const float& f = i; // reference to a constant float
```

Attempting to initialize a nonconstant reference with an object that requires a conversion is an error.

References

Once a reference has been initialized, it cannot be modified to refer to another object. For example:

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;           // valid
int &RefOne = num2;          // error, two definitions of RefOne
RefOne = num2;               // assign num2 to num1
int &RefTwo;                 // error, uninitialized reference
int &RefTwo = num2;          // valid
```

Note that the initialization of a reference is not the same as an assignment to a reference. Initialization operates on the actual reference by initializing the reference with the object it is an alias for. Assignment operates through the reference on the object referred to.

A reference can be declared without an initializer:

- When it is used in an argument declaration
- In the declaration of a return type for a function call
- In the declaration of class member within its class declaration
- When the **extern** specifier is explicitly used.

You cannot have references to any of:

- Other references
- Bit fields
- Arrays of references
- Pointers to references

Related Information

- “Passing Arguments by Reference” on page 154
- “Pointers” on page 55
- “Declarators” on page 83
- “Initializers” on page 88
- “Temporary Objects” on page 302

Operator Precedence and Associativity

Chapter 4. Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.

This chapter discusses:

Operator Precedence and Associativity	93
Operands	96
lvalues	96
Primary Expressions	97
Unary Expressions	103
Cast Expressions	106
Binary Expressions	114
Conditional Expressions	123
Assignment Expressions	126
Comma Expression ,	129

An expression can result in an lvalue, rvalue, or no value, and can produce side effects in each case.

C++ Note: C++ operators can be defined to behave differently when applied to operands of class type. This is called operator *overloading*. This chapter describes the behavior of operators that are not overloaded.

Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.

For example, in the following statements, the value of 5 is assigned to both a and b because of the right-to-left associativity of the = operator. The value of c is assigned to b first, and then the value of b is assigned to a.

```
b = 9;  
c = 5;  
a = b = c;
```

Because the order of subexpression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```


Operator Precedence and Associativity

the * and / operations are performed before + because of precedence. b is multiplied by c before it is divided by d because of associativity.

The following table lists the C and C++ language operators in order of precedence and shows the direction of associativity for each operator. In C++, the primary scope resolution operator (::) has the highest precedence, followed by the other primary operators. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

Operator Name	Associativity	Operators
Primary scope resolution	left to right	::
Primary	left to right	() [] . ->
Unary	right to left	++ -- + - ! ~ & * (<i>type_name</i>) sizeof new delete
C++ Pointer to Member	left to right	.* ->*
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise Logical AND	left to right	&
Bitwise Exclusive OR	left to right	^ or ~
Bitwise Inclusive OR	left to right	
Logical AND	left to right	&&
Logical OR	left to right	
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^= =
Comma	left to right	,

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing such ambiguous expressions as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, ++y and func1(y) might not be evaluated in the same order by all C language implementations. If y had the value of 1 before the first statement, it is not known whether or not the value of 1 or 2 is passed to func1(). In the second

Operator Precedence and Associativity

statement, if `i` had the value of 1, it is not known whether the first or second array element of `x[]` is passed as the second argument to `func2()`.

The order of grouping operands with operators in an expression containing more than one instance of an operator with both associative and commutative properties is not specified. The operators that have the same associative and commutative properties are: `*`, `+`, `&`, `|`, and `^` (or `~`). The grouping of operands can be forced by grouping the expression in parentheses.

Examples of Expressions and Precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators. If parentheses did not appear in these expressions, the operands and operators are grouped in the same manner as indicated by the parentheses.

```
total = (4 + (5 * 3));  
total = (((8 * 5) / 10) / 3);  
total = (10 + (5/3));
```

Because the order of grouping operands with operators that are both associative and commutative is not specified, the compiler can group the operands and operators in the expression:

```
total = price + prov_tax + city_tax;
```

in the following ways (as indicated by parentheses):

```
total = (price + (prov_tax + city_tax));  
total = ((price + prov_tax) + city_tax);  
total = ((price + city_tax) + prov_tax);
```

If the values in this expression are integers, the grouping of operands and operators does not affect the result. Because intermediate values are rounded, different groupings of floating-point operators may give different results.

In certain expressions, the grouping of operands and operators can affect the result. For example, in the following expression, each function call might be modifying the same global variables.

```
a = b() + c() + d();
```

This expression can give different results depending on the order in which the functions are called.

Ivalue

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable `a`.

```
a = b();
a += c();
a += d();
```

Operands

Most expressions can contain several different, but related, types of operands. The following *type classes* describe related types of operands:

- Integral** Character objects and constants, objects having an enumeration type, and objects having the type **short**, **int**, **long**, **unsigned short**, **unsigned int**, or **unsigned long**.
- Arithmetic** Integral objects and objects having the type **float**, **double**, and **long double**.
- Scalar** Arithmetic objects and pointers to objects of any type. Also C++ references.
- Aggregate** Arrays, structures, and unions. Also C++ classes.

Many operators cause conversions from one data type to another. Conversions are discussed in Chapter 5, "Implicit Type Conversions" on page 131.

Ivalues

An *Ivalue* is an expression that represents an object that can be examined or changed. A *modifiable Ivalue* is an expression representing an object that can be changed. It is typically the left operand in an assignment expression. For example, arrays and **const** objects are not modifiable Ivalues, but **static int** objects are.

All assignment operators evaluate their right operand and assign that value to their left operand. The left operand must evaluate to a reference to an object.

The address operator (&) requires an Ivalue as an operand while the increment (++) and the decrement (--) operators require a modifiable Ivalue as an operand.

Examples of Ivalues

Expression	Lvalue
<code>x = 42;</code>	<code>x</code>
<code>*ptr = newvalue;</code>	<code>*ptr</code>
<code>a++</code>	<code>a</code>

Primary Expressions

Related Information

- “Dot Operator .” on page 102
- “Arrow Operator ->” on page 103
- “Assignment Expressions” on page 126
- “Address &” on page 106

Primary Expressions

A *primary expression* can be:

- An identifier
- A qualified class name
- A string literal
- A parenthesized expression
- A constant expression
- A function call
- An array element specification
- A structure or union member specification.

All primary operators have the same precedence and have left-to-right associativity.

C++ Scope Resolution Operator ::

The :: (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a file scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int i = 10;
int f(int i)
{
    return i ? i : :: i; // return global i if local i is zero
}
```

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator. Whenever a name is followed by a :: operator, the name is interpreted as a class name.

In the following example, the declaration of the variable `X` hides the class type `X`, but you can still use the static class member `count` by qualifying it with the class type `X` and the scope resolution operator.

Primary Expressions

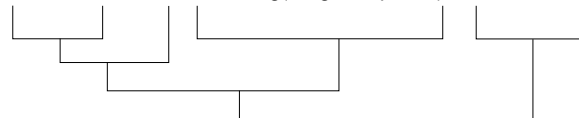
```
#include <iostream.h>
class X
{
public:
    static int count;
};
int X::count = 10;           // define static data member
void main ()
{
    int X = 0;               // hides class type X
    cout << X::count << endl; // use static member of class X
}
```

The scope resolution operator is also discussed in “Class Names” on page 242 and in “Scope of Class Names” on page 246.

Parenthesized Expressions ()

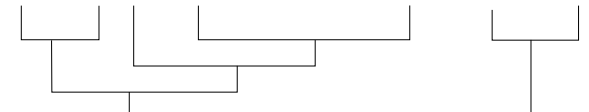
Use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding `weight`, `zipcode` are used to form a function call. Note how the compiler groups the operands and operators in the expression according to the rules for operator precedence and associativity:

```
-discount * item + handling(weight, zipcode) < .10 * item
```



The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:

```
(-discount * (item + handling(weight, zipcode) ) ) < (.10 * item)
```



Primary Expressions

In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:

```
x = f + (g + h);
```

Constant Expressions

A *constant expression* is an expression with a value that may be determined during compilation and cannot be changed at runtime, it can only be evaluated. Constant expressions can be composed of integer constants, character constants, floating-point constants, and enumeration constants, address constants, and other constant expressions. Some constant expressions, such as a string literal or an address constant, are lvalues.

The C++ language requires integral constant expressions in the following places:

- In the subscript declarator, as the description of an array bound
- After the keyword **case** in a **switch** statement
- In an enumerator, as the numeric value of an enum constant
- In a bit-field width specifier
- In the preprocessor **#if** statement (Enumeration constants, address constants, and **sizeof** cannot be specified in the preprocessor **#if** statement.)

In all these contexts except for an initializer of a file scope data definition, the constant expression can contain integer, character, and enumeration constants, casts to integral types, and **sizeof** expressions. Function-scope **static** and **extern** declarations can be initialized with the address of a previously defined **static** or **extern**.

In a file scope data definition, the initializer must evaluate to a constant or to the address of a static storage (**extern** or **static**) object (plus or minus an integer constant) that is defined or declared earlier in the file. The constant expression in the initializer can contain:

- integer, character, enumeration, and float constants
- casts to any type
- **sizeof** expressions
- unary address expressions (static objects only)

Functions, class objects, pointers, and references are not allowed unless they occur in **sizeof** expressions. Comma operators and assignment operators cannot appear in constant expressions.

Examples of Constant Expressions

The following examples show constants used in expressions.

Expression	Constant
<code>x = 42;</code>	42
<code>extern int cost = 1000;</code>	1000
<code>y = 3 * 29;</code>	3 * 29

Primary Expressions

Function Calls ()

A *function call* is a primary expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty.

For example:

```
stub()  
overdue(account, date, amount)  
notify(name, date + 5)  
report(error, time, date, ++num)
```

The arguments are evaluated, and each formal parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment. Assigning a value to a formal parameter within the function body changes the value of the parameter within the function, but has no effect on the argument.

The type of a function call expression is the return type of the function. The return value is determined by the return statement in the function definition. The result of a function call is an lvalue only if the function returns a reference. A function can call itself.

If you want a function to change the value of a variable, pass a pointer to the variable you want changed. When a pointer is passed as a parameter, the pointer is copied; the object pointed to is not copied. (See “Pointers” on page 55.)

Arguments that are arrays and functions are converted to pointers before being passed as function arguments.

Arguments passed to nonprototyped C functions undergo conversions: type **short** or **char** parameters are converted to **int**, and **float** parameters to **double**. Use a cast expression for other conversions. (See “Cast Expressions” on page 106.)

In C only, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is explicitly declared because an implicit declaration of `extern int func();` is assumed. This is *not* true for C++.

The compiler compares the data types provided by the calling function with the data types that the called function expects. The compiler also performs type conversions if the declaration of the function is either:

- In function prototype format and the parameters differ from the prototype
- OR
- Visible at the point where the function is called.

Primary Expressions

For example, the declaration of `func` is a prototype. When function `func` is called, parameter `f` is converted to a **double**, and parameter `c` is converted to an **int**:

```
char * func (double d, int i);
    /* ... */
main
{
    float f;
    char c;
    func(f, c) /* f is a double, c is an int */
}
```

The order in which parameters are evaluated is not specified. Avoid such calls as:
`method(sample1, batch.process--, batch.process);`

In this example, `batch.process--` might be evaluated last, causing the last two arguments to be passed with the same value.

In the following example, `main` passes `func` two values: 5 and 7. The function `func` receives copies of these values and accesses them by the identifiers: `a` and `b`. The function `func` changes the value of `a`. When control passes back to `main`, the actual values of `x` and `y` are not changed. The called function `func` only receives copies of `x` and `y`, not the values themselves.

```
/**
 ** This example illustrates function calls
 **/

#include <stdio.h>

int main(void)
{
    int x = 5, y = 7;

    func(x, y);
    printf("In main, x = %d    y = %d\n", x, y);
}

void func (int a, int b)
{
    a += b;
    printf("In func, a = %d    b = %d\n", a, b);
}
```

This program produces the following output:

```
In func, a = 12    b = 7
In main, x = 5    y = 7
```

See Chapter 6, "Functions" on page 137 for detailed characteristics of functions.

Primary Expressions

Array Subscript [] (Array Element Specification)

A primary expression followed by an expression in [] (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a *subscript*.

The primary expression must have a pointer type, and the subscript must have integral type. The result of an array subscript is an lvalue.

The first element of each array has the subscript 0. The expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the rightmost subscript most frequently.

For example, the following statement gives the value 100 to each element in the array `code[4][3][6]`:

```
for (first = 0; first <= 3; ++first)
  for (second = 0; second <= 2; ++second)
    for (third = 0; third <= 5; ++third)
      code[first][second][third] = 100;
```

By definition, the expression:

`*((exp1) + (exp2))`

is identical to the expression:

`exp1[exp2]`

which is also identical to:

`exp2[exp1]`

“Arrays” on page 61 explains how to define and use an array.

Dot Operator .

The . (dot) operator is used to access structure or C++ class members using a structure object. The member is specified by a postfix expression, followed by a . (dot) operator, followed by a *name*. The postfix expression must be an object of type **class**, **struct** or **union**. The name must be a member of that object.

The value of the expression is the value of the selected member. If the postfix expression and the name are lvalues, the expression value is also an lvalue.

For more information on class members, see Chapter 10, “Class Members and Friends” on page 253. See also “Unions” on page 78 and “Structures” on page 70.

Unary Expressions

Arrow Operator `->`

The `->` (arrow) operator is used to access structure or C++ class members using a pointer. A postfix expression, followed by an `->` (arrow) operator, followed by a *name*, designates a member of the object to which the pointer points. The postfix expression must be a pointer to an object of type **class**, **struct** or **union**. The name must be a member of that object.

The value of the expression is the value of the selected member. If the name is an lvalue, the expression value is also an lvalue.

For more information on class members, see Chapter 10, “Class Members and Friends” on page 253. See also “Unions” on page 78 and “Structures” on page 70.

Unary Expressions

A *unary expression* contains one operand and a unary operator. All unary operators have the same precedence and have right-to-left associativity.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most unary expressions. See “Arithmetic Conversions” on page 135 for more information.

The following table summarizes the operators for unary expressions:

Increment (<code>++</code>)	Logical Negation (<code>!</code>)	Cast (<i>type_name</i>)
Decrement (<code>--</code>)	Bitwise Negation (<code>~</code>)	sizeof
Unary Plus (<code>+</code>)	Address (<code>&</code>)	new
Unary Minus (<code>-</code>)	Indirection (<code>*</code>)	delete

Increment `++`

The `++` (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.

You can put the `++` before or after the operand. If it appears before the operand, the operand is incremented. Then the incremented value is used in the expression. If you put the `++` after the operand, the value of the operand is used in the expression *before* the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is equivalent to the following three expressions:

```
play1 = play1 + 1;  
play = play1 + play2;  
play2 = play2 + 1;
```

Unary Expressions

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions on the operand are performed. See “Arithmetic Conversions” on page 135.

Decrement `--`

The `--` (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.

You can put the `--` before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the `--` appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

```
play = --play1 + play2--;
```

is equivalent to the following three expressions:

```
play1 = play1 - 1;  
play = play1 + play2;  
play2 = play2 - 1;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions are performed on the operand. See “Arithmetic Conversions” on page 135.

Unary Plus `+`

The `+` (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

The result has the same type as the operand after integral promotion.

Note: Any plus sign in front of a constant is not part of the constant.

Unary Expressions

Unary Minus -

The - (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value 100, `-quality` has the value -100.

The result has the same type as the operand after integral promotion.

Note: Any minus sign in front of a constant is not part of the constant.

Logical Negation !

The ! (logical negation) operator determines whether the operand evaluates to 0 (false) or nonzero (true). The expression yields the value 1 (true) if the operand evaluates to 0, and yields the value 0 (false) if the operand evaluates to a nonzero value. The operand must have a scalar data type, but the result of the operation has always type `int` and is not an lvalue.

The following two expressions are equivalent:

```
!right;  
right == 0;
```

Bitwise Negation ~

The ~ (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.

Suppose `x` represents the decimal value 5. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `~x` yields the following result (represented here as a 16-bit binary number):

```
111111111111010
```

Note that the ~ character can be represented by the trigraph `??~`.

The 16-bit binary representation of `~0` is:

```
1111111111111111
```

Cast Expressions

Address &

The & (address) operator yields a pointer to its operand. The operand must be an lvalue, a function designator, or a qualified name. It cannot be a bit field, nor can it have the storage class **register**.

If the operand is an lvalue or function, the resulting type is a pointer to the expression type. For example, if the expression has type **int**, the result is a pointer to an object having type **int**.

If the operand is a qualified name and the member is not static, the result is a pointer to a member of class and has the same type as the member. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

See also “Pointers” on page 55.

C++ Note: You can use the & operator with overloaded functions only in an initialization or assignment where the left side uniquely determines which version of the overloaded function is used. For more information, see “Overloading Functions” on page 277.

Indirection *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. The operation yields an lvalue or a function designator if the operand points to a function. Arrays and functions are converted to pointers.

The type of the operand determines the type of the result. For example, if the operand is a pointer to an **int**, the result has type **int**.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`. The result is not defined.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the expressions:

```
p_to_y = &y;  
*p_to_y = 3;
```

cause the variable `y` to receive the value 3.

See also “Pointers” on page 55.

Cast Expressions

Cast Expressions

The cast operator is used for *explicit type conversions*. It converts the value of the operand to a specified data type and performs the necessary conversions to the operand for the type.

For C, the operand must be scalar and the type must be either scalar or **void**. For C++, the operand can have class type. If the operand has class type, it can be cast to any type for which the class has a user-defined conversion function. User-defined conversion functions are described in “Conversion Functions” on page 305.

The result of a cast is not an lvalue unless the cast is to a reference type. When you cast to a reference type, no user-defined conversions are performed and the result is an lvalue.

There are two types of casts that take one argument:

- *C-style* casts, with the format $(X)a$.
- *function-style* casts with one argument, such as $X(a)$.

Both types of casts convert the argument a to the type X . In C++, they can invoke a constructor, if the target type is a class, or they can invoke a conversion function, if the source type is a class. They can be ambiguous if both conditions hold.

A function-style cast with no arguments, such as $X()$, creates a temporary object of type X . If X is a class with constructors, the default constructor $X::X()$ is called.

A function-style cast with more than one argument, such as $X(a,b)$, creates a temporary object of type X . This object must be a class with a constructor that takes two arguments of types compatible with the types of a and b . The constructor is called with a and b as arguments.

- For more information on implicit conversions using constructors, see “Conversion by Constructor” on page 304.
- Explicit conversions can also be done using conversion functions. For more information, see “Conversion Functions” on page 305.
- Implicit conversions using standard types are described in “Standard Type Conversions” on page 132.

sizeof (Size of an Object)

The **sizeof** operator yields the size in *bytes* of the operand. Types cannot be defined in a **sizeof** expression. The **sizeof** operation cannot be performed on

- A bit field
- A function
- An undefined structure or class
- An incomplete type (such as **void**)

The operand can be the parenthesized name of a type or expression.

The compiler must be able to evaluate the size at compile time. The expression is not evaluated; there are no side effects. For example, the value of b is 5 from initialization to the end of program runtime:

Cast Expressions

```
#include <stdio.h>

int main(void){
    int b = 5;
    sizeof(b++);
}
```

The result is an integer constant.

The size of a **char** object is the size of a byte. For example, if a variable *x* has type **char**, the expression `sizeof(x)` always evaluates to 1.

The result of a `sizeof` operation has type **size_t**, which is an unsigned integral type defined in the `<stddef.h>` header.

The size of an object is determined on the basis of its definition. The **sizeof** operator does not perform any conversions. If the operand contains operators that perform conversions, the compiler does take these conversions into consideration. The following expression causes the usual arithmetic conversions to be performed. The result of the expression `x + 1` has type **int** (if *x* has type **char**, **short**, or **int** or any enumeration type) and is equivalent to `sizeof(int)`:

```
sizeof (x + 1);
```

Except in preprocessor directives, you can use a **sizeof** expression wherever an integral constant is required. One of the most common uses for the **sizeof** operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.

Another use of **sizeof** is in porting code across platforms. You should use the **sizeof** operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

The result of a `sizeof` expression depends on the type it is applied to:

An array The result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression.

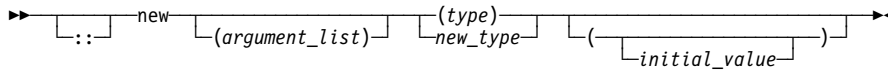
A class The result is always nonzero, and is equal to the number of bytes in an object of that class including any padding required for placing class objects in an array.

A reference The result is the size of the referenced object.

C++ new Operator

The **new** operator provides dynamic storage allocation. The syntax for an allocation expression containing the **new** operator is:

Cast Expressions



If you prefix **new** with the scope resolution operator (`::`), the **global operator new()** is used. If you specify an *argument_list*, the overloaded **new** operator that corresponds to that *argument_list* is used. The *type* is an existing built-in or user-defined type. A *new_type* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the **new** operator is used to find storage in free store for the object being created. The *new expression* returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

You can use the routine **set_new_handler()** to change the default behavior of **new**. See “set_new_handler() — Set Behavior for new Failure” on page 112 for more information.

You cannot use the **new** operator to allocate function types, **void**, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the **new** operator. You cannot create a reference with the **new** operator.

When the object being created is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when an existing *type* is used. You can create an array with zero bounds with the **new** operator. For example:

```
char * c = new char[0];
```

In this case, a pointer to a unique object is returned.

An object created with **operator new()** or **operator new[]()** exists until the **operator delete()** or **operator delete[]()** is called to deallocate the object's memory, or until program ends.

If parentheses are used within a *new_type*, parentheses should also surround the *new_type* to prevent syntax errors. In the following example, storage is allocated for an array of pointers to functions:

Cast Expressions

```
void f();
void g();
void main()
{
    void (**p)(), (**q)();
    // declare p and q as pointers to pointers to void functions
    p = new (void (*[3])());
    // p now points to an array of pointers to functions
    q = new void(*[3])(); // error
    // error - bound as 'q = (new void) (*[3])()';
    p[0] = f; // p[0] to point to function f
    q[2] = g; // q[2] to point to function g
    p[0](); // call f()
    q[2](); // call g()
}
```

However, the second use of **new** causes an erroneous binding of `q = (new void) (*[3])()`.

The type of the object being created cannot contain class declarations, enumeration declarations, or **const** or **volatile** types. It can contain pointers to **const** or **volatile** objects.

For example, `const char*` is allowed, but `char* const` is not.

Additional arguments can be supplied to **new** by using the *argument_list*, also called the *placement syntax*. If placement arguments are used, a declaration of **operator new()** or **operator new[]()** with these arguments must exist. For example:

```
#include <stddef.h>
class X
{
public:
    void* operator new(size_t,int, int){ /* ... */ }
};
// .
// .
// .
void main ()
{
    X* ptr = new(1,2) X;
}
```

For more information on the class member **operator new()** and **operator new[]()** function, see “Overloaded new and delete” on page 291 and “Free Store” on page 299. For more information on constructing and destructing class objects with **new** and **delete**, see “Constructors and Destructors Overview” on page 293.

Cast Expressions

Member Functions and the Global operator new() and operator new[]()

When an object of a class type is created with the **new** operator, the member **operator new()** function (for objects that are not arrays) or the member **operator new[]()** function (for arrays of any number of dimensions) is implicitly called. The first argument is the amount of space requested.

The following rules determine which storage allocation function is used:

1. If your own **operator new[]()** exists, the object is an array, and the **::** (scope resolution) operator is not used, your **operator new[]()** is used.
2. If you have not defined an **operator new[]()** function, the global **::operator new[]()** function defined in `<new.h>` is used. The allocation expression of the form **::operator new[]()** ensures that the global new operator is called, rather than your class member operator.
3. If your own **operator new()** exists, and the object is not an array, and the **::** operator is not used, your **operator new()** is used.
4. If you have not defined an **operator new()** function, the global **::operator new()** function defined in `<new.h>` is used. The allocation expression of the form **::operator new()** ensures that the global new operator is called, rather than your class member operator.

When a nonclass object is created with the new operator, the global **::operator new()** is used.

The order of evaluation of a call to an **operator new()** is undefined in the evaluation of arguments to constructors. If **operator new()** returns 0, the arguments to a constructor may or may not have been evaluated.

Initializing Objects Created with the new Operator

You can initialize objects created with the **new** operator in several ways. For nonclass objects, or for class objects without constructors, a *new initializer* expression can be provided in a new expression by specifying (*expression*) or (). For example:

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

If a class has a constructor, the new initializer must be provided when any object of that class is allocated. The arguments of the new initializer must match the arguments of a class constructor, unless the class has a default constructor.

You cannot specify an initializer for arrays. You can initialize an array of class objects only if the class has a default constructor. The constructor is called to initialize each array element (class object).

Initialization using the new initializer is performed only if **new** successfully allocates storage.

Cast Expressions

For more information on the class member **operator new()** and **operator new[]()** function, see “Overloaded new and delete” on page 291 in Special Overloaded Operators, and “Free Store” on page 299. For more information on constructing and destructing class objects with **new** and **delete**, see “Constructors and Destructors Overview” on page 293.

set_new_handler() — Set Behavior for new Failure

When the **new** operator creates a new object, it calls the **operator new()** or **operator new[]()** function to obtain the needed storage.

When **new** cannot allocate storage to create a new object, it calls a *new handler* function if one has been installed by a call to **set_new_handler()**. The **set_new_handler()** function is defined in `<new.h>`. Use it to call a new handler you have defined or the default new handler.

The **set_new_handler()** function has the prototype:

```
typedef void(*PNH) ();
PNH set_new_handler(PNH);
```

set_new_handler() takes as an argument a pointer to a function (the new handler), which has no arguments and returns void. It returns a pointer to the previous new handler function.

If you do not specify your own **set_new_handler()** function, **new** returns the NULL pointer.

The following program fragment shows how you could use **set_new_handler()** to return a message if the **new** operator cannot allocate storage:

```
#include <iostream.h>
#include <new.h>
void no_storage()
{
    cerr << "Operator new failed: no storage is available.\n";
    exit(1);
}
main()
{
    set_new_handler(&no_storage);
    // Rest of program ...
}
```

If the program fails because **new** cannot allocate storage, the program exits with the message:

```
Operator new failed: no storage is available.
```

Cast Expressions

C++ delete Operator

The **delete** operator destroys the object created with **new** by deallocating the memory associated with the object.

The **delete** operator has a **void** return type. It has the syntax:

```
▶▶ [::] delete object_pointer ▶▶
```

The operand of **delete** must be a pointer returned by **new**, and cannot be a pointer to constant. If an attempt to create an object with **new** fails, the pointer returned by **new** will have a zero value, but it can still be used with **delete**. Deleting a null pointer has no effect.

The **delete[]** operator frees storage allocated for array objects created with **new[]**. The **delete** operator frees storage allocated for individual objects created with **new**.

It has the syntax:

```
▶▶ [::] delete [ ] array ▶▶
```

The result of deleting an array object with **delete** is undefined, as is deleting an individual object with **delete[]**. The array dimensions do not need to be specified with **delete[]**.

The results of attempting to access a deleted object are undefined because the deletion of an object can change its value.

If a destructor has been defined for a class, **delete** invokes that destructor. Whether a destructor exists or not, **delete** frees the storage pointed to by calling the function **operator delete()** of the class if one exists.

The global **::operator delete()** is used if:

- The class has no **operator delete()**.
- The object is of a nonclass type.
- The object is deleted with the **::delete** expression.

Binary Expressions

The global `::operator delete[]()` is used if:

- The class has no `operator delete[]()`
- The object is of a nonclass type
- The object is deleted with the `::delete[]` expression.

The default global `operator delete()` only frees storage allocated by the default global `operator new()`. The default global `operator delete[]()` only frees storage allocated for arrays by the default global `operator new[]()`.

For more information on the class member `operator new()` and `operator new[]()` functions, see “Overloaded new and delete” on page 291 in “Special Overloaded Operators,” and “Free Store” on page 299. For more information on constructing and destructing class objects with `new` and `delete`, see “Constructors and Destructors Overview” on page 293.

C++ throw Expressions

A *throw* expression is used to throw exceptions to C++ exception handlers. It causes control to be passed out of the block enclosing the **throw** statement to the first C++ exception handler whose catch argument matches the throw expression. A throw expression is a unary expression of type **void**.

For more information on the throw expression, see Chapter 15, “Exception Handling.”

Binary Expressions

A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence. The table in the section “Operator Precedence and Associativity” on page 93 shows the order of precedence among operators. All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. To ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most binary expressions. See “Arithmetic Conversions” on page 135 for more information.

The following table summarizes the operators for binary expressions:

Multiplication (*)	Subtraction (-)	Bitwise AND (&)
Division (/)	Bitwise Shifts (<< >>)	Bitwise Exclusive OR (^)
Remainder (%)	Relational (< > <= >=)	Bitwise Inclusive OR ()
Addition (+)	Equality (== !=)	Logical AND (&&)
Logical OR ()	Pointer to Member (. * ->*)	

Binary Expressions

Multiplication *

The * (multiplication) operator yields the product of its operands. The operands must have an arithmetic type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed. See “Arithmetic Conversions” on page 135.

Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost  
sites * (number * cost)  
(cost * sites) * number
```

Division /

The / (division) operator yields the quotient of its operands. The operands must have an arithmetic type. The result is not an lvalue.

If both operands are positive integers and the operation produces a remainder, the remainder is ignored. For example, expression $7 / 4$ yields the value 1 (rather than 1.75 or 2). On all IBM C and C++ compilers, if either operand is negative, the result is rounded towards zero.

The result is undefined if the second operand evaluates to 0.

The usual arithmetic conversions on the operands are performed. See “Arithmetic Conversions” on page 135.

Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression $5 \% 3$ yields 2. The result is not an lvalue.

Both operands must have an integral type. If the right operand evaluates to 0, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of a if b is not 0 and a/b is representable:

$$(a / b) * b + a \% b;$$

The sign of the remainder is the same as the sign of the quotient.

The usual arithmetic conversions on the operands are performed. See “Arithmetic Conversions” on page 135.

Binary Expressions

Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type and the other operand must have an integral type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount specified by the integral value. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. The compiler does not provide boundary checking on the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

See “Pointer Conversions” on page 133 and “Pointer Arithmetic” on page 57 for more information about expressions containing pointers.

Subtraction –

The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral type. You cannot subtract a pointer from an integral value.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to the same type, the compiler converts the result to an integral type that represents the number of objects separating the two addresses. Behavior is undefined if the pointers do not refer to objects in the same array.

See “Pointer Conversions” on page 133 and “Pointer Arithmetic” on page 57 for more information about expressions containing pointers.

Binary Expressions

Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

Table 6. Bitwise Shift Operators

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

Each operand must have an integral type. The compiler performs integral promotions on the operands. Then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

If the right operand has the value 0, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `left_op` has the value 4019, the bit pattern (in 16-bit format) of `left_op` is:

```
0000111110110011
```

The expression `left_op << 3` yields:

```
0111110110011000
```

The following table shows the behavior of the >> operator:

Left Operand Type	Result of >>
unsigned type	The vacated bits are filled with zeros.
Nonnegative unsigned type	The integral part of the quotient of the left operand divided by the quantity 2, raised to the power of the right operand. The vacated bits of a signed value are filled with a copy of the sign bit of the unshifted value.
Negative signed type	The language does not specify how the vacated bits produced by the >> operator are filled.

Binary Expressions

Relational < > <= >=

The relational operators compare two operands and determine the validity of a relationship. If the relationship stated by the operator is true, the value of the result is 1. If false, the value of the result is 0. The result is not an lvalue.

The following table describes the four relational operators:

Operator	Usage
<	Indicates whether the value of the left operand is less than the value of the right operand.
>	Indicates whether the value of the left operand is greater than the value of the right operand.
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type. The result has type **int**.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

A pointer can be compared to a constant expression that evaluates to 0. You can also compare a pointer to a pointer of type **void***. The pointer is converted to a pointer of type **void***.

If two pointers refer to the same object, they are considered equal. If two pointers refer to nonstatic members of the same object, the pointer to the object declared later has the higher address value. If two pointers refer to data members of the same union, they have the same address value.

If two pointers refer to elements of the same array, or to the first element beyond the last element of an array, the pointer to the element with the higher subscript value has the higher address value.

You can only compare members of the same object with relational operators.

Relational operators have left-to-right associativity. For example, the expression:

```
a < b <= c
```

is interpreted as:

Binary Expressions

`(a < b) <= c`

If the value of `a` is less than the value of `b`, the first relationship is true and yields the value 1. The compiler then compares the value 1 with the value of `c`.

Equality `==` `!=`

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship stated by an equality operator is true, the value of the result is 1. Otherwise, the value of the result is 0.

The following table describes the two equality operators:

Table 8. Equality Operators

Operator	Usage
<code>==</code>	Indicates whether the value of the left operand is equal to the value of the right operand.
<code>!=</code>	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type, or one operand must have a pointer type and the other operand must be a pointer to void or NULL. The result has type `int`.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value 0, the `==` expression is true only if the pointer operand evaluates to NULL. The `!=` operator evaluates to true if the pointer operand does *not* evaluate to NULL.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Binary Expressions

Note: The equality operator (==) should not be confused with the assignment (=) operator.

For example,

```
if(x == 3)    evaluates to 1 if x is equal to three. Equality tests like this should be
              coded with spaces between the operator and the operands to prevent
              unintentional assignments.

              while
if(x = 3)     is taken to be true because (x = 3) evaluates to a non-zero value (3).
              The expression also assigns the value 3 to x.
```

Bitwise AND &

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.

Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator.

The following example shows the values of a, b, and the result of a & b represented as 16-bit binary numbers:

```
bit pattern of a          0000000001011100
bit pattern of b          0000000000101110
bit pattern of a & b     0000000000001100
```

Note: The bitwise AND (&) should not be confused with the logical AND. (&&) operator. For example,

```
1 & 4 evaluates to 0
while
1 && 4 evaluates to 1
```

Bitwise Exclusive OR ^

The bitwise exclusive OR operator (in EBCDIC, the ^ symbol is represented by the - symbol) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Binary Expressions

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive OR operator. Note that the ^ character can be represented by the trigraph `??^`.

The following example shows the values of a, b, and the result of `a ^ b` represented as 16-bit binary numbers:

```
bit pattern of a      0000000001011100
bit pattern of b      0000000000101110
bit pattern of a ^ b  0000000001110010
```

Bitwise Inclusive OR |

The `|` (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. Note that the `|` character can be represented by the trigraph `??|`.

The following example shows the values of a, b, and the result of `a | b` represented as 16-bit binary numbers:

```
bit pattern of a      0000000001011100
bit pattern of b      0000000000101110
bit pattern of a | b  0000000001111110
```

Note: The bitwise OR (`|`) should not be confused with the logical OR (`||`) operator. For example,

```
1 | 4 evaluates to 5
while
1 || 4 evaluates to 1
```

Logical AND &&

The `&&` (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

Binary Expressions

Unlike the `&` (bitwise AND) operator, the `&&` operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to `0`, the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
<code>1 && 0</code>	<code>0</code>
<code>1 && 4</code>	<code>1</code>
<code>0 && 0</code>	<code>0</code>

The following example uses the logical AND operator to avoid division by zero:

```
(y != 0) && (x / y)
```

The expression `x / y` is not evaluated when `y != 0` evaluates to `0`.

Note: The logical AND (`&&`) should not be confused with the bitwise AND (`&`) operator. For example:

```
1 && 4 evaluates to 1
while
1 & 4 evaluates to 0
```

Logical OR `||`

The `||` (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value `1`. Otherwise, the result has the value `0`.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

Unlike the `|` (bitwise inclusive OR) operator, the `||` operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
<code>1 0</code>	<code>1</code>
<code>1 4</code>	<code>1</code>
<code>0 0</code>	<code>0</code>

The following example uses the logical OR operator to conditionally increment `y`:

```
++x || ++y;
```

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero quantity.

Conditional Expressions

Note: The logical OR (`||`) should not be confused with the bitwise OR (`|`) operator. For example:

```
1 || 4 evaluates to 1
while
1 | 4 evaluates to 5
```

C++ Pointer to Member Operators `.*` `->*`

There are two pointer to member operators: `.*` and `->*`.

The `.*` operator is used to dereference pointers to class members. The first operand must be a class type. If the type of the first operand is class type `T`, or is a class that has been derived from class type `T`, the second operand must be a pointer to a member of a class type `T`.

The `->*` operator is also used to dereference pointers to class members. The first operand must be a pointer to a class type. If the type of the first operand is a pointer to class type `T`, or is a pointer to a class derived from class type `T`, the second operand must be a pointer to a member of class type `T`.

The `.*` and `->*` operators bind the second operand to the first, resulting in an object or function of the type specified by the second operand.

If the result of `.*` or `->*` is a function, you can only use the result as the operand for the `()` (function call) operator. If the second operand is an lvalue, the result of `.*` or `->*` is an lvalue.

For more information on pointer to member operators, see “Pointers to Members” on page 260.

Conditional Expressions

A *conditional expression* is a compound expression that contains a condition (*operand*₁), an expression to be evaluated if the condition has a nonzero value (*operand*₂), and an expression to be evaluated if the condition has the value 0 (*operand*₃).

Conditional expressions have right-to-left associativity. The left operand is evaluated first, and then only one of the remaining two operands is evaluated.

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. All expressions that occur between the `?` and `:` are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:

- An arithmetic type
- A compatible pointer, structure, or union type

Conditional Expressions

- void.

The second and third operands can also be a pointer or a null pointer constant.

Two objects are compatible when they have the same type but not necessarily the same type qualifiers (**volatile**, **const**, or **_Packed**). Pointer objects are compatible if they have the same type or are pointers to void.

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is not equal to 0, the second operand is evaluated.
- If the value is equal to 0, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, the usual arithmetic conversions are performed on the values. The types of the second and third operands determine the type of the result as shown in the following tables.

Type of Conditional C Expressions

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic type after usual arithmetic conversions
Structure or union type	Compatible structure or union type	Structure or union type with all the qualifiers on both operands
void	void	void
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	NULL pointer (the constant 0)	Pointer to type
Pointer to object or incomplete type	Pointer to void	Pointer to void with all the qualifiers specified for the type

Type of Conditional C++ Expressions

Conditional Expressions

Type of One Operand	Type of Other Operand	Type of Result
Reference to type	Reference to type	Reference after usual reference conversions
Class T	Class T	Class T
Class T	Class X	Class type for which a conversion exists. If more than one possible conversion exists, the result is ambiguous.
throw expression	Other (type, pointer, reference)	Type of the expression that is not a throw expression

Examples of Conditional Expressions

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

```
x = (y > z) ? y : z;
```

The following is an equivalent statement:

```
if (y > z)
    x = y;
else
    x = z;
```

The following expression calls the function `printf`, which receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, `printf` receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the `=` operator has higher precedence than the `?:` operator in the following expression:

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

This expression generates an error because it is interpreted as if it were parenthesized this way:

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

That is, `k` is treated as the third operand, not the entire assignment expression `k = j`. The error arises because a conditional expression is not an lvalue, and the assignment is not valid.

To make the expression evaluate correctly, enclose the last operand in parentheses:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```


Assignment Expressions

Assignment Expressions

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators: simple assignment and compound assignment.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

The result of an assignment expression is not an lvalue.

All assignment operators have the same precedence and have right-to-left associativity.

Simple Assignment =

The simple assignment operator stores the value of the right operand in the object designated by the left operand.

Both operands must have arithmetic types, the same structure type, or the same union type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant 0 or NULL.

If both operands have arithmetic types, the system converts the type of the right operand to the type of the left operand before the assignment.

If the right operand is a pointer to a type, the left operand can be a pointer to a **const** of the same type. If the right operand is a pointer to a **const** type, the left operand must also be a pointer to a **const** type.

If the right operand is a pointer to a type, the left operand can be a pointer to a **volatile** of the same type. If the right operand is a pointer to a **volatile** type, the left operand must also be a pointer to a **volatile** type.

If the left operand is a pointer to a member, the right operand must be a pointer to a member or a constant expression that evaluates to zero. The right operand is converted to the type of the left operand before assignment.

If the left operand is an object of reference type, the assignment is to the object denoted by the reference.

If the left operand is a pointer and the right operand is the constant 0, the result is NULL.

Pointers to void can appear on either side of the simple assignment operator.

A packed structure or union can be assigned to a nonpacked structure or union of the same type, and a nonpacked structure or union can be assigned to a packed structure or union of the same type.

Assignment Expressions

If one operand is packed and the other is not, the layout of the right operand is remapped to match the layout of the left. This remapping of structures might degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

Note: If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked. See “Initializing Pointers” on page 56 for more information on assignments with pointers.

You can assign values to operands with the type qualifier **volatile**. You cannot assign a pointer of an object with the type qualifier **const** to a pointer of an object without the **const** type qualifier. For example:

```
const int *p1;
int *p2;
p2 = p1; /* this is NOT allowed */

p1 = p2; /* this IS allowed */
```

The following example assigns the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

The following example assigns in order the value 0 (zero) to `strangeness`, the value of `strangeness` to `charm`, the value of `charm` to `beauty`, and the value of `beauty` to `truth`:

```
truth = beauty = charm = strangeness = 0;
```

Note: The assignment (`=`) operator should not be confused with the equality comparison (`==`) operator. For example:

```
if(x == 3)    evaluates to 1 if x is equal to three

while
if(x = 3)    is taken to be true because (x = 3) evaluates to a non-zero value (3).
             The expression also assigns the value 3 to x.
```

Assignment Expressions

Compound Assignment

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and give the result of that operation to the left operand.

The following table shows the operand types of compound assignment expressions:

Operator	Left Operand	Right Operand
<code>+=</code> or <code>-=</code>	Arithmetic	Arithmetic
<code>+=</code> or <code>-=</code>	Pointer	Integral type
<code>*=</code> , <code>/=</code> , and <code>%=</code>	Arithmetic	Arithmetic
<code><<=</code> , <code>>>=</code> , <code>&=</code> , <code>^=</code> , and <code> =</code>	Integral type	Integral type

Note that the expression

```
a *= b + c
```

is equivalent to

```
a = a * (b + c)
```

and *not*

```
a = a * b + c
```

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
<code>+=</code>	<code>index += 2</code>	<code>index = index + 2</code>
<code>-=</code>	<code>*(pointer++) -= 1</code>	<code>*pointer = *(pointer++) - 1</code>
<code>*=</code>	<code>bonus *= increase</code>	<code>bonus = bonus * increase</code>
<code>/=</code>	<code>time /= hours</code>	<code>time = time / hours</code>
<code>%=</code>	<code>allowance %= 1000</code>	<code>allowance = allowance % 1000</code>
<code><<=</code>	<code>result <<= num</code>	<code>result = result << num</code>
<code>>>=</code>	<code>form >>= 1</code>	<code>form = form >> 1</code>
<code>&=</code>	<code>mask &= 2</code>	<code>mask = mask & 2</code>
<code>^=</code>	<code>test ^= pre_test</code>	<code>test = test ^ pre_test</code>
<code> =</code>	<code>flag = 0N</code>	<code>flag = flag 0N</code>

Although the equivalent expression column shows the left operands (from the example column) evaluated twice, the left operand is evaluated only once.

Comma Expression ,

A *comma expression* contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

In the following example, if `omega` has the value 11, the expression increments `delta` and assigns the value 3 to `alpha`:

```
alpha = (delta++, omega % 4);
```

Any number of expressions separated by commas can form a single expression. The compiler evaluates the leftmost expression first. The value of the rightmost expression becomes the value of the entire expression.

For example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

The primary use of the comma operator is to produce side effects in the following situations:

- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

To use the comma operator in a context where the comma has other meanings, such as in a list of function arguments or a list of initializers, you must enclose the comma operator in parentheses. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of `a`, the value 5, and the value of `c`. The value of the second argument is the result of the comma expression in parentheses:

```
t = 3, t + 2
```

which has the value 5.

Comma Expression

The following table gives some examples of the uses of the comma operator:

Statement	Effects
<pre>for (i=0; i<2; ++i, f());</pre>	A for statement in which <i>i</i> is incremented and <i>f()</i> is called at each iteration.
<pre>if (f(), ++i, i>1) { /* ... */ }</pre>	An if statement in which function <i>f()</i> is called, variable <i>i</i> is incremented, and variable <i>i</i> is tested against a value. The first two expressions within this comma expression are evaluated before the expression <i>i>1</i> . Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the if statement is processed.
<pre>func((++a, f(a)));</pre>	A function call to <i>func()</i> in which <i>a</i> is incremented, the resulting value is passed to a function <i>f()</i> , and the return value of <i>f()</i> is passed to <i>func()</i> . The function <i>func()</i> is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

Chapter 5. Implicit Type Conversions

There are two kinds of implicit type conversions: standard conversions and user-defined conversions. This chapter describes the following standard type conversions:

Integral Promotions	131
Standard Type Conversions	132
Arithmetic Conversions	135

Related Information

- Chapter 4, “Expressions and Operators” on page 93
- Chapter 6, “Functions” on page 137
- “Cast Expressions” on page 106.
- “User-Defined Conversions” on page 304.

Integral Promotions

Certain fundamental types can be used wherever an integer can be used. The fundamental types that can be converted through integral promotion are:

- **char**
- **wchar_t**
- **short int**
- enumerators
- objects of enumeration type
- integer bit fields (both signed and unsigned)

Except for **wchar_t**, if the value cannot be represented by an **int**, the value is converted to an **unsigned int**. For **wchar_t**, if an **int** can represent all the values of the original type, the value is converted to the type that can best represent all the values of the original type. For example, if a **long** can represent all the values, the value is converted to a **long**.

Standard Type Conversions

Standard Type Conversions

Many C and C++ operators cause *implicit type conversions*, which change the type of a value. When you add values having different data types, both values are first converted to the same type. For example, when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type.

Implicit type conversions can occur when:

- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A prototyped function is provided a value that has a different type than the parameter.
- The value specified in the **return** statement of a function has a different type from the defined return type for the function.

You can perform explicit type conversions using the cast operator or the function style cast. For more information on explicit type conversions, see “Cast Expressions” on page 106.

Signed-Integer Conversions

The compiler converts a signed integer to a shorter integer by truncating the high-order bits and converting the variable to a longer signed integer by sign-extension.

Conversion of signed integers to floating-point values takes place without loss of information, except when an **int** or **long int** value is converted to a **float**, in which case some precision may be lost. When a signed integer is converted to an unsigned integer, the signed integer is converted to the size of the unsigned integer, and the result is interpreted as an unsigned value.

Unsigned-Integer Conversions

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Zero-extending pads the leftmost bits of the longer integer with binary zeros.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value changes if the sign bit is set.

Standard Type Conversions

Floating-Point Conversions

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the compiler cannot exactly represent the **double** value as a **float**, the value loses precision. If the value is too large to fit into a **float**, the result is undefined.

When a floating-point value is converted to an integer value, the decimal fraction portion of the floating-point value is discarded in the conversion. If the result is too large for the given integer type, the result of the conversion is undefined.

Pointer Conversions

Pointer conversions are performed when pointers are used, including pointer assignment, initialization, and comparison.

A constant expression that evaluates to zero can be converted to a pointer. This pointer will be a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

Any pointer to an object that is not a **const** or **volatile** object can be converted to a **void***. You can also convert any pointer to a function to a **void***, provided that a **void*** has sufficient bits to hold it.

You can convert an expression with type array of some type to a pointer to the initial element of the array, except when the expression is used as the operand of the **&** (address) operator or the **sizeof** operator.

You can convert an expression with a type of function returning T to a pointer to a function returning T, except when the expression is used as the operand of the **&** (address) operator, the **()** (function call) operator, or the **sizeof** operator.

You can convert an integer value to an address offset.

You can convert a pointer to a class A to a pointer to an accessible base class B of that class, as long as the conversion is not ambiguous. The conversion is ambiguous if the expression for the accessible base class can refer to more than one distinct class. The resulting value points to the base class subobject of the derived class object. A null pointer (pointer with a zero value) is converted into itself.

Note: You cannot convert a pointer to a class into a pointer to its base class if the base class is a virtual base class of the derived class.

For more information on pointer conversions, see “Pointer Arithmetic” on page 57.

Conversion of AS/400 pointers is subject to certain restrictions. See *VisualAge for C++ for AS/400 C++ Programming Guide*, for more information on AS/400 pointer conversion.

Standard Type Conversions

Reference Conversions

A reference conversion can be performed wherever a reference initialization occurs, including reference initialization done in argument passing and function return values. A reference to a class can be converted to a reference to an accessible base class of that class as long as the conversion is not ambiguous. The result of the conversion is a reference to the base class subobject of the derived class object.

Reference conversion is allowed if the corresponding pointer conversion is allowed.

Pointer-to-Member Conversions

Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared.

A constant expression that evaluates to zero is converted to a distinct pointer to a member.

Note: A pointer to a member is not the same as a pointer to an object or a pointer to a function.

A pointer to a member of a base class can be converted to a pointer to a member of a derived class if the following conditions are true:

- The conversion is not ambiguous. The conversion is ambiguous if multiple instances of the base class are in the derived class.
- A pointer to the derived class can be converted to a pointer to the base class. If this is the case, the base class is said to be *accessible*. See “Derivation Access of Base Classes” on page 325 for more information.

For more information, see “Pointers to Members” on page 260 and “C++ Pointer to Member Operators `.*` `->*`” on page 123.

Function Argument Conversions

If no function prototype declaration is visible when a function is called, the compiler can perform default argument promotions, which consist of the following:

- Integral promotions
- Arguments with type **float** are converted to type **double**.

Arithmetic Conversions

Other Conversions

By definition, the **void** type has no value. Therefore, it cannot be converted to any other type, and no other value can be converted to **void** by assignment. However, a value can be explicitly cast to **void**.

No conversions between structure or union types are allowed.

There are no standard conversions between class types.

In C, when you define a value using the **enum** type specifier, the value is treated as an **int**. Conversions to and from an **enum** value proceed as for the **int** type.

You can convert from an **enum** to any integral type but not from an integral type to an **enum**.

Arithmetic Conversions

The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These standard conversions are known as the *arithmetic conversions* because they apply to the types of values ordinarily used in arithmetic.

Arithmetic conversions are used for matching operands of arithmetic operators.

Arithmetic Conversions

Arithmetic conversion proceeds in the following order:

Operand Type	Conversion
One operand has long double type	The other operand is converted to long double type.
One operand has double type	The other operand is converted to double .
One operand has float type	The other operand is converted to float .
One operand has unsigned long int type	The other operand is converted to unsigned long int .
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int can be represented in a long int	The operand with unsigned int type is converted to long int .
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int cannot be represented in a long int	Both operands are converted to unsigned long int
One operand has long int type	The other operand is converted to long int .
One operand has unsigned int type	The other operand is converted to unsigned int .
Both operands have int type	The result is type int .

Chapter 6. Functions

This chapter describes the structure and use of C++ functions. It discusses the following topics:

Functions Overview	137
C++ Enhancements to C Functions	138
Function Declarations	138
Function Definitions	143
The main() Function	148
Calling Functions and Passing Arguments	150
Default Arguments in C++ Functions	156
Function Return Values	159
Pointers to Functions	161
Inline Functions	163

Related Information

- "Member Functions" on page 256
- "Inline Member Functions" on page 257
- Chapter 11, "Overloading" on page 277
- Chapter 12, "Special Member Functions" on page 293
- "Virtual Functions" on page 338

Functions Overview

Functions specify the logical structure of a program and define how particular operations are to be implemented.

A function *declaration* consists of a return type, a name, and an argument list. It is used to declare the format and existence of a function prior to its use.

A function *definition* contains a function declaration and the body of the function. A function can only have one definition.

Both C++ and ANSI/ISO C use the style of declaration called *prototyping*. A prototype refers to the return type, name, and argument list components of a function. It is used by the compiler for argument type checking and argument conversions. Prototypes can appear several times in a program, provided the declarations are compatible. They allow the compiler to check for mismatches between the parameters of a function call and those in the function declaration. C++ functions *must* use prototypes. They are usually placed in header files, while function definitions appear in source files. Nonprototype functions are allowed in C only.

Function Declarations

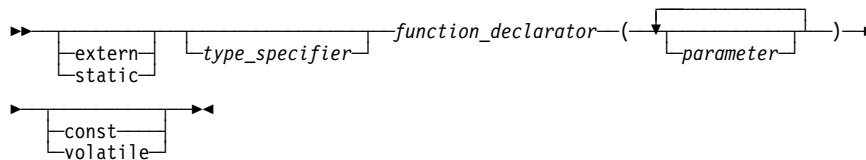
C++ Enhancements to C Functions

The C++ language provides many enhancements to C functions. These are:

- Reference arguments, described on page 154
- Default arguments, described on page 156
- Reference return types, described on page 160
- Inline functions, described on page 163
- Member functions, introduced on page 256
- Overloaded functions, introduced on page 277
- Operator functions, introduced on page 281
- Constructor and destructor functions, introduced on page 293
- Conversion functions, introduced on page 305
- Virtual functions, introduced on page 338
- Function templates, introduced on page 356

Function Declarations

A function declaration establishes the name and the parameters of the function.



A function is declared implicitly by its appearance in an expression if it has not been defined or declared previously; the implicit declaration is equivalent to a declaration of `extern int func_name()`. The default return type of a function is **int**.

To indicate that the function does not return a value, declare it with a return type of **void**.

C++ Note: The use of the **const** and **volatile** specifiers is only supported by C++.

A function cannot be declared as returning a data object having a **volatile** or **const** type but it can return a pointer to a **volatile** or **const** object. Also, a function cannot return a value that has a type of array or function.

If the called function returns a value that has a type other than **int**, you must declare the function before the function call. Even if a called function returns a type **int**, explicitly declaring the function prior to its call is good programming practice.

Some declarations do not have parameter lists; the declarations simply specify the types of parameters and the return values, such as in the following example:

```
int func(int,long);
```

Function Declarations

C++ Function Declarations

In C++, you can specify the qualifiers **volatile** and **const** in member function declarations. You can also specify exception specifications in function declarations. All C++ functions must be declared before they can be called.

Types cannot be defined in return or argument types. For example, the following declarations are not valid in C++:

```
void print(struct X { int i; } x);           //error
enum count{one, two, three} counter();     //error
```

This example attempts to declare a function `print()` that takes an object `x` of class `X` as its argument. However, the class definition is not allowed within the argument list. In the attempt to declare `counter()`, the enumeration type definition cannot appear in the return type of the function declaration. The two function declarations and their corresponding type definitions can be rewritten as follows:

```
struct X { int i; };
void print(X x);
enum count {one, two, three};
count counter();
```

Multiple Function Declarations

All function declarations for a particular function must have the same number and type of arguments, and must have the same return type and the same linkage keywords. These return and argument types are part of the function type, although the default arguments are not.

For the purposes of argument matching, ellipsis and linkage keywords are considered a part of the function type. They must be used consistently in all declarations of a function. If the only difference between the argument types in two declarations is in the use of **typedef** names or unspecified argument array bounds, the declarations are the same. A **const** or **volatile** specifier is also part of the function type, but can only be part of a declaration or definition of a nonstatic member function.

Declaring two functions differing only in return type is not valid function overloading, and is flagged as an error. For example:

```
void f();
int f();           // error, two definitions differ only in
                  // return type

int g()
{
    return f();
}
```

Checking Function Calls

The compiler checks C++ function calls by comparing the number and type of the actual arguments used in the function call with the number and type of the formal arguments in the function declaration. Implicit type conversion is performed when necessary.

Function Declarations

Argument Names in Function Declarations

You can supply argument names in a function declaration, but the compiler ignores them except in the following two situations:

1. If two argument names have the same name within a single declaration. This is an error.
2. If an argument name is the same as a name outside the function. In this case the name outside the function is hidden and cannot be used in the argument declaration.

In the following example, the third argument `intersects` is meant to have enumeration type `subway_line`, but this name is hidden by the name of the first argument. The declaration of the function `subway()` causes a compile-time error because `subway_line` is not a valid type name in the context of the argument declarations.

```
enum subway_line {yonge, university, spadina, bloor};
int subway(char * subway_line, int stations,
            subway_line intersects);
```

Examples of Function Declarations

The following example defines the function `absolute` with the return type `double`. Because this is a noninteger return type, `absolute` is declared prior to the function call.

```
/**
 ** This example shows how a function is declared and defined
 **/

#include <stdio.h>
double absolute(double);

int main(void)
{
    double f = -3.0;

    printf("absolute number = %lf\n", absolute(f));

    return (0);
}

double absolute(double number)
{
    if (number < 0.0)
        number = -number;

    return (number);
}
```

Function Declarations

Specifying a return type of `void` on a function declaration indicates that the function does not return a value. The following example defines the function `absolute` with the return type `void`. Within the function `main`, `absolute` is declared with the return type `void`.

```
/**
 ** This example uses a function with a void return type
 **/

#include <stdio.h>

int main(void)
{
    void absolute(float);
    float f = -8.7;

    absolute(f);

    return(0);
}

void absolute(float number)
{
    if (number < 0.0)
        number = -number;

    printf("absolute number = %f\n", number);
}
```

The following code fragments show several function declarations. The first declares a function `f` that takes two integer arguments and has a return type of **void**:

```
void f(int, int);
```

The following code fragment declares a function `f1` that takes an integer argument, and returns a pointer to a function that takes an integer argument and returns an integer:

```
int (*f1(int))(int);
```

Alternatively, a **typedef** can be used for the complicated return type of function `f1`:

```
typedef int pf1(int);
pf1* f1(int);
```

The following code fragment declares a pointer `p1` to a function that takes a pointer to a constant character and returns an integer:

```
int (*p1) (const char*);
```

The following declaration is of an external function `f2` that takes a constant integer as its first argument, can have a variable number and variable types of other arguments, and returns type **int**.

```
int extern f2(const int ...);
```


Function Declarations

Function `f3` takes an `int` argument with a default value that is the value returned from function `f2`, and that has a return type of `int`:

```
const int j = 5;
int f3( int x = f2(j) );
```

See “Default Arguments in C++ Functions” on page 156 for more information about default function arguments.

Function `f6` is a constant class member function of class `X` with no arguments, and with an `int` return type:

```
class X
{
public:
    int f6() const;
};
```

See “const and volatile Member Functions” on page 256 for more information about constant member functions.

Function `f4` takes no arguments, has return type `void`, and can throw class objects of types `X` and `Y`.

```
class X;
class Y;
//      .
//      .
//      .
void f4() throw(X,Y);
```

Function `f5` takes no arguments, has return type `void`, and cannot throw an exception.

```
void f5() throw();
```

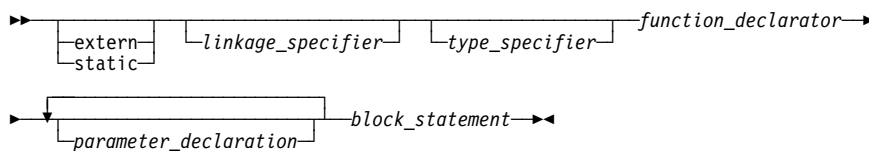
Related Information

- “Declarators” on page 83
- “extern Storage Class Specifier” on page 35

Function Definitions

Function Definitions

A *function definition* contains a function declaration and the body of a function. It specifies the function name, formal parameters, the return type, and storage class of the function.



A function definition (either prototype or nonprototype) contains the following:

- An optional *storage class specifier* **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.
- An optional *type specifier*, which determines the type of value that the function returns. If a type specifier is not given, the function has type **int**.
- A *function declarator*, which provides the function with a name, can further describe the type of the value that the function returns, and can list any parameters that the function expects and their types. The parameters that the function is expecting are enclosed in parentheses.
- A *block statement*, which contains data definitions and code.

A nonprototype function definition can also have a list of *parameter declarations*, which describe the types of parameters that the function receives. In nonprototype functions, parameters that are not declared have type **int**.

A function can be called by itself or by other functions. Unless a function definition has the storage class specifier **static**, the function also can be called by functions that appear in other files. Functions with a storage class specifier of **static** can only be directly invoked from within the same source file.

If a function has the storage class specifier **static** or a return type other than **int**, the function definition or a declaration for the function must appear before, and in the same file as, a call to the function.

In C only, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed. This is *not* true for C++.

All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

The default type for the return value and parameters of a function is **int**, and the default storage class specifier is **extern**. If the function does not return a value or it is not passed any parameters, use the keyword **void** as the type specifier.

Function Definitions

A function can return a pointer or reference to a function, array, or to an object with a **volatile** or **const** type. In C, you cannot declare a function as a struct or union member. (*This restriction does not apply to C++.*)

A function cannot have a return type of function or array. In C, a function cannot return any type having the **volatile** or **const** qualifier. (*This restriction does not apply to C++.*)

You cannot define an array of functions. You can, however, define an array of pointers to functions.

In the following example, ary is an array of two function pointers. Type casting is performed to the values assigned to ary for compatibility:

```
/**
 ** This example uses an array of pointers to functions
 **/

#include <stdio.h>

int func1(void);
void func2(double a);

int main(void)
{
    double num;
    int retnum;
    void (*ary[2]) ();
    ary[0] = ((void(*)())func1);
    ary[1] = ((void(*)())func2);

    retnum=((int (*)())ary[0])();    /* calls func1 */
    printf("number returned = %i\n", retnum);
    ((void (*)(double))ary[1])(num); /* calls func2 */

    return(0);
}

int func1(void)
{
    int number=3;
    return number;
}

void func2(double a)
{
    a=333.3333;
    printf("result of func2 = %f\n", a);
}
```

Function Definitions

The following example is a complete definition of the function `sum`:

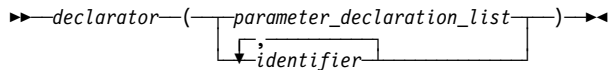
```
int sum(int x,int y)
{
    return(x + y);
}
```

The function `sum` has external linkage, returns an object that has type `int`, and has two parameters of type `int` declared as `x` and `y`. The function body contains a single statement that returns the sum of `x` and `y`.

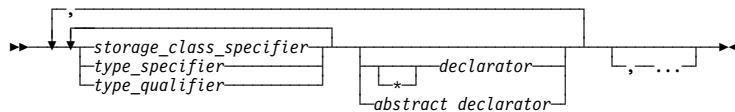
Function Declarator

The *function declarator* names the function and lists the function parameters. It contains an identifier that names the function and a list of the function parameters. You should always use prototype function declarators because of the parameter checking that can be performed. C++ functions *must* have prototype function declarators.

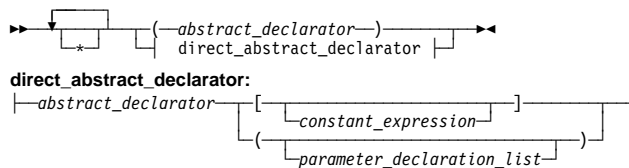
Function Declarator Syntax



Parameter Declaration List Syntax



Abstract Declarator Syntax



Function Definitions

Ellipsis and void

An ellipsis at the end of a parameter declaration indicates that the number of arguments is equal to, or greater than, the number of specified argument types. At least one parameter declaration must come before the ellipsis. Where it is permitted, an ellipsis preceded by a comma is equivalent to a simple ellipsis.

```
int f(int,...);
```

The comma before the ellipsis is optional in C++ only

Parameter promotions are performed as needed, but no type checking is done on the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);    // ANSI/ISO C Standard

int f();       // C++ enhancement
               // Note: In ANSI/ISO C, this declaration means that
               // f may take any number or type or parameters
```

An empty argument declaration list or the argument declaration list of (void) indicates a function that takes no arguments. **void** cannot be used as an argument type, although types derived from **void** (such as pointers to **void**) can be used.

In the following example, the function `f()` takes one integer parameter and returns no value, while `g()` expects no parameters and returns an integer.

```
void f(int);
int g(void);
```

Function Body

The body of a function is a block statement.

The following function body contains a definition for the integer variable `big_num`, an if-else control statement, and a call to the function **printf**:

```
void largest(int num1, int num2)
{
    int big_num;

    if (num1 >= num2)
        big_num = num1;
    else
        big_num = num2;

    printf("big_num = %d\n", big_num);
}
```

Function Definitions

Examples of Function Declarators

The following example contains a function declarator `sort` with `table` declared as a pointer to `int` and `length` declared as type `int`. Note that arrays as parameters are implicitly converted to a pointer to the type.

```
/**
 ** This example illustrates function declarators.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void sort(int table[ ], int length);

int main(void)
{
    int table[ ]={1,5,8,4};
    int length=4;
    printf("length is %d\n",length);
    sort(table,length);
}

void sort(int table[ ], int length)
{
    int i, j, temp;

    for (i = 0; i < length -1; i++)
        for (j = i + 1; j < length; j++)
            if (table[i] > table[j])
                {
                    temp = table[i];
                    table[i] = table[j];
                    table[j] = temp;
                }
}
```

The following examples contain prototype function declarators:

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

main

The following example illustrates how a **typedef** identifier can be used in a function declarator:

```
typedef struct tm_fmt { int minutes;
                       int hours;
                       char am_pm;
                       } struct_t;
long time_seconds(struct_t arrival)
```

The following function `set_date` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class specifier **register**.

```
set_date(register struct date *date_ptr)
{
    date_ptr->mon = 12;
    date_ptr->day = 25;
    date_ptr->year = 87;
}
```

Related Information

- “Block” on page 166
- “Function Definitions” on page 143
- “Function Declarations” on page 138

The main() Function

When a program begins running, the system automatically calls the function **main**, which marks the entry point of the program. Every program must have one function named **main**. No other function in the program can be called **main**. A **main** function has the form:

The diagram shows the syntax of the `main` function. It starts with a right-pointing arrow, followed by a box containing `void` and `int` stacked vertically. This is followed by the text `main`, an opening parenthesis `(`, another box containing `void` and `parameters` stacked vertically, a closing parenthesis `)`, the text `block_statement`, and finally a double-headed arrow.

By default, **main** has the storage class **extern** and a return type of **int**. It can also be declared to return **void**.

main

In C++, you cannot declare **main** as **inline** or **static**. You cannot call **main** from within a program or take the address of **main**.

Arguments to main

The function **main** can be declared with or without parameters. Although any name can be given to these parameters, they are usually referred to as *argc* and *argv*.

The first parameter, *argc* (argument count), has type **int** and indicates how many arguments were entered on the command line.

The second parameter, *argv* (argument vector), has type array of pointers to **char** array objects. **char** array objects are null-terminated strings.

The value of *argc* indicates the number of pointers in the array *argv*. If a program name is available, the first element in *argv* points to a character array that contains the program name or the invocation name of the program that is being run. If the name cannot be determined, the first element in *argv* points to a null character.

This name is counted as one of the arguments to the function **main**. For example, if only the program name is entered on the command line, *argc* has a value of 1 and *argv[0]* points to the program name.

Regardless of the number of arguments entered on the command line, *argv[argc]* always contains NULL.

Example of Arguments to main

The following program `backward` prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s ", argv[argc]);
}
```

Invoking this program from a command line with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```


Calling Functions and Passing Arguments

The arguments *argc* and *argv* would contain the following values:

Object	Value
<i>argc</i>	3
<i>argv</i> [0]	pointer to string "backward"
<i>argv</i> [1]	pointer to string "string1"
<i>argv</i> [2]	pointer to string "string2"
<i>argv</i> [3]	NULL

Note: Be careful when entering mixed case characters on a command line because some environments are not case sensitive. Also, the exact format of the string pointed to by *argv*[0] is system dependent.

Related Information

- "Calling Functions and Passing Arguments"
- "Parameter Declaration List Syntax" on page 145
- "Type Specifiers" on page 45
- "Identifiers" on page 18
- "Block" on page 166

Calling Functions and Passing Arguments

A function call specifies a function name and a list of arguments. The calling function passes the value of each argument to the specified function. The argument list is surrounded by parentheses, and each argument is separated by a comma. The argument list can be empty. When a function is called, the *actual* arguments are used to initialize the *formal* arguments.

The type of an actual argument is checked against the type of the corresponding formal argument in the function prototype. All standard and user-defined type conversions are applied as necessary.

For example:

```
#include <iostream.h>
#include <math.h>
extern double root(double, double); // declaration
double root(double value, double base) // definition
{
    double temp = exp(log(value)/base);
    return temp;
}
void main()
{
    int value = 144;
    int base = 2;
    // Call function root and print return value
    cout << "The root is: " << root(value,base) << endl;
}
```

Calling Functions and Passing Arguments

The output is The root is: 12

In the above example, because the function `root` is expecting arguments of type **double**, the two **int** arguments `value` and `base` are implicitly converted to type **double** when the function is called.

The arguments to a function are evaluated before the function is called. When an argument is passed in a function call, the function receives a copy of the argument value. If the value of the argument is an address, the called function can use indirection to change the contents pointed to by the address. If a function or array is passed as an argument, the argument is converted to a pointer that points to the function or array.

Arguments passed to parameters in prototype declarations will be converted to the declared parameter type. For nonprototype function declarations, **char** and **short** parameters are promoted to **int**, and **float** to **double**.

You can pass a packed structure argument to a function expecting a nonpacked structure of the same type and vice versa. (The same applies to packed and nonpacked unions.)

Note: If you do not use a function prototype and you send a packed structure when a nonpacked structure is expected, a runtime error may occur.

The order in which arguments are evaluated and passed to the function is implementation-defined. For example, the following sequence of statements calls the function `tester`:

```
int x;  
x = 1;  
tester(x++, x);
```

The call to `tester` in the example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first or `x` may be evaluated first. To avoid the ambiguity and have `x++` evaluated first, replace the preceding sequence of statements with the following:

```
int x, y;  
x = 1;  
y = x++;  
tester(y, x);
```

In C++, if a nonstatic class member function is passed as an argument, the argument is converted to a pointer to member.

If a class has a destructor or a copy constructor that does more than a bitwise copy, passing a class object by value results in the construction of a temporary that is actually passed by reference.

It is an error when a function argument is a class object and all of the following properties hold:

- The class needs a copy constructor.

Calling Functions and Passing Arguments

- The class does not have a user-defined copy constructor.
- A copy constructor cannot be generated for that class.

For more information on copy constructors, see “Constructors” on page 294.

Examples of Calling Functions

The following statement calls the function `startup` and passes no parameters:

```
startup();
```

The following function call causes copies of `a` and `b` to be stored in a local area for the function `sum`. The function `sum` runs using the copies of `a` and `b`.

```
sum(a, b);
```

The following function call passes the value 2 and the value of the expression `a + b` to `sum`:

```
sum(2, a + b);
```

The following statement calls the function `printf`, which receives a character string and the return value of the function `sum`, which receives the values of `a` and `b`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of `count` to the function `increment`. `increment` increases the value of the parameter `x` by 1.

```
/**
** This example shows how a parameter is passed to a function
**/

#include <stdio.h>

void increment(int);

int main(void)
{
    int count = 5;

    /* value of count is passed to the function */
    increment(count);
    printf("count = %d\n", count);

    return(0);
}

void increment(int x)
{
    ++x;
    printf("x = %d\n", x);
}
```

Calling Functions and Passing Arguments

The output illustrates that the value of `count` in `main` remains unchanged:

```
x = 6
count = 5
```

In the following program, `main` passes the address of `count` to `increment`. The function `increment` was changed to handle the pointer. The parameter `x` is declared as a pointer. The contents to which `x` points are then incremented.

```
/**
 ** This example shows how an address is passed to a function
 **/

#include <stdio.h>

int main(void)
{
    void increment(int *x);
    int count = 5;

    /* address of count is passed to the function */
    increment(&count);
    printf("count = %d\n", count);

    return(0);
}

void increment(int *x)
{
    ++*x;
    printf("*x = %d\n", *x);
}
```

The output shows that the variable `count` is increased:

```
*x = 6
count = 6
```

Calling Functions and Passing Arguments

Passing Arguments by Reference

If you use a reference type as a formal argument, you can make a pass-by-reference call to a function. In a pass-by-reference call, the values of arguments in the calling function can be modified in the called function. In pass-by-value calls, only copies of the arguments are passed to the function.

Note: The term *pass by reference* describes a general method of passing arguments from a calling routine to a called routine. The term *reference* in the context of C++ refers to a specific way of declaring objects and functions.

Arguments by reference in the following two cases causes the compiler to issue a warning:

- Ellipsis arguments cannot be passed as references.
- If the last argument specified in the function declaration before the ellipsis is a reference argument, arguments passed using an ellipsis (variable arguments) are not accessible using the mechanism from the `<stdarg.h>` standard header file.

In addition, when the actual argument cannot be referenced directly by the formal argument, the compiler creates a temporary variable that is referenced by the formal argument and initialized using the value of the actual argument. In this case, the formal argument must be a **const** reference.

Reference arguments declared **const** can be used to pass large objects efficiently to functions without making a temporary copy of the object that is passed to the function. Because the reference is declared **const**, the actual arguments cannot be changed by the function. For example:

```
void printbig (const bigvar&); // Function prototype
```

When a function `printbig` is called, it cannot modify the object of type `bigvar` because the object was passed by constant reference.

Calling Functions and Passing Arguments

The following example shows how arguments are passed by reference. Note that reference formal arguments are initialized with the actual arguments when the function is called.

```
/**
 ** This example shows how arguments are passed by reference
 **/

#include <iostream.h>
void swapnum(int &i, int &j)
{
    int temp = i;
    i = j;
    j = temp;
}
// .
// .
// .
main()
{
    int a = 10, // a is 10
        b = 20; // b is 20
    swapnum(a,b); // now a is 20 and b is 10
    cout << "A is : " << a
         << " and B is : " << b << endl;
}
```

When the function `swapnum()` is called, the actual values of the variables `a` and `b` are exchanged because they are passed by reference. The output is:

```
A is : 20 and B is : 10
```

You must define the formal arguments of `swapnum()` as references if you want the values of the actual arguments to be modified by the function `swapnum()`.

Default Arguments in C++ Functions

Default Arguments in C++ Functions

You can provide default values for function arguments. All default argument names of a function are bound when the function is declared. All functions have their types checked at declaration, and are evaluated at each point of call.

For example:

```
/**
 ** This example illustrates default function arguments
 **/

#include <iostream.h>
int a = 1;
int f(int a) {return a;}
int g(int x = f(a)) {return f(a);}

int h()
{
    a=2;
    {
        int a = 3;
        return g();
    }
}

main()
{
    cout << h() << endl;
}
```

This example prints 2 to standard output, because the `a` referred to in the declaration of `g()` is the one at file scope, which has the value 2 when `g()` is called. The value of `a` is determined after entry into function `h()` but before the call to `g()` is resolved.

A default argument can have any type.

A pointer to a function must have the same type as the function. Attempts to take the address of a function by reference without specifying the type of the function produce an error. The type of a function is not affected by arguments with default values.

Default Arguments in C++ Functions

The following example shows that a function with default arguments does not change its type. The default argument allows you to call a function without specifying all of the arguments, it does not allow you to create a pointer to the function that does not specify the types of all the arguments. Function `f` can be called without an explicit argument, but the pointer `badpointer` cannot be defined without specifying the type of the argument:

```
int f(int = 0);
void g()
{
    int a = f(1);           // ok
    int b = f();           // ok, default argument used
}
int (*pointer)(int) = &f;  // ok, type of f() specified (int)
int (*badpointer)() = &f; // error, badpointer and f have
                          // different types. badpointer must
                          // be initialized with a pointer to
                          // a function taking no arguments.
```

Restrictions on Default Arguments

Of the operators, only the function call operator and the operator **new** can have default arguments when they are overloaded.

Arguments with default values must be the trailing arguments in the function declaration argument list. For example:

```
void f(int a, int b = 2, int c = 3); // trailing defaults
void g(int a = 1, int b = 2, int c); // error, leading defaults
void h(int a, int b = 3, int c);    // error, default in middle
```

Once a default argument has been given in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments not given in previous declarations. For example, the last declaration below attempts to redefine the default values for `a` and `b`:

```
void f(int a, int b, int c=1); // valid
void f(int a, int b=1, int c); // valid, add another default
void f(int a=1, int b, int c); // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

You can supply any default argument values in the function declaration or in the definition. All subsequent arguments must have default arguments supplied in this or a previous declaration of the function.

You cannot use local variables in default argument expressions. For example, the compiler generates errors for both function `g()` and function `h()` below:

Default Arguments in C++ Functions

```
void f(int a)
{
    int b=4;
    void g(int c=a); // Local variable "a" inaccessible
    void h(int d=b); // Local variable "b" inaccessible
}
```

Evaluating Default Arguments

When a function defined with default arguments is called with trailing arguments missing, the default expressions are evaluated. For example:

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);           // same as call f(a,2,3)
f(a,10);        // same as call f(a,10,3)
f(a,10,20);     // no default arguments
```

Default arguments are checked against the function declaration and evaluated when the function is called. The order of evaluation of default arguments is undefined. Default argument expressions cannot use formal arguments of a function. For example:

```
int f(int q = 3, int r = q); // error
```

The argument `r` cannot be initialized with the value of the argument `q` because the value of `q` may not be known when it is assigned to `r`. If the above function declaration is rewritten:

```
int q=5;
int f(int q = 3, int r = q); // error
```

the value of `r` in the function declaration still produces an error because the variable `q` defined outside of the function is hidden by the argument `q` declared for the function.

Similarly:

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

Here the type `D` is interpreted within the function declaration as the name of an integer. The type `D` is hidden by the argument `D`. The cast `D(5.3)` is therefore not interpreted as a cast because `D` is the name of the argument not a type.

In the following example, the nonstatic member `a` cannot be used as an initializer because `a` does not exist until an object of class `X` is constructed. You can use the static member `b` as an initializer because `b` is created independently of any objects of class `X`. You can declare the member `b` after its use as a default argument because the default values are not analyzed until after the final bracket `}` of the class declaration.

Function Return Values

```
class X
{
    int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

You must put parentheses around default argument expressions that contain template references. In the following example:

```
class C {
    void f(int i = X<int,5>::y);
};
```

the compiler cannot tell that the < represents the start of a template argument list and not the less than operator because the default argument `X<int,5>::y` cannot be processed until the end of the class.

To avoid error messages, put parentheses around the expression containing the default argument:

```
class C {
    void f( int i = (X<int,5>::y) );
};
```

Function Return Values

A value must be returned from a function unless the function has a return type of **void**. The return value is specified in a return statement. The following code fragment shows a function definition, including the **return** statement:

```
int add(int i, int j)
{
    return i + j; // return statement
}
```

The function `add()` can be called as shown in the following code fragment:

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

In this example, the return statement initializes a variable of the returned type. The variable `answer` is initialized with the **int** value 30. The type of the returned expression is checked against the returned type. All standard and user-defined conversions are performed as necessary.

The following **return** statements show different ways of returning values to a caller:

Function Return Values

```
return;                // Returns no value
return result;        // Returns the value of result
return 1;             // Returns the value 1
return (x * x);       // Returns the value of x * x
```

Other than **main()**, if a function that does not have type **void** returns without a value (as in the first **return** statement shown in the example above) the result returned is undefined. In C++, the compiler issues an error message as well.

If **main** has a return type of `int`, and does not contain a return expression, it returns the value zero.

Each time a function is called, new copies of its local variables are created. Because the storage for a local variable may be reused after the function has terminated, a pointer to a local variable or a reference to a local variable should not be returned.

If a class object is returned, a temporary object may be created if the class has copy constructors or a destructor. For more information, see “Temporary Objects” on page 302.

Using References as Return Types

References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers. This allows you to place function calls on the left side of assignment statements. Referenced return values are used when assignment operators and subscripting operators are overloaded so that the results of the overloaded operators can be used as actual values.

Note: Returning a reference to an automatic variable gives unpredictable results.

For more information, see “Special Overloaded Operators” on page 287.

Pointers to Functions

A pointer to a function points to the address of the function's executable code. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

The type of a pointer to a function is based on both the return type and argument types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);      // function f returning an int*
int (*g)(int a);   // pointer g to a function returning an int
```

In the first declaration, `f` is interpreted as a function that takes an `int` as argument, and returns a pointer to an `int`. In the second declaration, `g` is interpreted as a pointer to a function that takes an `int` argument and that returns an `int`.

Under VisualAge for C++ for AS/400, if a function pointer is passed to a function, or returned from a function, the declared or implied linkages must be the same. The `extern` keyword is used with declarations in order to specify different linkages (refer to "extern Storage Class Specifier" on page 35 for more information). Linkages may also be implied, as a result of compiling with the VisualAge for C++ for AS/400 compiler.

The following example illustrates the correct and incorrect uses of function pointers under VisualAge for C++ for AS/400 :

```
#include <stdlib.h>

extern "C"   int cf();
extern "C++" int cxxf(); // C++ is included here for clarity;
                       // it is not required; if it is
                       // omitted, cxxf() will still have
                       // C++ linkage.

extern "C"   int (*c_fp)();
extern "C++" int (*cxx_fp)();

typedef int (*dft_fp_T)();
typedef int (dft_f_T)();

extern "C" {
    typedef void (*cfp_T)();
    typedef int (*cf_pT)();
    void cfn();
    void (*cfp)();
}

extern "C++" {
    typedef int (*cxxf_pT)();
```

Pointers to Functions

```
void cxxfn();
void (*cxxfp)();
}

extern "C" void f_cprm(int (*f)()) {
    int (*s)() = cxxf;    // invalid, incompatible linkages-cxxf has
                        // C++ linkage, s has C linkage as it
                        // is included in the extern "C" wrapper
    cxxf_pT j = cxxf;    // valid, both have C++ linkage
    int (*i)() = cf;     // valid, both have C linkage
}

extern "C++" void f_cxprm(int (*f)()) {
    int (*s)() = cf;     // invalid, incompatible linkages-cf has C
                        // linkage, s has C++ linkage as it is
                        // included in the extern "C++" wrapper
    int (*i)() = cxxf;   // valid, both have C++ linkage
    cf_pT j = cf;       // valid, both have C linkage
}

main() {

    c_fp    = cxxf;      // invalid - c_fp has C linkage and cxxf has
                        // C++ linkage
    cxx_fp  = cf;       // invalid - cxx_fp has C++ linkage and
                        // cf has C linkage

    dft_fp_T dftfpT1 = cf; // invalid - dftfpT1 has C++ linkage and
                        // cf has C linkage
    dft_f_T  *dftfT3 = cf; // invalid - dftfT3 has C++ linkage and
                        // cf has C linkage

    dft_fp_T dftfpT5 = cxxf; // valid
    dft_f_T  *dftfT6 = cxxf; // valid

    c_fp    = cf;       // valid
    cxx_fp  = cxxf;     // valid
    f_cprm(cf);        // valid
    f_cxprm(cxxf);     // valid

    // The following errors are due to incompatible linkage of function
    // arguments, type conversion not possible
    f_cprm(cxxf);      // invalid - f_cprm expects a parameter with
                        // C linkage, but cxxf has C++ linkage
    f_cxprm(cf);       // invalid - f_cxprm expects a parameter
                        // with C++ linkage, but cf has C linkage
}
```

Inline Functions

For more information on pointers, see “Pointers” on page 55 and “Pointer Conversions” on page 133.

Inline Functions

Inline functions are used to reduce the overhead of a normal function call. A function is declared inline by using the **inline** function specifier or by defining a member function within a class or structure definition.

The **inline** specifier is a suggestion to the compiler that an inline expansion can be performed. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call.

An inline function can be declared and defined simultaneously. If it is declared with the keyword **inline**, it can be declared without a definition. The following code fragment shows an inline function definition. Note that the definition includes both the declaration and body of the inline function.

```
inline int add(int i, int j) { return i + j; }
```

Both member and nonmember functions can be inline, and both have internal linkage.

The use of the inline specifier does not change the meaning of the function. The inline expansion of a function may not preserve the order of evaluation of the actual arguments.

For more information on inlining, see “Inline Member Functions” on page 257.

Inline Functions

Chapter 7. Statements

This chapter describes the following C++ language statements:

Labels	165
Block	166
break	168
continue	171
do	173
Expression	175
for	177
goto	179
if	180
Null Statement	182
return	182
switch	184
while	189

Related Information

- Chapter 3, “Declarations” on page 29
- Chapter 4, “Expressions and Operators” on page 93
- Chapter 6, “Functions” on page 137

Labels

A *label* is an identifier that allows your program to transfer control to other statements within the same function. It is the only type of identifier that has function scope. Control is transferred to the statement following the label by means of the **goto** or **switch** statements. In VisualAge for C++ for AS/400, control can also be transferred to a label when an exception occurs while the function containing the label is running, if the label has been named as the branch point handler on a **#pragma exception_handler** directive.

A labelled statement has the form:

```
▶▶—identifier—:—statement—▶▶
```

The label is the *identifier* and the colon (:) character.

The **case** and **default** labels can only appear within the body of a **switch** statement.

Examples

```
comment_complete : ;          /* null statement label */
test_for_null : if (NULL == pointer)
```


Block Statement

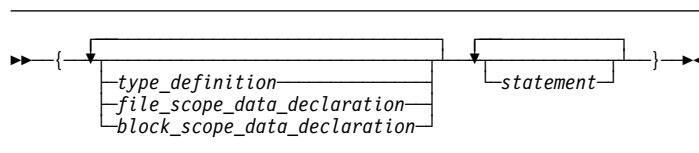
Related Information

- “goto” on page 179
- “switch” on page 184

Block

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

A block statement has the form:



In C, Any definitions and declarations must come before the statements.

Redefining a data object inside a nested block hides the outer object while the inner block runs. Defining several variables that have the same identifier can make a program difficult to understand and maintain. You should avoid redefining of identifiers within nested blocks. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Initialization within Block Statements

Initialization of an **auto** or **register** variable occurs each time the block is run from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed. You cannot initialize an **extern** variable within a block.

A **static** local object is initialized only once, when control passes through its declaration for the first time. A **static** variable initialized with an expression other than a constant expression is initialized to 0 before its block is first entered.

C++ Note: Unlike ANSI/ISO C, in C++ it is an error to jump over a declaration or definition containing an initializer. For example, the following code produces an error in C++:

```
goto skiplabel;
int i=3 // error, jumped over declaration of i with initializer
skiplabel: i=4;
```

When control exits from a block, all objects with destructors that are defined in the block are destroyed. The destructor for a **static** local object is called only if the object was constructed. The destructor must be called before or as part of the **atexit** function.

Block Statement

Local variables declared in a block are also destroyed on exit. Automatic variables defined in a loop are destroyed at each iteration.

Example

The following program shows how the values of data objects change in nested blocks:

```
1 /**
2  ** This example shows how data objects change in nested blocks.
3  **/
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int x = 1;                /* Initialize x to 1 */
9      int y = 3;
10
11     if (y > 0)
12     {
13         int x = 2;            /* Initialize x to 2 */
14         printf("second x = %4d\n", x);
15     }
16     printf("first x = %4d\n", x);
17
18     return(0);
19 }
```

The program produces the following output:

```
second x =    2
first x =    1
```

Two variables named `x` are defined in `main`. The definition of `x` on line 8 retains storage while `main` is running. However, because the definition of `x` on line 13 occurs within a nested block, line 14 recognizes `x` as the variable defined on line 13. Because line 16 is not part of the nested block, `x` is recognized as the variable defined on line 8.

Related Information

- “Storage Class Specifiers” on page 31
- “Type Specifiers” on page 45

break Statement

break

A *break statement* lets you end an *iterative* (**do**, **for**, **while**) or **switch** statement and exit from it at any point other than the logical end.

A **break** statement has the form:

```
▶▶—break—;—▶▶
```

In an iterative statement the **break** statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the **break** statement ends only the smallest enclosing **do**, **for**, **switch**, or **while** statement.

In a **switch** body, the **break** passes control out of the **switch** body to the next statement outside the **switch** body.

Restrictions

A **break** statement can only appear in the body of an iterative statement or a **switch** statement.

Examples

The following example shows a **break** statement in the action part of a **for** statement. If the *i*th element of the array *string* is equal to `'\0'`, the **break** statement causes the **for** statement to end.

```
for (i = 0; i < 5; i++)
{
    if (string[i] == '\0')
        break;
    length++;
}
```

The following is an equivalent **for** statement, if *string* does not contain any embedded null characters:

```
for (i = 0; (i < 5)&& (string[i] != '\0'); i++)
{
    length++;
}
```

break Statement

The following example shows a **break** statement in a nested iterative statement. The outer loop goes through an array of pointers to strings. The inner loop examines each character of the string. When the **break** statement is processed, the inner loop ends and control returns to the outer loop.

```
/**
** This program counts the characters in the strings that are
** part of an array of pointers to characters. The count stops
** when one of the digits 0 through 9 is encountered
** and resumes at the beginning of the next string.
**/

#include <stdio.h>
#define SIZE 3

int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;

    for (i = 0; i < SIZE; i++)          /* for each string */
        for (pointer = strings[i]; *pointer != '\0'; ++pointer) /* for each character */
        {                               /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                break;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);

    return(0);
}
```

The program produces the following output:

```
letter count = 4
```

break Statement

The following example is a **switch** statement that contains several **break** statements. Each **break** statement indicates the end of a specific clause and ends the **switch** statement.

```
/**
** This example shows a switch statement with break statements.
**/

#include <stdio.h>

enum {morning, afternoon, evening} timeofday = morning;

int main(void) {

    switch (timeofday) {
        case (morning):
            printf("Good Morning\n");
            break;

        case (evening):
            printf("Good Evening\n");
            break;

        default:
            printf("Good Day, eh\n");
    }
}
```

Related Information

- “do” on page 173
- “for” on page 177
- “switch” on page 184
- “while” on page 189

continue Statement

continue

A *continue statement* lets you end the current iteration of a loop. Program control is passed from the **continue** statement to the end of the loop body.

A **continue** statement has the form:

```
▶▶—continue—;—▶▶
```

The **continue** statement ends the processing of the action part of an iterative (**do**, **for**, or **while**) statement and moves control to the condition part of the statement. If the iterative statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the **continue** statement ends only the current iteration of the **do**, **for**, or **while** statement immediately enclosing it.

Restrictions

A **continue** statement can only appear within the body of an iterative statement.

Examples

The following example shows a **continue** statement in a **for** statement. The **continue** statement causes processing to skip over those elements of the array rates that have values less than or equal to 1.

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define SIZE 5

int main(void)
{
    int i;
    static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

    printf("Rates over 1.00\n");
    for (i = 0; i < SIZE; i++)
    {
        if (rates[i] <= 1.00) /* skip rates <= 1.00 */
            continue;
        printf("rate = %.2f\n", rates[i]);
    }

    return(0);
}
```

continue Statement

The program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a **continue** statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the `'\0'` escape sequence is encountered.

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters. The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define SIZE 3

int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < SIZE; i++)                /* for each string */
        for (pointer = strings[i]; *pointer != '\0'; ++pointer) /* for each each character */
        {
            if (*pointer >= '0' && *pointer <= '9') /* if a number */
                continue;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);

    return(0);
}
```

do Statement

The program produces the following output:

```
letter count = 5
```

Compare this program with the program on page 169, which shows the use of the **break** statement to perform a similar function.

Related Information

- “do”
- “for” on page 177
- “while” on page 189

do

A *do statement* repeatedly runs a statement until the test expression evaluates to 0. Because of the order of processing, the statement is run at least once.

A **do** statement has the form:

```
►►do—statement—while—(—expression—);►◄
```

The body of the loop is run before the controlling **while** clause is evaluated. Further processing of the **do** statement depends on the value of the **while** clause. If the **while** clause does not evaluate to 0, the statement runs again. When the **while** clause evaluates to 0, the statement ends. The controlling expression must be evaluate to a scalar type.

A **break**, **return**, or **goto** statement can cause the processing of a **do** statement to end, even when the **while** clause does not evaluate to 0.

do Statement

Example

The following statement prompts the user to enter a 1. If the user enters a 1, the statement ends. If not, it displays another prompt. The example contains error-checking code to verify that the user entered an integer value and to clear the input stream if an error occurs.

```
/**
 ** This example illustrates the do statement.
 **/

#include <iostream.h>
void main()
{
    int reply1;
    char c;
    do
    {
        cout << "Enter a 1: ";
        cin >> reply1;
        if (cin.fail())
        {
            cerr << "Not a valid number." << endl;
            // Clear the error flag.
            cin.clear(cin.rdstate() & ~ios::failbit);
            // Purge incorrect input.
            cin.ignore(cin.rdbuf()->in_avail());
        }
    }
    while (reply1 != 1);
}
```

Related Information

- “break” on page 168
- “continue” on page 171
- “while” on page 189

Expression

An *expression statement* contains an expression. The expression can be null. Expressions are described in Chapter 4, “Expressions and Operators” on page 93.

An expression statement has the form:

```

>> ┌──────────┐ ──▶; ──▶
    │ expression │
  
```

An expression statement evaluates the given expression. It is used to assign the value of the expression to a variable or to call a function.

Examples

```

printf("Account Number: \n");           /* call to the printf */
marks = dollars * exch_rate;            /* assignment to marks */
(difference < 0) ? ++losses : ++gain;    /* conditional increment */
switches = flags | BIT_MASK;           /* assignment to switches */
  
```

Resolving Ambiguous Statements in C++

There are situations in C++ where a statement can be parsed as both a declaration and as an expression. Specifically, a declaration can look like a function call in certain cases. The compiler resolves these ambiguities by applying the following rules to the *whole* statement:

- If the statement can be parsed as a declaration but there are no declaration specifiers in the declaration and the statement is inside the body of a function, the statement is assumed to be an expression.

The following statement, for example, is a declaration at file scope of the function `f()` that returns type `int`. There is no declaration specifier and `int` is the default, but at function scope this is a call to `f()`:

```
f();
```

- In every other case, if the statement can be parsed as a declaration, it is assumed to be a declaration. The following statement, for example, is a declaration of `x` with redundant parentheses around the declarator, not a function-style cast of `x` to type `int`:

```
int (x);
```

In some cases, C++ syntax does not disambiguate between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its leftmost subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.

Expression

Note: The ambiguity is resolved only on a syntactic level. The disambiguation does not use the meaning of the names, except to assess whether or not they are type names.

The following expressions disambiguate into expression statements because the ambiguous subexpression is followed by an assignment or an operator. `type_spec` in the expressions can be any type specifier:

```
type_spec(i)++;           // expression statement
type_spec(i,3)<<d;        // expression statement
type_spec(i)->l=24;       // expression statement
```

In the following examples, the ambiguity cannot be resolved syntactically, and the statements are interpreted as declarations. `type_spec` is any type specifier:

```
type_spec(*i)(int);       // declaration
type_spec(j)[5];          // declaration
type_spec(m) = { 1, 2 };  // declaration
type_spec(*k) (float(3)); // declaration
```

The last statement above causes a compile-time error because you cannot initialize a pointer with a float value.

Any ambiguous statement that is not resolved by the above rules is by default a declaration statement. All of the following are declaration statements:

```
type_spec(a);             // declaration
type_spec(*b)();          // declaration
type_spec(c)=23;          // declaration
type_spec(d),e,f,g=0;     // declaration
type_spec(h)(e,3);        // declaration
```

Another C++ ambiguity between expression statements and declaration statements is resolved by requiring an explicit return type for function declarations within a block:

```
a();           // declaration of a function returning int
               // and taking no arguments
void func()
{
    int a(); // declaration of a function
    int b;   // declaration of a variable
    a();     // expression-statement calling function a()
    b;       // expression-statement referring to a variable
}
```

The last statement above does not produce any action. It is semantically equivalent to a null statement. However, it is a valid C++ statement.

for Statement

for

A *for statement* lets you do the following:

- Evaluate an expression before the first iteration of the statement (*initialization*)
- Specify an expression to determine whether or not the statement should be processed (*controlling part*)
- Evaluate an expression after each iteration of the statement
- Repeatedly process the statement if the controlling part does not evaluate to zero.

A **for** statement has the form:

```
▶▶ for ( [expression1] ; [expression2] ; [expression3] ) statement ▶▶
```

Expression1 Is the *initialization expression*. It is evaluated only before the *statement* is processed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

Expression2 Is the *controlling part*. It is evaluated before each iteration of the *statement*. It must evaluate to a scalar type.

If it evaluates to 0 (zero), the statement is not processed and control moves to the next statement following the **for** statement. If *expression2* does not evaluate to 0, the statement is processed. If you omit *expression2*, it is as if the expression had been replaced by a nonzero constant, and the **for** statement is not terminated by failure of this condition.

Expression3 Is evaluated after each iteration of the *statement*. You can use this expression to increase, decrease, or reinitialize a variable. This expression is optional.

A **break**, **return**, or **goto** statement can cause a **for** statement to end, even when the second expression does not evaluate to 0. If you omit *expression2*, you must use a **break**, **return**, or **goto** statement to end the **for** statement.

C++ Note: In C++ programs, you can also use *expression1* to declare a variable as well as initialize it. If you declare a variable in this expression, the variable has the same scope as the **for** statement and is not local to the **for** statement.

for Statement

Examples

The following **for** statement prints the value of `count` 20 times. The **for** statement initially sets the value of `count` to 1. After each iteration of the statement, `count` is incremented.

```
for (count = 1; count <= 20; count++)
    printf("count = %d\n", count);
```

The following sequence of statements accomplishes the same task. Note the use of the **while** statement instead of the **for** statement.

```
count = 1;
while (count <= 20)
{
    printf("count = %d\n", count);
    count++;
}
```

The following **for** statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
    list[index] = var1 + var2;
    printf("list[%d] = %d\n", index, list[index]);
}
```

The following **for** statement will continue running until `scanf` receives the letter `e`:

```
for (;;)
{
    scanf("%c", &letter);
    if (letter == '\n')
        continue;
    if (letter == 'e')
        break;
    printf("You entered the letter %c\n", letter);
}
```

The following **for** statement contains multiple initializations and increments. The comma operator makes this construction possible. The first comma in the **for** expression is a punctuator for a declaration. It declares and initializes two integers, `i` and `j`. The second comma, a comma operator, allows both `i` and `j` to be incremented at each step through the loop.

```
for (int i = 0, j = 50; i < 10; ++i, j += 50)
{
    cout << "i = " << i << "and j = " << j << endl;
}
```

goto Statement

The following example shows a nested **for** statement. It prints the values of an array having the dimensions [5][3].

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 3; column++)
        printf("%d\n", table[row][column]);
```

The outer statement is processed as long as the value of `row` is less than 5. Each time the outer **for** statement is executed, the inner **for** statement sets the initial value of `column` to zero and the statement of the inner **for** statement is executed 3 times. The inner statement is executed as long as the value of `column` is less than 3.

Related Information

- “break” on page 168
- “continue” on page 171

goto

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the **goto** statement.

A **goto** statement has the form:

```
▶▶—goto—label_identifier—;—▶◀
```

Because the **goto** statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a **break** statement, a **continue** statement, or a function call can eliminate the need for a **goto** statement.

If you use a **goto** statement to transfer control to a statement inside of a loop or block, initializations of automatic storage for the loop do not take place and the result is undefined. The label must appear in the same function as the **goto**.

If an active block is exited using a **goto** statement, any local variables are destroyed when control is transferred from that block.

Example

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

if Statement

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
    int matrix[3][3]={1,2,3,4,5,2,8,9,10};
    display(matrix);
    return(0);
}

void display(int matrix[3][3])
{
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
                goto out_of_bounds;
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    return;
    out_of_bounds: printf("number must be 1 through 6\n");
}
```

if

An *if statement* lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The expression must evaluate to a scalar type. You can optionally specify an **else** clause on the **if** statement. If the test expression evaluates to 0 and an **else** clause exists, the statement associated with the **else** clause runs. If the test expression evaluates to a nonzero value, the statement following the expression runs and the **else** clause is ignored.

An **if** statement has the form:

```
► if (—expression—) —statement— ─┬─ else —statement— ◀
```

When **if** statements are nested and **else** clauses are present, a given **else** is associated with the closest preceding **if** statement within the same block.

if Statement

Examples

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
    grade = 'A';
```

The following example displays Number is positive if the value of number is greater than or equal to 0. If the value of number is less than 0, it displays Number is negative.

```
if (number >= 0)
    printf("Number is positive\n");
else
    printf("Number is negative\n");
```

The following example shows a nested **if** statement:

```
if (paygrade == 7)
    if (level >= 0 && level <= 8)
        salary *= 1.05;
    else
        salary *= 1.04;
else
    salary *= 1.06;
cout << "salary is " << salary << endl;
```

The following example shows a nested **if** statement that does not have an **else** clause. Because an **else** clause always associates with the closest **if** statement, braces might be needed to force a particular **else** clause to associate with the correct **if** statement. In this example, omitting the braces would cause the **else** clause to associate with the nested **if** statement.

```
if (kegs > 0) {
    if (furlongs > kegs)
        fpk = furlongs/kegs;
}
else
    fpk = 0;
```

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire **if** statement ends.

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```


return Statement

Null Statement

The *null statement* performs no operation. It has the form:

```
▶▶;▶▶
```

A null statement can hold the label of a labeled statement or complete the syntax of an iterative statement.

Example

The following example initializes the elements of the array `price`. Because the initializations occur within the **for** expressions, a statement is only needed to finish the **for** syntax; no operations are required.

```
for (i = 0; i < 3; price[i++] = 0)
    ;
```

A null statement can be used when a label is needed before the end of a block statement. For example:

```
void func(void) {
    if (error_detected)
        goto depart;
    /* further processing */
    depart: ; /* null statement required */
}
```

return

A *return statement* ends the processing of the current function and returns control to the caller of the function.

A **return** statement has the form:

```
▶▶return expression ;▶▶
```

A **return** statement in a function is optional. The compiler issues a warning if a return statement is not found in a function declared with a return type. If the end of a function is reached without encountering a **return** statement, control is passed to the caller as if a **return** statement without an expression were encountered. A function can contain multiple **return** statements.

return Statement

Value of a return Expression and Function Value

If an expression is present on a **return** statement, the value of the expression is returned to the caller. If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

If an expression is not present on a **return** statement, the value of the **return** statement is undefined. If an expression is not given on a **return** statement in a function declared with a nonvoid return type, an error message is issued, and the result of calling the function is unpredictable. For example:

```
int func1()
{
    return;
}
int func2()
{
    return (4321);
}

void main() {
int a=func1(); // result is unpredictable!
int b=func2();
}
```

You cannot use a **return** statement with an expression when the function is declared as returning type **void**.

C++ Note: If a function returns a class object with constructors, a temporary class object might be constructed. The temporary object is not in the scope of the function returning the temporary object but is local to the caller of the function.

When a function returns, all temporary local variables are destroyed. If local class objects with destructors exist, destructors are called. For more details, see “Temporary Objects” on page 302.

switch Statement

Examples

```
return;           /* Returns no value          */
return result;   /* Returns the value of result */
return 1;        /* Returns the value 1         */
return (x * x);  /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable number. If a match exists, the function match returns the value of i. If a match does not exist, the function match returns the value -1 (negative one).

```
int match(int number, int array[ ], int n)
{
    int i;

    for (i = 0; i < n; i++)
        if (number == array[i])
            return (i);
    return(-1);
}
```

Related Information

- Chapter 6, “Functions” on page 137
- “Expression” on page 175

switch

A *switch statement* lets you transfer control to different statements within the **switch** body depending on the value of the switch expression. The **switch** expression must evaluate to an integral value. The body of the **switch** statement contains *case clauses* that consist of

- A **case** label
- An optional **default** label
- A **case** expression
- A list of statements.

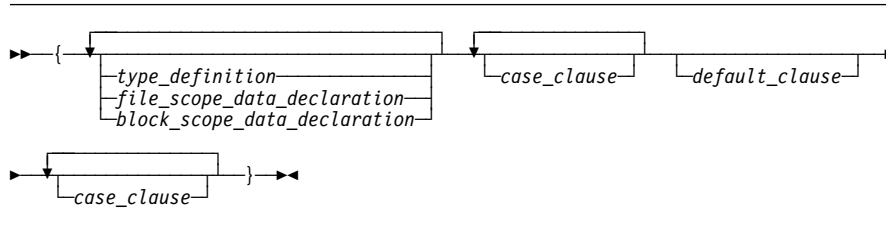
If the value of the **switch** expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

A **switch** statement has the form:

```
►►—switch—(—expression—)—switch_body—►◄
```

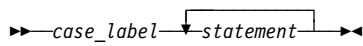
The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.

switch Statement

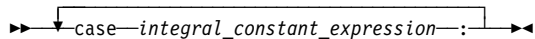


Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

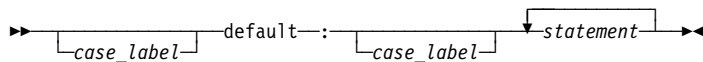
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:



A *case label* contains the word **case** followed by an integral constant expression and a colon. Anywhere you can put one **case** label, you can put multiple **case** labels. A case label has the form:



A *default clause* contains a **default** label followed by one or more statements. You can put a **case** label on either side of the **default** label. A **switch** statement can have only one **default** label. A *default_clause* has the form:



The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the *switch expression*.

The value of the **switch** expression is compared with the value of the expression in each **case** label. If a matching value is found, control is passed to the statement following the **case** label that contains the matching value. If there is no matching value but there is a **default** label in the **switch** body, control passes to the **default** labelled statement. If no matching value is found, and there is no **default** label anywhere in the **switch** body, no part of the **switch** body is processed.

switch Statement

When control passes to a statement in the **switch** body, control only leaves the **switch** body when a **break** statement is encountered or the last statement in the **switch** body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the **case** statements are converted to the same type as the controlling expression.

Restrictions

The **switch** expression and the **case** expressions must have an integral type. The value of each **case** expression must represent a different value and must be a constant expression.

Only one **default** label can occur in each **switch** statement. You cannot have duplicate **case** labels in a **switch** statement.

You can put data definitions at the beginning of the **switch** body, but the compiler does not initialize **auto** and **register** variables at the beginning of a **switch** body. You can have declarations in the body of the **switch** statement. In C++, you cannot transfer control over a declaration containing an initializer unless the declaration is located in an inner block that is completely bypassed by the transfer of control. All declarations within the body of a **switch** statement that contain initializers must be contained in an inner block.

Examples

The following **switch** statement contains several **case** clauses and one **default** clause. Each clause contains a function call and a **break** statement. The **break** statements prevent control from passing down through each statement in the **switch** body.

If the **switch** expression evaluated to `'/'`, the switch statement would call the function `divide`. Control would then pass to the statement following the **switch** body.

switch Statement

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
    case '+':
        add();
        break;

    case '-':
        subtract();
        break;

    case '*':
        multiply();
        break;

    case '/':
        divide();
        break;

    default:
        printf("invalid key\n");
        break;
}
```

If the switch expression matches a case expression, the statements following the case expression are processed until a **break** statement is encountered or the end of the **switch** body is reached. In the following example, **break** statements are not present. If the value of `text[i]` is equal to 'A', all three counters are incremented. If the value of `text[i]` is equal to 'a', `lettera` and `total` are increased. Only `total` is increased if `text[i]` is not equal to 'A' or 'a'.

```
char text[100];
int capa, lettera, total;

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
        case 'A':
            capa++;
        case 'a':
            lettera++;
        default:
            total++;
    }
}
```

switch Statement

The following **switch** statement performs the same statements for more than one **case** label:

```
/**
** This example contains a switch statement that performs
** the same statement for more than one case label.
**/

#include <stdio.h>

int main(void)
{
    int month;

    /* Read in a month value */
    printf("Enter month: ");
    scanf("%d", &month);

    /* Tell what season it falls into */
    switch (month)
    {
        case 12:
        case 1:
        case 2:
            printf("month %d is a winter month\n", month);
            break;

        case 3:
        case 4:
        case 5:
            printf("month %d is a spring month\n", month);
            break;

        case 6:
        case 7:
        case 8:
            printf("month %d is a summer month\n", month);
            break;

        case 9:
        case 10:
        case 11:
            printf("month %d is a fall month\n", month);
            break;

        case 66:
        case 99:
        default:
            printf("month %d is not a valid month\n", month);
    }
}
```

while Statement

```
    return(0);  
}
```

If the expression `month` has the value 3, control passes to the statement:

```
printf("month %d is a spring month\n", month);
```

The **break** statement passes control to the statement following the **switch** body.

Related Information

- “break” on page 168.

while

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to 0.

A **while** statement has the form:

```
▶▶ while (—expression—) statement ▶▶
```

The expression is evaluated to determine whether or not to process the body of the loop. The expression must be convertible to a scalar type. If the expression evaluates to 0, the body of the loop never runs. If the expression does not evaluate to 0, the loop body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

A **break**, **return**, or **goto** statement can cause a **while** statement to end, even when the condition does not evaluate to 0.

while Statement

Example

In the following program, `item[index]` triples each time the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the **while** statement ends.

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
    static int item[ ] = { 12, 55, 62, 85, 102 };
    int index = 0;

    while (index < MAX_INDEX)
    {
        item[index] *= 3;
        printf("item[%d] = %d\n", index, item[index]);
        ++index;
    }

    return(0);
}
```

Related Information

- “break” on page 168
- “goto” on page 179
- “return” on page 182

Chapter 8. Preprocessor Directives

This chapter describes the C preprocessor directives. This chapter discusses the following topics:

Preprocessor Overview	191
Preprocessor Directive Format	192
Macro Definition and Expansion (#define)	192
Scope of Macro Names (#undef)	197
# Operator	198
Macro Concatenation with the ## Operator	199
Preprocessor Error Directive (#error)	201
File Inclusion (#include)	201
Predefined Macro Names	203
Conditional Compilation Directives	207
Line Control (#line)	211
Null Directive (#)	213
Pragma Directives (#pragma)	213

Preprocessor Overview

Preprocessing is a step that takes place before compilation that lets you:

- Replace tokens in the current file with specified replacement tokens.
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file.

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C or C++ program.

The preprocessor is controlled by the following directives:

#define	Defines a preprocessor directive.
#undef	Removes a preprocessor macro definition.
#error	Defines text for a compile-time error message.
#include	Inserts text from another source file.
#if	Conditionally suppresses portions of source code, depending on the result of a constant expression.
#ifdef	Conditionally includes source text if a macro name is defined.

#define

#ifndef	Conditionally includes source text if a macro name is not defined.
#else	Conditionally includes source text if the previous #if , #ifdef , #ifndef , or #elif test fails.
#elif	Conditionally includes source text if the previous #if , #ifdef , #ifndef , or #elif test fails, depending on the value of a constant expression.
#endif	Ends conditional text.
#line	Supplies a line number for compiler messages.
#pragma	Specifies implementation-defined instructions to the compiler.

The format of a preprocessor directive is described in “Preprocessor Directive Format.”

Preprocessor Directive Format

Preprocessor directives begin with the **#** token followed by a preprocessor keyword. The **#** token must appear as the first character that is not white space on a line. The **#** is not part of the directive name and can be separated from the name with white spaces.

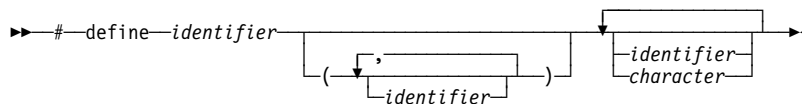
A preprocessor directive ends at the new-line character unless the last character of the line is the **** (backslash) character. If the **** character appears as the last character in the preprocessor line, the preprocessor interprets the **** and the new-line character as a continuation marker. The preprocessor deletes the **** (and the following new-line character) and splices the physical source lines into continuous logical lines.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

Macro Definition and Expansion (#define)

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

A preprocessor **#define** directive has the form:



The **#define** directive can contain an object-like definition or a function-like definition

#define

Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000`:

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value `10.2` but in a syntax error.

```
#define a 10  
a.2
```

Using the following also results in a syntax error:

```
#define a 10  
#define b a.11
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg  
abcd
```

#define

Function-Like Macros

Function-like macro definition:

An identifier followed by a parenthesized parameter list in parenthesis and the replacement tokens. The parameters are imbedded in the replacement code. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, you should not have more than 31 parameters for a macro.

Function-like macro invocation:

An identifier followed by a list of arguments in parentheses. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

The following line defines the macro SUM as having two parameters a and b and the replacement tokens (a + b):

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);  
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);  
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c) ((c) * (c))
```

requires parentheses around each parameter c in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

#define

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

See “Operator Precedence and Associativity” on page 93, and “Parenthesized Expressions ()” on page 98 for more information about using parentheses.

Arguments of the # and ## operators are converted *before* replacement of parameters in a function-like macro.

The number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition.

Commas in the macro invocation argument list do not act as argument separators when they are:

- in character constants
- in string literals
- surrounded by parentheses.

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and does not end until a corresponding **#undef** directive is encountered. If there is no corresponding **#undef** directive, the scope of the macro definition lasts until the end of the compilation unit.

A recursive macro is not fully expanded. For example, the definition

```
#define x(a,b) x(a+1,b+1) + 4
```

expands

```
x(20,10)
```

to

```
x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a call to function `x()`.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor **#define** directive only if the second preprocessor **#define** directive is preceded by a preprocessor **#undef** directive, described in “Scope of Macro Names (#undef)” on page 197. The **#undef** directive nullifies the first definition so that the same identifier can be used in a redefinition.

#define

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

Examples of #define Directives

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s) ((s) * (s))
#define PRNT(a,b) \
    printf("value 1 = %d\n", a); \
    printf("value 2 = %d\n", b) ;

int main(void)
{
    int x = 2;
    int y = 3;

    PRNT(SQR(x),y);

    return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
    int x = 2;
    int y = 3;

    printf("value 1 = %d\n", ( (x) * (x) ) );
    printf("value 2 = %d\n", y);

    return(0);
}
```

#undef

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Related Information

- “Scope of Macro Names (#undef)”
- “# Operator” on page 198
- “Macro Concatenation with the ## Operator” on page 199

Scope of Macro Names (#undef)

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

A preprocessor **#undef** directive has the form:

```
▶▶ #undef identifier ▶▶
```

If the identifier is not currently defined as a macro, **#undef** is ignored

Examples of #undef Directives

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these **#undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **#undef** directive, the identifier can be used in a new **#define** directive.

Related Information

- “Macro Definition and Expansion (#define)” on page 192

Operator

Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
#define ABC(x) #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC>Hello there)	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function-like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro is replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.
- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- If the argument passed to the macro contains a ' (single quotation mark) character, a \ character is inserted before the ' when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

See "Function-Like Macros" on page 194 for more information about function-like macros.

Operator

Examples of the # Operator

The following examples demonstrate the use of the # operator:

```
#define STR(x)      #x
#define XSTR(x)     STR(x)
#define ONE        1
```

Invocation	Result of Macro Expansion
STR(\n " \n" ' \n')	"\n \\"\\n\" ' \\n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"\hello\""

Related Information

- "Macro Definition and Expansion (#define)" on page 192
- "Scope of Macro Names (#undef)" on page 197

Macro Concatenation with the ## Operator

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

For example,

Invocation	Result of Macro Expansion
XY(1, 2)	12
XY(Green, house)	Greenhouse

Operator

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Examples of the ## Operator

The following examples demonstrate the use of the ## operator:

```
#define ArgArg(x, y)      x##y
#define ArgText(x)       x##TEXT
#define TextArg(x)       TEXT##x
#define TextText         TEXT##text
#define Jitter           1
#define bug              2
#define Jitterbug        3
```

Invocation	Result of Macro Expansion
ArgArg(lady, bug)	"ladybug"
ArgText(con)	"conTEXT"
TextArg(book)	"TEXTbook"
TextText	"TEXTtext"
ArgArg(Jitter, bug)	3

Related Information

- "Macro Definition and Expansion (#define)" on page 192

Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

The **#error** directive has the form:

```
▶▶ #error → character →▶▶
```

Use the **#error** directive as a safety check during compilation. For example, if your program uses preprocessor conditional compilation directives, put **#error** directives in the source file to prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#error Error in TESTPGM1 - This section should not be compiled
```

generates the following error message:

```
Error in TESTPGM1 - This section should not be compiled
```

Related Information

- “Conditional Compilation Directives” on page 207

File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

A preprocessor **#include** directive has the form:

```
▶▶ #include → "file_name" →▶▶  
                  <file_name>  
                  <header_name>  
                  identifiers
```

The preprocessor resolves macros contained in a **#include** directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >.

#include

For example:

```
#define MONTH <july.h>
#include MONTH
```

If the file name is enclosed in double quotation marks, for example: `#include "payroll.h"` the preprocessor treats it as a user-defined file, and searches for the file in:

1. The directory where the original .cpp source file resides.
2. Any directories specified using the
3. /I compiler option (that have not been removed by the /Xc option). Directories specified in the ICCAS environment variable are searched before those specified on the command line.
4. Any directories specified using the INCLUDE environment variable, provided the /Xi option is not in effect.

If the file name is enclosed in angle brackets, for example: `#include <stdio.h>` it is treated as a system-defined file, and the preprocessor searches the following places in the order given:

1. Any directories specified using the /I compiler option (that have not been removed by the /Xc option). Directories specified in the ICCAS environment variable are searched before those specified on the command line.
2. Any directories specified using the INCLUDE environment variable, provided the /Xi option is not in effect.

Note: If the file name is fully qualified, the preprocessor searches only the directory specified by the name.

A fully qualified path include name must include a double back slash character (\\), for example "c:\\abc\\aaa.h". A relative path include name must also include a double back slash character (\\), for example "proj1\\aaa.h".

The new-line and > characters cannot appear in a file name delimited by < and >. The new-line and " (double quotation marks) character cannot appear in a file name delimited by " and ", although > can.

For more information about include file search paths and compiler options, see the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

Predefined Macros

Declarations that are used by several files can be placed in one file and included with **#include** in each file that uses them. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

In the following example, a **#define** combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A **#include** makes the header file available to the program.

```
#define IO_HEADER <stdio.h>
.
.
.
#include IO_HEADER /* equivalent to specifying #include <stdio.h> */
.
.
.
```

Predefined Macro Names

VisualAge for C++ for AS/400 provides the following predefined macro names:

- “ANSI/ISO Standard Predefined Macro Names”
- “VisualAge for C++ for AS/400 Predefined Macro Names” on page 204

These predefined names cannot be subject to a **#define** or **#undef** preprocessor directive.

ANSI/ISO Standard Predefined Macro Names

Both C and C++ provide the following predefined macro names as specified in the ANSI/ISO C language standard:

Macro Name	Description
<code>__DATE__</code>	A character string literal containing the date when the source file was compiled. The value of <code>__DATE__</code> changes as the compiler processes any include files that are part of your source program. The date is in the form:

Predefined Macros

"Mmm dd yyyy"

where:

Mmm Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).

dd Represents the day. If the day is less than 10, the first d is a blank character.

yyyy Represents the year.

`__FILE__` A character string literal containing the name of the source file.

The value of `__FILE__` changes as the compiler processes include files that are part of your source program. It can be set with the **#line** directive, described in "Line Control (#line)" on page 211.

`__LINE__` An integer representing the current source line number.

The value of `__LINE__` changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the **#line** directive, described in "Line Control (#line)" on page 211.

`__STDC__` The integer 0 (zero) indicates that C++ does not conform to the ANSI/ISO C language standard.

Note: This macro is not defined if the language level is set to anything other than ANSI.

`__TIME__` A character string literal containing the time when the source file was compiled.

The value of `__TIME__` changes as the compiler processes any include files that are part of your source program. The time is in the form:

"hh:mm:ss"

where:

hh Represents the hour.

mm Represents the minutes.

ss Represents the seconds.

`__cplusplus` For C++ programs, this macro is set to the integer 1, indicating that the compiler is a C++ compiler. Note that this macro name has no trailing underscores.

VisualAge for C++ for AS/400 Predefined Macro Names

VisualAge for C++ for AS/400 provides the following predefined macros. The value of all these macros is defined when the corresponding **#pragma** directive or compiler option is used.

Predefined Macros

Macro Name	Description														
<code>__ANSI__</code>	Allows only language constructs that conform to ANSI/ISO C standard. Defined as 1 when using the #pragma langlvl directive or <code>/Sa</code> option.														
<code>__EXTENDED__</code>	Allows additional language constructs provided by the VisualAge for C++ for AS/400 implementation. Defined using the #pragma langlvl directive or <code>/S2</code> option.														
<code>__COMPAT__</code>	Macro defined as 1 when the <code>/Sc</code> compiler option or the #pragma langlvl(compat) directive is specified for C++ language files. Indicates language constructs compatible with earlier versions of the C++ language are allowed. This macro is not defined for C.														
<code>__TEMPINC__</code>	Indicates the template-implementation file method of resolving template functions is being used.														
<code>__TIMESTAMP__</code>	<p>A character string literal containing the date and time when the source file was last modified.</p> <p>The value of <code>__TIMESTAMP__</code> changes as the compiler processes any include files that are part of your source program. The date and time are in the form:</p> <pre>"Day Mmm dd hh:mm:ss yyyy"</pre> <p>where:</p> <table><tbody><tr><td>Day</td><td>Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).</td></tr><tr><td>Mmm</td><td>Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</td></tr><tr><td>dd</td><td>Represents the day. If the day is less than 10, the first d is a blank character.</td></tr><tr><td>hh</td><td>Represents the hour.</td></tr><tr><td>mm</td><td>Represents the minutes.</td></tr><tr><td>ss</td><td>Represents the seconds.</td></tr><tr><td>yyyy</td><td>Represents the year.</td></tr></tbody></table>	Day	Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).	Mmm	Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).	dd	Represents the day. If the day is less than 10, the first d is a blank character.	hh	Represents the hour.	mm	Represents the minutes.	ss	Represents the seconds.	yyyy	Represents the year.
Day	Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).														
Mmm	Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).														
dd	Represents the day. If the day is less than 10, the first d is a blank character.														
hh	Represents the hour.														
mm	Represents the minutes.														
ss	Represents the seconds.														
yyyy	Represents the year.														

Additional VisualAge for C++ for AS/400 Predefined Macros

The macros identified in this section are provided to allow you to write programs that use VisualAge for C++ for AS/400 services. Only those macros identified in this section should be used to request or receive VisualAge for C++ for AS/400 services.

Macro	Description
<code>__ASV3R7__</code>	Indicates source code contains AS/400 V3R7 features. Set to the integer 1. Used when the <code>/ASv3r7</code> compiler option is set.

Predefined Macros

<code>__ASV3R6__</code>	Indicates source code contains AS/400 V3R6 features. Set to the integer 1. Used when the <code>/ASv3r6</code> compiler option is set.
<code>__ASV3R2__</code>	Indicates source code contains AS/400 V3R2 features. Set to the integer 1. Used when the <code>/ASv3r2</code> compiler option is set.
<code>__CHAR_UNSIGNED</code>	Indicates default character type is unsigned. Defined when the #pragma chars(unsigned) directive is in effect, or when the <code>/J+</code> compiler option is set.
<code>__CHAR_SIGNED</code>	Indicates default character type is signed. Defined when the #pragma chars(signed) directive is in effect, or when the <code>/J-</code> compiler option is set.
<code>__DBCS__</code>	Indicates DBCS support is enabled. Defined using the <code>/Sn</code> compiler option.
<code>__FUNCTION__</code>	Indicates the name of the function currently being compiled. For C++ programs, expands to the actual function prototype.
<code>__HWW_INTEL__</code>	Indicates that the host hardware is an Intel** processor.
<code>__HOS_WIN__</code>	Indicates that the host operating system is Windows.
<code>__IBMCPP__</code>	Indicates the version number of the VisualAge for C++ for AS/400 compiler.
<code>__IFS_IO__</code>	Indicates that the C and C++ stream I/O uses the integrated file system.
<code>__POSIX_LOCALE__</code>	Set to the integer 1. Indicates the locale support is POSIX.
<code>__THW_AS400__</code>	Indicates that the target hardware is an AS/400 processor.
<code>__TOS_OS400__</code>	Indicates that the target operating system is OS/400.
<code>__OS400__</code>	Set to the integer 1. Indicates the product is an OS/400 compiler.

The value of the `__IBMCPP__` macro is 300. When you compile C++ code, `__IBMCPP__` is defined. The remaining macros, with the exception of `__FUNCTION__`, are defined when the corresponding **#pragma** directive or compiler option is used.

Examples of Predefined Macros

Related Information

- “Macro Definition and Expansion (`#define`)” on page 192
- “Scope of Macro Names (`#undef`)” on page 197
- “Line Control (`#line`)” on page 211

Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- **#if**
- **#ifdef**
- **#ifndef**
- **#else**
- **#elif**
- **#endif**

The preprocessor conditional compilation directive spans several lines:

- The condition specification line
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#else** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor **#endif** directive

For each **#if**, **#ifdef**, and **#ifndef** directive, there are zero or more **#elif** directives, zero or one **#else** directive, and one matching **#endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if** directive.

```
#ifdef MACNAME
    /* tokens added if MACNAME is defined */
#   if TEST <=10
    /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
    /* tokens added if MACNAME is defined and TEST > 10 */
#   endif
#else
    /* tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

Conditional Compilation

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a **#else** directive, the block following the **#else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **#else** directive, the entire nesting level is ignored.

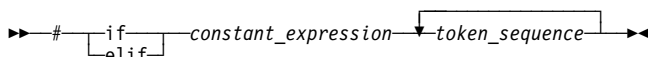
#if, #elif

The **#if** and **#elif** directives compare the value of the expression to zero.

If the constant expression evaluates to a nonzero value, the tokens that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive, the source text located between the **#elif** and the next **#elif** or preprocessor **#else** directive is selected by the preprocessor to be passed on to the compiler. The **#elif** directive cannot appear after the preprocessor **#else** directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token `0`.



The expressions that are tested must be integer constant expressions with the following properties:

- No casts are performed.
- Arithmetic is performed using **long int** values.
- The expression can contain defined macros. No other identifiers can appear in the expression.
- The constant expression can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **#if**. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to 0:

```
defined identifier  
defined(identifier)
```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

Note: If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

Conditional Compilation

```
#if TEST >= 1
    printf("i = %d\n", i);
    printf("array[i] = %d\n", array[i]);
#elif TEST < 0
    printf("array subscript out of bounds \n");
#endif
```

#ifdef

The **#ifdef** directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor **#ifdef** directive has the form:

▶▶ #ifdef *identifier* token_sequence ◀◀

The following example defines `MAX_LEN` to be 75 if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 50.

```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef

The **#ifndef** directive checks for the existence of macro definitions.

If the identifier specified is not defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor **#ifndef** directive has the form:

▶▶ #ifndef *identifier* token_sequence ◀◀

An identifier must follow the **#ifndef** keyword. The following example defines `MAX_LEN` to be 50 if `EXTENDED` is not defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

Conditional Compilation

#else

If the condition specified in the **#if**, **#ifdef**, or **#ifndef** directive evaluates to 0, and the conditional compilation directive contains a preprocessor **#else** directive, the source text located between the preprocessor **#else** directive and the preprocessor **#endif** directive is selected by the preprocessor to be passed on to the compiler.

The preprocessor **#else** directive has the form:

```
▶▶ #else token_sequence ▶▶
```

#endif

The preprocessor **#endif** directive ends the conditional compilation directive.

It has the form:

```
▶▶ #endif ▶▶
```

Examples of Conditional Compilation Directives

The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#       define MAX_PHASE 2
#   else
#       define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

#line

The following program contains preprocessor conditional compilation directives:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
    static int array[ ] = { 1, 2, 3, 4, 5 };
    int i;

    for (i = 0; i <= 4; i++)
    {
        array[i] *= 2;

#ifdef TEST >= 1
        printf("i = %d\n", i);
        printf("array[i] = %d\n", array[i]);
#endif

    }
    return(0);
}
```

Line Control (#line)

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

A preprocessor **#line** directive has the form:

```
▶▶ # [line] [decimal_constant] [file_name] ▶▶
    [characters]
```

In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts **#line** directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

#line

The token sequence on a **#line** directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

Example of #line Directives

You can use **#line** control directives to make the compiler provide more meaningful error messages. The following program uses **#line** control directives to give each function an easily recognizable line number:

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
    func_1();
    func_2();
}

#line 100
func_1()
{
    printf("Func_1 - the current line number is %d\n", __LINE__);
}

#line LINE200
func_2()
{
    printf("Func_2 - the current line number is %d\n", __LINE__);
}
```

This program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

#pragma

Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```
#ifdef MINVAL
#
#else
#define MINVAL 1
#endif
```

Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:

```
▶▶ #pragma character_sequence ◀◀
```

The diagram illustrates the syntax of a pragma directive. It shows the keyword #pragma followed by a character sequence. The character sequence is enclosed in a box with a downward-pointing arrow from the top of the box to the #pragma text. The entire construct is flanked by double arrows pointing outwards.

where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any.

The *character_sequence* on a pragma is subject to macro substitutions. For example,

```
#define XX_ISO_DATA isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

More than one pragma construct can be specified on a single **#pragma** directive. The compiler ignores unrecognized pragmas.

The VisualAge for C++ for AS/400 compiler recognizes the following pragmas:

- “cancel_handler” on page 214
- “chars” on page 215
- “comment” on page 215
- “define” on page 216
- “disable_handler” on page 216
- “enumsize” on page 217
- “exception_handler” on page 218
- “implementation” on page 220
- “langlvl” on page 221
- “mapinc” on page 222
- “operational descriptor” on page 225
- “pack” on page 227

#pragma

- “page” on page 235
- “pagesize” on page 235
- “pointer” on page 235
- “priority” on page 236
- “skip” on page 237
- “strings” on page 237
- “subtitle” on page 238
- “title” on page 238

cancel_handler

The **#pragma cancel_handler** specifies that the function named is to be enabled as a user-defined ILE cancel handler at the point in the code where the **#pragma cancel_handler** directive is located.

```
▶▶ #pragma cancel_handler (—function_name— [, —0— ] —com_area— ) ▶▶
```

Any cancel handler that is enabled by a **#pragma cancel_handler** directive is implicitly disabled when the call to the function containing the directive is finished and is removed from the call stack. Otherwise, the handler can be explicitly disabled using the **#pragma disable_handler** directive.

function The name of the function to be used as a user-defined ILE cancel handler.

com_area A variable used to pass information to the exception handler. If no *com_area* is required, specify zero as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types:

- integral
- pointer
- **float**
- **double**
- **struct**
- **class**
- **union**
- **array**
- **enum**

The *com_area* should be declared with the **volatile** qualifier. The *com_area* cannot be a member of a structure or a union. The *com_area* may be qualified to specify a scope and may be a static member of a class. If the handler is a *function*, then the address of *com_area* is supplied in the exception handler parameter block. If the handler is a *label* then the storage defined by *com_area* is used as the storage for the exception handler parameter block. If the storage required for the exception handler parameter block exceeds the

#pragma

storage defined by *com_area*, then the remaining bytes are truncated.

This pragma can only occur at a C or C++ language statement boundary or inside a function definition.

The compiler issues an error message if any of the following occurs:

- The directive occurs outside a C++ function body or inside a C++ statement, with the exception of a control statement. You can place the directive right after the control statement or right after the expression in an iterative statement.
- The handler function is not declared or defined.
- The identifier named as the handler function is not a function.
- The *com_area* variable is not declared.
- The *com_area* variable does not have a valid object type.

chars

The **#pragma chars** directive specifies that the compiler is to treat all **char** objects as **signed** or **unsigned**.

```
▶▶ #pragma chars ( [ unsigned ] ) ▶▶  
                   [ signed ]
```

This pragma must appear before any statements in a file. Once specified, it applies to the rest of the file and cannot be turned off. If a source file contains any functions that you want to be compiled without **#pragma chars**, place these functions in a different file.

The default character type behaves like an **unsigned char**.

comment

The **#pragma comment** directive places a comment into the program or service program object. This can be shown by DSPPGM or DSPSRVPGM with DETAIL(*COPYRIGHT). This pragma must appear before any C or C++ code or directive in a source file. This pragma has a 256-byte limit.

```
▶▶ #pragma comment ( [ compiler  
                    [ date  
                    [ timestamp  
                    [ copyright  
                    [ user ] [ , "token_sequence" ] ] ] ) ▶▶
```

The comment type can be:

compiler The name and version of the compiler are appended to the end of the generated program object.

#pragma

date	The date and time of compilation are appended to the end of the generated program object.
timestamp	The date and time of the last modification of the source is appended to the end of the generated program object.
copyright	The text specified by the <i>token_sequence</i> is placed by the compiler into the generated program object and is loaded into memory when the program is run.
user	The text specified by the <i>token_sequence</i> is placed by the compiler into the generated object but is <i>not</i> loaded into memory when the program is run.

Notes:

1. The copyright and user comment types are virtually the same on the AS/400. One has no advantage over the other.
2. The maximum number of characters in the text portion of a #pragma comment(copyright) or #pragma comment(user) directive is 256. (This is an AS/400 restriction.)
3. The maximum number of **#pragma comment** directives that can appear in a single compilation unit is 8. (This is an AS/400 restriction.)

define

The **#pragma define** directive forces the definition of a template class without actually defining an object of the class.

```
▶▶ #pragma define (—template_class_name—) ▶▶
```

The pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

disable_handler

The **#pragma disable_handler** directive disables the handler most recently enabled by either the **exception_handler** or **cancel_handler** pragmas.

```
▶▶ #pragma disable_handler ▶▶
```

This directive is only required when a handler has to be explicitly disabled before the end of a function; otherwise, all enabled handlers are implicitly disabled at the end of the function in which they have been enabled.

This pragma can only occur at a C or C++ language statement boundary or inside a function definition. The compiler issues an error message if the **#pragma disable_handler** is specified when no handler is currently enabled.

#pragma

The compiler issues an error message if any of the following occurs:

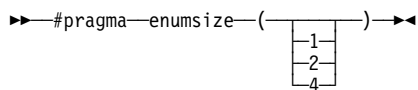
- The directive occurs outside a C++ function body or inside a C++ statement, with the exception of a control statement. You can place the directive right after the control statement or right after the expression in an iterative statement.
- A corresponding **#pragma exception_handler** or the **#pragma cancel_handler** has not been previously enabled.

enumsize

The **#pragma enumsize** directive specifies the number of bytes the compiler uses to represent enumerations. This pragma affects all subsequent enumeration definitions until the end of the compilation unit or until another **#pragma enumsize** directive is encountered. If more than one pragma is used, the most recently encountered pragma is in effect. If the size is not specified in the pragma, the compiler uses the default size, that is, the minimum number of bytes required to represent each value of the enumeration.

If the number of bytes specified in the pragma is less than that required to represent each value of the enumeration, the compiler issues a warning and sets the size of the enumeration type to 4 bytes.

This pragma may appear anywhere that a preprocessor directive is valid.



The size of an enumeration variable is determined by the size of its enumeration type. For example:

```
#include <stdio.h>

#pragma enumsize (4)
enum size {Small, Big}; /* size of enum is 4 bytes */

int main(void)
{
    #pragma enumsize()
    enum length{Long,Short}; /* size of enum is 1 byte */

    enum size s1;
    enum length l1;

    printf("sizeof s1 = %d;size of l1 = %d\n",
           sizeof(s1),sizeof(l1));
}
```

The output of this example is:

```
sizeof s1 = 4; size of l1 = 1
```

#pragma

exception_handler

The **#pragma exception_handler** enables a user-defined ILE exception handler at the point in the code where the **#pragma exception_handler** is located.

```
▶ #pragma exception_handler ( [function_name] [label], [0] [, -class1-, ▶  
                             [,-com_area-]                                ▶  
▶ -class2- [,-ctl_action- [,-msgid_list-] ] ) ▶◀
```

Any exception handlers enabled by **#pragma exception_handler** that are not disabled using **#pragma disable_handler** are implicitly disabled at the end of the function in which they are enabled.

function The name of the function to be used as a user-defined ILE exception handler. The function is passed a single parameter of type pointer to `_INTRPT_Hndl_r_Parms_T`, which is defined in the `<except.h>` header file. The function must be prototyped with a single parameter of this type. The compiler issues a warning if the function is not prototyped with a parameter of type pointer. The function must be defined before the pragma. If an overloaded function is specified, then one of the functions must be prototyped with a single parameter of type pointer to `_INTRPT_Hndl_r_Parms_T`. The *function* name may be qualified to indicate that it is a global function or a static member function.

label A *label* defined within the function to be used as a user-defined ILE exception handler. When the handler gets control, the exception is implicitly handled and control resumes at the *label* defined by the handler in the invocation containing the **#pragma exception_handler** directive. The call stack is cancelled from the newest invocation to, but not including, the invocation containing the **#pragma exception_handler** directive. The *label* can be placed anywhere in the statement part of the function definition, regardless of the position of the **#pragma exception_handler**.

com_area A variable used for the communications area. If no *com_area* is required, specify zero as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types:

- integral
- pointer
- **float**
- **double**
- **struct**
- **class**
- **union**
- **array**
- **enum**

#pragma

The *com_area* should be declared with the **volatile** qualifier. The *com_area* cannot be a member of a structure or a union. The *com_area* may be qualified to specify a scope and may be a static member of a class. If the handler is a *function*, then the address of *com_area* is supplied in the exception handler parameter block. If the handler is a *label*, then the storage defined by *com_area* is used as the storage for the exception handler parameter block. If the storage required for the exception handler parameter block exceeds the storage defined by *com_area*, then the remaining bytes are truncated.

<i>class1, class2</i>	The first four bytes and the last four bytes, respectively, of the exception class mask. The <except.h> header file describes the values that can be used for the class masks and contains macro definitions for these values. <i>class1</i> and <i>class2</i> have to evaluate to integer constant expressions after any necessary macro expansions. You can monitor for the valid <i>class2</i> values of <code>_C2_MH_ESCAPE</code> , <code>_C2_MH_STATUS</code> , <code>_C2_MH_NOTIFY</code> , and <code>_C2_FUNCTION_CHECK</code> .
<i>ctl_action</i>	An integer constant that indicates what action should take place for this exception handler. If the handler is a <i>function</i> , the default value is <code>_CTLA_INVOKE</code> . If the handler is a <i>label</i> , the default value is <code>_CTLA_HANDLE</code> . This parameter is optional.
<i>msgid_list</i>	An optional string literal containing the list of message identifiers that the exception handler monitors. The list is a series of seven-character message identifiers, where the first three characters are the message prefix and the last four characters are the message number. Each message identifier is separated by one or more spaces or commas. This parameter is optional, but if it is specified, <i>ctl_action</i> must also be specified.

For the exception handler to get control, the selection criteria for *class1* and *class2* must be satisfied. If the *msgid_list* is specified, the exception must also match at least one of the message identifiers in the list, based on the following criteria:

- The message identifier matches the exception exactly.
- A message identifier whose two rightmost characters are 00 matches any exception identifier that has the same five leftmost characters. For example, a message identifier of CPF5100 matches any exceptions whose message identifier begins with CPF51.
- A message identifier whose four rightmost characters are 0000 matches any exception identifier that has the same prefix. For example, a message identifier of CPF0000 matches any exception whose message identifier has the prefix CPF (CPF0000 to CPF9999).
- If *msgid_list* is specified, but the exception generated is not one specified in the list, the exception handler does not get control.

#pragma

The macro `_C1_ALL`, defined in the `<except.h>` header file, can be used as the equivalent of all the valid *class1* exception masks. The macro `_C2_ALL`, defined in the `<except.h>` header file, can be used as the equivalent of all four of the valid *class2* exception masks.

You can use the binary OR operator to monitor for different types of messages. For example,

```
#pragma exception_handler(myhandler, my_comarea, 0, _C2_MH_ESCAPE |\
                        _C2_MH_STATUS | _C2_MH_NOTIFY, _CTLA_IGNORE,
                        "MCH0000")
```

sets up an exception monitor for three of the four C2 exception classes that can be monitored.

The compiler issues an error message if any of the following occurs:

- The directive occurs outside a C++ function body or inside a C++ statement, with the exception of a control statement. You can place the directive right after the control statement or right after the expression in an iterative statement.
- The handler named is not a declared *function* or a defined *label*.
- The *com_area* variable has not been declared or does not have a valid object type.
- Either of the exception class masks is not a valid integral constant.
- The *ctl_action* is one of the disallowed values when the handler specified is a *label* (`_CTLA_INVOKE`, `_CTLA_IGNORE`, `_CTLA_IGNORE_NO_MSG`).
- The *msgid_list* is specified, but the *ctl_action* is not specified.
- A message in the *msgid_list* is not valid. Message prefixes that are not in uppercase are not considered valid.
- The messages in the string are not separated by a blank or comma.
- The string is not enclosed in “ ” or is longer than 4K bytes.

implementation

The **#pragma implementation** directive tells the compiler the name of the file containing the function-template definitions that correspond to the template declarations in the include file which contains the pragma.

```
▶▶—#—pragma—implementation—(—string_literal—)—▶▶
```

This pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

#pragma

info

The **#pragma info** directive controls the diagnostic messages generated by the

C++ Note: This directive is valid for C++ programs only.

Specifying **#pragma info(group)** causes all messages associated with that diagnostic group to be generated. Specifying **#pragma info(nogroup)** suppresses all messages associated with that group.

For example, to generate messages for missing function prototypes and statements with no effect, but not for uninitialized variables, specify:

```
#pragma info(pro, eff, nouni)
```

The `/wgroup` options are described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

langlvl

The **#pragma langlvl** directive selects the C or C++ language level for compilation.

```
▶▶ #pragma langlvl (ansi | extended | compat) ▶▶
```

This pragma can be specified only once in a source file, and it must appear before any statements in a source file. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level.

- | | |
|-----------------|---|
| ansi | Defines the predefined macros <code>__ANSI__</code> and <code>__STDC__</code> and undefines other langlvl variables. Allows only language constructs that conform to ANSI/ISO C standards. |
| extended | Defines the predefined macro <code>__EXTENDED__</code> and undefines other langlvl variables. The default language level is extended . |
| compat | Defines the predefined macro <code>__COMPAT__</code> and undefines other langlvl variables. This macro is not supported for C. It is provided for cfront compatibility. |

The **langlvl** compiler options `/Sa`, `/Se`, and `Sc` have the same effect as this pragma. These options are described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

#pragma

map

The **#pragma map** directive tells the compiler that all references to an identifier are to be converted to "name".

```
▶ #pragma map ( [ identifier ] , "name" ) ▶  
                [ func_or_op_identifier ( argument_list ) ]
```

<i>identifier</i>	A name of a data object or a nonoverloaded function with external linkage.
<i>func_or_op_identifier</i>	A name of a function or operator with internal linkage. The name can be qualified.
<i>argument_list</i>	A prototype list for the named function or operator.
<i>name</i>	The external name that is to be bound to the given object, function or operator.

The directive can appear anywhere within a single compilation unit. It can appear before any declaration or definition of the named object, function, or operator. The identifiers appearing in the directive, including any type names used in the prototype argument list, are resolved as though the directive had appeared at file scope, independent of its actual point of occurrence.

For example:

```
int func(int);  
  
class X  
{  
public:  
    void func(void);  
#pragma map(func, "funcname1") // maps ::func  
#pragma map(X::func, "funcname2") // maps X::func  
};
```

In C++, you should not use **#pragma map** to map the following:

- C++ Member functions
- Overloaded functions
- Objects generated from templates
- Functions with C++ or **builtin** linkage

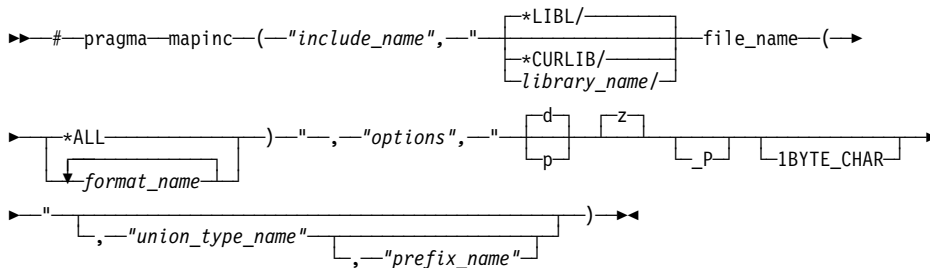
Such mappings override the compiler-generated names, which could cause binder errors.

mapinc

The **#pragma mapinc** directive indicates that AS/400 external file descriptions (DDS) are to be included in the C++ source. The directive identifies the file and DDS record formats, and provides information on the fields to be included. This pragma, along with

#pragma

an include directive, causes the compiler to automatically generate typedefs from the record formats specified in the file descriptions.



<i>include_name</i>	The name of the file that you reference on the <code>#include</code> directive in the source program. It is the name of the file in which the generated header information is stored. The <i>include_name</i> must be a valid workstation include file name. See "File Inclusion (<code>#include</code>)" on page 201 for information on how workstation include files are processed. Either a fully qualified path <code>"c:/abc/aaa.h"</code> or a relative path <code>"proj1/aaa.h"</code> can be specified.				
<i>library_name</i>	The name of the library that contains the externally described file. The <i>library_name</i> follows the AS/400 file naming conventions. See <i>VisualAge for C++ for AS/400 C++ Programming Guide</i> for a description of these conventions.				
<i>file_name</i>	The name of the externally described file. It follows the AS/400 file-naming conventions.				
<i>format_name</i>	A required parameter that indicates the DDS record format to be included in your program. You can specify more than one record format (<code>format1 format2</code>), or all the formats in a file (<code>*ALL</code>). It follows the AS/400 record format naming convention.				
<i>options</i>	The possible <i>options</i> are: <table border="0" style="margin-left: 20px;"> <tr> <td style="vertical-align: top;">input</td> <td>Fields declared as either INPUT or BOTH in the DDS are included in the typedef structure. Response indicators are included in the input structure when the keyword INDARA is not specified in the external file description (DDS source) for device files.</td> </tr> <tr> <td style="vertical-align: top;">output</td> <td>Fields declared as either OUTPUT or BOTH in DDS are included in the typedef structure. Option indicators are included in the output structure when the keyword INDARA is not</td> </tr> </table>	input	Fields declared as either INPUT or BOTH in the DDS are included in the typedef structure. Response indicators are included in the input structure when the keyword INDARA is not specified in the external file description (DDS source) for device files.	output	Fields declared as either OUTPUT or BOTH in DDS are included in the typedef structure. Option indicators are included in the output structure when the keyword INDARA is not
input	Fields declared as either INPUT or BOTH in the DDS are included in the typedef structure. Response indicators are included in the input structure when the keyword INDARA is not specified in the external file description (DDS source) for device files.				
output	Fields declared as either OUTPUT or BOTH in DDS are included in the typedef structure. Option indicators are included in the output structure when the keyword INDARA is not				

#pragma

	specified in the external file description (DDS source) for device files.
both	Fields declared as INPUT, OUTPUT or BOTH in DDS are included in the typedef structure. Option and response indicators are included in both structures when the keyword INDARA is not specified in the external file description (DDS source) for device files.
key	Fields declared as keys in the external file description are included and the keyword INDARA is specified in the DDS source. This option is only valid for database files and DDM files.
indicators	A separate 99-byte structure for indicators is created when the indicator option is specified. This option is only valid for device files.
lvlchk	A typedef of an array of struct is generated (type name <code>_LVLCHK_T</code>) for the level check information. A pointer to an object of type <code>_LVLCHK_T</code> is also generated and is initialized with the level check information (format name and level identifier).
nullflds	If there is at least one null-capable field in the record format of the DDS, a null map typedef is generated containing a character field for every field in the format. With this typedef, you can specify which fields are to be null (set value of each null field to '1', otherwise set to '0'). Also, if the key option is used along with the option <code>nullflds</code> , and there is at least one null-capable key field in the format, an additional typedef is generated containing a character field for every KEY field in the format. For physical and logical files, you can specify <code>input</code> , <code>both</code> , <code>key</code> , <code>lvlchk</code> , and <code>nullflds</code> . For device files you can specify <code>input</code> , <code>output</code> , <code>both</code> , <code>indicators</code> and <code>lvlchk</code> .
d	Binary coded decimal class (<code>_DecimalT</code> class template). This is the default.
p	Packed fields that are declared as character fields.
z	Zoned fields that are declared as character fields. This is the default because the zoned data type is not supported.
_P	Packed structure that is generated.
1BYTE_CHAR	A single byte character field for one-byte characters that is generated.

#pragma

" "	Default values of d and z that are used.
<i>union_type_name</i>	Union definition of the included type definitions that is created with the name <i>union_type_name.t</i> . This parameter is optional.
<i>prefix_name</i>	Replaces the library and file of the generated typedef structure name. If the <i>prefix_name</i> is not specified, the <i>prefix_name</i> is <i>library_file_prefix</i> . If a <i>prefix_name</i> is required but a <i>union_type_name</i> is not required, " " should be specified in the equivalent position for the <i>union_type_name</i> parameter.

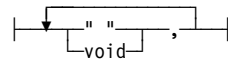
operational descriptor

The **#pragma descriptor** directive is used to identify functions whose arguments have operational descriptors.

An operational descriptor is an optional piece of information associated with a function argument. This information is used to describe an argument's attributes, for example, its data type and length.

▶▶ #pragma descriptor (—void—function_name—(—| od_specifiers |—)—)▶▶

od_specifiers:



Operational descriptors are useful when passing arguments to functions written in other languages that may have a different definition of the data types of the arguments. For example, C++ defines a string as a contiguous sequence of characters terminated by and including the first null character. In another language, a string may be defined as consisting of a length specifier and a character sequence. When passing a string from a C++ function to a function written in another language, an operational descriptor can be provided with the argument to allow the called function to determine the length and type of the string being passed.

The compiler generates operational descriptors for arguments passed to a function specified in a **#pragma descriptor** directive. The generated descriptor contains the descriptor type, data type, and length for each argument identified as requiring an operational descriptor. The information in an operational descriptor can be retrieved by the called function using the ILE APIs CEEGSI and CEED0D. See *System API Reference* for information about the ILE APIs for operational descriptors.

For the operational descriptor to determine the correct string length when passed through a function, the string has to be initialized. If the string is not initialized, the first element is auto-initialized to NULL '\0'; consequently, the operational descriptor generates a string length of 1.

The compiler supports operational descriptors for describing strings.

Note: A character string in C++ is defined by using any one of the following:

#pragma

• char string_name[n]	
• char * string_name	
• A string literal	
void	The return value of a function. No operational descriptor is used for the return value of a function.
<i>function_name</i>	The name of the function whose arguments require operational descriptors.
od_specifiers	A list of symbols, consisting of either " " or void, separated by commas, that specify which of a function's arguments are to have operational descriptors. An od_specifier list is similar to the argument list of a function, except that an od_specifier list for a function can have fewer specifiers than its argument list. If a string operational descriptor is required for an argument, then " " should be specified in the equivalent position for the od_specifier parameter. If an operational descriptor is not required for an argument, then void is specified for that parameter in the equivalent position for the od_specifier list.

The compiler issues a warning and ignores the **#pragma descriptor** directive if any of the following conditions occur:

- The function `function_name` is not prototyped before its **#pragma descriptor** directive.
- A call to the function `function_name` occurs before its **#pragma descriptor** directive.
- The function `function_name` is a static function or a user entry procedure, that is, `main()`.

When using operational descriptors, consider the following:

- Operational descriptors are only generated for functions that are called by their function name. Functions that are called by function pointers do not have operational descriptors generated.
- If there are fewer `od_specifiers` than function arguments, the remaining `od_specifiers` default to void.
- If a function requires a variable number of arguments, the **#pragma descriptor** directive can specify that operational descriptors are to be generated for the required arguments but not for the variable arguments.
- It is not valid to do pointer arithmetic on a literal or array while it is also used as an argument that requires an operational descriptor, unless explicitly cast to `char *`. For example, if `F` is a function that takes as an argument a string, and `F` requires an operational descriptor for this argument, then the argument on the following call to `F` is not valid: `F(a + 1)` where `"a"` is defined as `char a[10]`.

#pragma

pack

The **#pragma pack** directive specifies the alignment rules to use for the structures, unions, and classes that follow it. In C++, packing is performed on *declarations* or types. This is different from C, where packing is also performed on *definitions*.

You can also use the /Sp compiler option to cause packing to be performed along the boundary specified by /Sp. This option is described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

```
▶▶ #pragma pack (Nmember [Nclass, className]) ▶▶
```

<i>Nmember</i>	The alignment requirement for each member of the class.				
<i>Nclass</i>	The alignment and size requirements for the whole class. This parameter is optional.				
<i>className</i>	The name of the class to which the pragma applies. This parameter is optional. <i>className</i> can have the following values: <table><tr><td><i>className</i></td><td>A class name that has already been declared. The pragma applies only to the class <i>className</i>.</td></tr><tr><td>*</td><td>The pragma applies to all class definitions that follow the pragma. For example, <code>#pragma pack (2, 4, *)</code></td></tr></table>	<i>className</i>	A class name that has already been declared. The pragma applies only to the class <i>className</i> .	*	The pragma applies to all class definitions that follow the pragma. For example, <code>#pragma pack (2, 4, *)</code>
<i>className</i>	A class name that has already been declared. The pragma applies only to the class <i>className</i> .				
*	The pragma applies to all class definitions that follow the pragma. For example, <code>#pragma pack (2, 4, *)</code>				

Nmember and *Nclass* can have the following values:

{ 1, 2, 4, 8, 16 }	A number in the set of { 1, 2, 4, 8, 16 }. For example, <code>#pragma pack (2, 4, className)</code>
*	The alignment requirements currently active are to be applied. For example, <code>#pragma pack (*, 8, className)</code>
default	The alignment requirements specified on the command line are to be applied. For example, <code>#pragma pack (default, default, className)</code>
system	The system alignment requirements associated with each member's data type are to be applied.

Nclass and *className* are optional. This pragma can be used with either one or three arguments. If *Nclass* is omitted then *className* must also be omitted. The reverse is also true: if *className* is omitted, then *Nclass* must also be omitted. For example, `#pragma pack (4)`

#pragma pack can be applied in two different ways:

1. **#pragma pack** applies to every class definition that follows it.
2. **#pragma pack** is associated with a particular class.

#pragma

If **#pragma pack** is specified such that it applies to every class definition that follows it, the pragma is *stackable*.

The use of **#pragma pack (reset)** or of **pragma pack()** pops the stack by one and resets the alignment requirements to the state that was active before the previous **#pragma pack** was seen. For example,

```
// Default alignment requirements used
.  
.  
#pragma pack (4)  
class A { };  
#pragma pack (2, 2, *)  
class B { };  
class C { };  
#pragma pack (reset)  
class D { };  
#pragma pack ()  
class E { };
```

When class A is mapped, its members are aligned according to #pragma pack(4). When class B and class C are mapped, their members are aligned according to pragma pack(2, 2, *).

The #pragma. pack (reset) pops the alignment requirements specified by #pragma pack(2, 2, *) and resets the alignment requirements as specified by #pragma pack(4).

When class D is mapped, its members are aligned according to pragma pack(4). The #pragma pack () pops the alignment requirements specified by #pragma pack(4) and resets the alignment requirements to the default values used at the beginning of the file.

When class E is mapped, E's members are aligned as specified by the default alignment requirements (specified on the command line) active at the beginning of the file.

When **#pragma pack** is associated with a class name, it overwrites the alignment requirements specified by any previous **#pragma pack**. For example,

```
#pragma pack (4)  
class A {  
#pragma pack (2, 2, A)  
.  
.  
};
```

The alignment requirements used for class A are the ones specified by #pragma pack (2, 2, A), which overwrites pragma pack (4) specified earlier.

If a *className* is specified in **#pragma pack**, the *className* must have been already declared. For example,

#pragma

```
class F001;
#pragma pack (16, 16, F001)

class F002 {
    #pragma pack (16, 16, F002)
    :
    :
};
```

#pragma pack must occur before the end of the class definition. Since inline member functions are generated at the end of the class member list, (when } is found), the **#pragma pack** must be seen before the semantic analysis for the inline member functions take place and before code is generated for them. For example,

```
class F00 {
    :
    :
    #pragma pack (F00, 16, *)    // legal
};
```

In this example, **#pragma pack** applies to the nested class B of class A only.

```
class A {
    :
    class B {
        #pragma pack (2, 2, B)
        :
    };
    :
};
```

Only one **#pragma pack** per class is allowed or an error message is generated.

Notes:

1. More than one **#pragma pack** can be associated with a *className* if the same *Nmember* and *Nclass* value are used.
2. If two **#pragma pack** directives are seen for the same class, the use of *, default, or system in one pragma can be interchanged with the use of numbers in the other pragma (if the number specified happens to have the same value as one of the special keywords). Be aware that such a practice is not recommended. Since the value associated with * and default may change over time, programs that used to work may not compile anymore because the values specified by the different **#pragma pack** directives do not match.

Every data member is aligned as specified by the **#pragma pack** *Nmember* alignment. Only members with a natural alignment stricter than the one specified by the *Nmember* (member alignment > *Nmember*) are affected.

The purpose of *Nclass* is to cause additional padding to be added at the end of the class to ensure that the whole class aligns on a *Nclass* boundary. For example,

#pragma

```
class C {
#pragma pack (2, 8, C)
    char c1;
    short s;
    char c2;
    int i;
};
```

On a machine where **int** is 4-bytes long and is normally aligned on a 4 byte boundary, the alignment is as follows:

```
-----
|c1|* | s |c2|* | i | * |
-----
0 1 2 4 5 6 10 16
```

where '*' represents padding

Notice that only the alignment of the member *i* is affected by the *Nmember* value 2. The **char** and **short** members are aligned on their natural boundaries. Padding was also added at the end of the class to ensure that the class would align on an 8-byte boundary (as specified by the *Nclass* value).

If the class contains members with stricter alignment requirements than is specified by *Nclass*, the compiler does not readjust the class alignment and the *Nclass* alignment still applies. For example,

```
class C {
#pragma pack (4, 2, C)
    int i;
    char c;
};
```

On a machine where **int** is 4-bytes long and is normally aligned on a 4-byte boundary, the alignment is as follows:

```
-----
| i | c | * |
-----
0 4 5 6
```

where '*' represents padding

Note:

Even though *i* is the member with the strictest alignment requirement and even though *i* is aligned on a 4-byte boundary, the class size is rounded to a multiple of 2 bytes and the class has an alignment requirement of 2 bytes.

However, on some systems this may not apply because class members of certain data types may not be influenced by the **#pragma pack** directive.

#pragma

On some systems, class members of certain data types may not be influenced by the **#pragma pack** directive. For these members, the natural alignment requirements are always respected. On AS/400, pointers have a size and alignment of 16. For example,

```
typedef __Packed struct s1_tag {  
    char c1; /* offset 0 */  
    int i1; /* offset 1 */  
    void *p1; /* offset 16 not 5 */  
    char c2; /* offset 32 */  
} s1;  
  
sizeof(s1) == 48;
```

If a class has a base class with an alignment requirement stricter than any of its non-inherited members, the alignment requirement of the derived class is the same as the alignment requirement of the base class, and the size of the derived class is a multiple of the alignment of the base class.

If a class has a nested class with an alignment requirement stricter than any of its other members, the alignment requirement of the enclosing class is the same as the alignment requirement of the nested class, and the size of the enclosing class is a multiple of the alignment of the nested class. For example,

```
struct nested {  
    int in;  
    char cn;  
};  
struct enclose {  
    short se;  
    struct nested ns;  
};
```

On a machine where **int** is 4 bytes long and is normally aligned on a 4-byte boundary, the alignment is as follows:

```
-----  
| se | ** | in | cn | ** |  
-----  
0   2   4       8   9       12  
      |<----- ns ----->|
```

where '*' represents padding

The size and alignment of `enclose` is determined by the size and alignment requirements of `nested` (which must be aligned on a 4-byte boundary).

If a **#pragma pack** is applied to a derived class, this affects the base class alignment inside the derived class, but does not affect the mapping of the base class members (that is, their position from the beginning of the base class does not change).

#pragma

If a **#pragma pack** is applied to an enclosing class, this affects the nested class alignment inside the enclosing class, but does not affect the mapping of the nested class members (that is, their position from the beginning of the nested class does not change). For example,

```
struct nested {
    char cn;
    int in;
};
struct enclose {
#pragma pack (2, 2, enclose)
    short se;
    struct nested ns;
};
```

On a machine where **int** is 4 bytes long and is normally aligned on a 4-byte boundary, the alignment is as follows:

```
-----
| se | cn | ** | in |
-----
0   2   3   6   10
    |<----- ns ----->|
```

where '*' represents padding

ns is aligned inside **enclose** as required by the **#pragma pack**, on a 2-byte boundary. The mapping of **cn** and **in** from the beginning of the nested class **ns** is not changed by the **pragma** appearing in the **enclose** class.

#pragma pack can be used together with templates. If the *stackable* form of **#pragma pack** is used with templates, the **pragma** that is active when the class template is defined applies to all instantiations of the template. For example,

```
#pragma pack (1)
template<class T>
class F00 {
    :
    :
};

#pragma pack (4)

F00<int> fi;
```

The members of **F00<int>** are mapped as specified by **#pragma pack (1)**, which is active when the class template **F00** is defined.

The *named* form of **#pragma pack** can also be used with class templates. For example,

#pragma

```
template<class T>
class F00 {
    #pragma pack (4, system, F00<*>)
    #pragma pack (16, system, F00<int>)
    :
    :
};
```

The first **#pragma pack** `#pragma pack (4, system, F00<*>)` applies to all instantiations of the template `F00` (with any type used as the template argument).

The second **#pragma pack** `#pragma pack (16, system, F00<int>)` applies when the template class `F00` is instantiated with type `int`. As is the case for non-template classes, this pragma specifies that the members are aligned on a 16-byte boundary, and that the template class are aligned on the boundary that is most appropriate for its members.

This pragma applies whether the `F00<int>` definition is generated automatically or is explicitly declared in the code. If the template class contains more than one parameter, then each parameter must be specified with one of the following:

type name	A template type argument.
value	A template value argument.
*	Any template type or value argument. An * in the position of a type argument indicates for any type. An * in the position of a value argument indicates for any value.

A **#pragma pack** where all the template parameters are specified using *, (`F00<*,*>`) is a default pragma pack. For example,

```
template<class T, int i>
class F00 {
    #pragma pack (4, system, F00<*>) // ERROR
    #pragma pack (4, system, F00<*,*>)
    :
    :
};
```

Since the template has two parameters, the default pragma pack must use the syntax `F00<*,*>`.

#pragma pack can be used to specify the alignment requirements for a specific template instantiation or for a family of template instantiation. For example,

#pragma

```
template<class T, int i>
class F00 {
    #pragma pack (4, system, F00<*,*>)
    #pragma pack (16, system, F00<int,*>)
    #pragma pack (8, system, F00<*,4>)
    #pragma pack (16, system, F00<int,4>)
    :
    :
};
```

If the template class `F00<int,4>` is instantiated, the `#pragma pack (16, system, F00<int,4>)` is used. If the template class `F00<int,8>` is instantiated, the `#pragma pack (16, system, F00<int,*>)` is selected. The more precise match is always used.

If no **#pragma pack** matches the instantiation and no default **#pragma pack** is provided, the default alignment requirements apply to the template class instantiation. For example,

```
template<class T, int i>
class F00 {
    #pragma pack (4, system, F00<*,4>)
    :
    :
};
```

If the template class `F00<int,8>` is instantiated, no **#pragma pack** matches the instantiation and the default alignment requirements apply to the template class instantiation.

In case of ambiguity, the compiler generates an error message. For example,

```
template<class T, int i>
class F00 {
    #pragma pack (16, system, F00<int,*>)
    #pragma pack (16, system, F00<*,4>)
    :
    :
};
```

If template class `F00<int,4>` is instantiated, as both **#pragma pack** directives do not match the instantiated class exactly, and both pragmas can be used just as well with the type being instantiated, an ambiguous situation occurs, in which case a warning message is generated.

_Packed can only be associated with a class definition and has the same effect as `#pragma pack (1, 1, ClassName)`, where *ClassName* is the name of the class being defined. The following are examples of legal and illegal usages of **_Packed**.

#pragma

```
_Packed class SomeClass { /* ... */ }; // OK
typedef _Packed class AnotherClass {} PClass; // OK
typedef _Packed class {} PAnonClass; // Illegal, class must be named
_Packed SomeClass someObject; // Illegal, specifier _Packed must
// be associated with class
// definition.
_Packed struct SomeStruct { }; // OK
_Packed union SomeUnion { }; // OK
```

page

The **#pragma page** directive skips the number of pages specified by *pages* of the generated source listing. If *pages* is not specified, the next page is started.

```
▶▶ #pragma page ( pages ) ▶▶
```

pagesize

The **#pragma pagesize** directive sets the number of lines per page to *n* for the generated source listing.

#pragma pagesize is a C/MVS directive.

```
▶▶ #pragma pagesize ( n ) ▶▶
```

The value of *n* must be between 15 and 65535, inclusive. The default page length is 66 lines.

You can also use the /Lp compiler option to set the listing page size. This option is described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

pointer

The **#pragma pointer** directive allows the use of the AS/400 **pointer** types: space pointer, system pointer, invocation pointer, label pointer, suspend pointer, and open pointer. A variable declared with a **typedef** named in the **#pragma pointer** directive has the pointer type associated with `typedef_name` in the directive.

```
▶▶ #pragma pointer ( typedef_name , pointer_type ) ▶▶
```

The `<pointer.h>` header file provided by the compiler contains typedefs and pragma directives for these **pointer** types. Including this header file in C++ source code allows you to use these typedefs directly for declaring pointer variables of these types.

The `pointer_type` can be one of:

#pragma

SPCPTR	Space pointer
OPENPTR	Open pointer
SYSPTR	System pointer
INVPTR	Invocation pointer
LBLPTR	Label code pointer
SUSPENDPTR	Suspend pointer

The compiler issues a warning and ignores the **#pragma pointer** directive if any of the following errors occur:

- The **pointer** type named in the directive is not one of SPCPTR, SYSPTR, INVPTR, LBLPTR, SUSPENDPTR or OPENPTR.
- The **typedef** named is not declared before the **#pragma pointer** directive.
- The identifier named as the first parameter of the directive is not a **typedef**.
- The **typedef** named is not a **typedef** of a void pointer.
- The **typedef** named is used in a declaration before the **#pragma pointer** directive.

The **typedef** named must be defined at file scope.

priority

The **#pragma priority** directive specifies the order in which static objects are to be initialized at run time.

```
▶▶ #pragma priority (—n—) ▶▶
```

Where n is an integer literal in the range of **INT_MIN** to **INT_MAX**. The default value is 0. A negative value indicates a higher priority; a positive value indicates a lower priority.

The first 1024 priorities (**INT_MIN** to **INT_MIN** + 1023) are reserved for use by the compiler and its libraries. The **#pragma priority** can appear anywhere in the source file many times. However, the priority of each pragma must be greater than the previous pragma's priority. This is necessary to ensure that the runtime static initialization occurs in the declaration order. For example,

#pragma

```
//File one called First.C

#pragma priority (1000)
class A { public: int a; A() {return;} } a;
#pragma priority (3000)
class C { public: int c; C() {return;} } c;
class B { public: int b; B() {return;} };
extern B b;
main()
{
    a.a=0;
    b.b=0;
    c.c=0;
}

//File two called Second.C
#pragma priority (2000)
class B { public: int b; B() {return;} } b;
```

In this example, the execution sequence of the runtime static initialization is:

1. Static initialization with priority 1000 from file First.C
2. Static initialization with priority 2000 from file Second.C
3. Static initialization with priority 3000 from file First.C

skip

The **#pragma skip** directive skips the specified number of lines of the generated source listing. The value of *lines* must be a positive integer less than 255. If *lines* is omitted, one line is skipped.

Note: This directive is valid for C programs only.

```
▶▶ #pragma skip ( lines ) ▶▶
```

strings

The **#pragma strings** directive sets the storage type for strings. It specifies that the compiler can place strings into read-only memory or must place strings into read/write memory.

```
▶▶ #pragma strings ( writable / readonly ) ▶▶
```

This pragma must appear before any C or C++ code in a file.

C strings are read/write by default. C++ strings are readonly by default.

#pragma

subtitle

The **#pragma subtitle** directive places the text specified by *subtitle* on all subsequent pages of the generated source listing.

Note: This directive is valid for C programs only.

```
▶—#—pragma—subtitle—(—"subtitle"—)→◀
```

The string *subtitle* must be less than 255 characters.

You can also use the /Lu compiler option to specify the listing subfile. The /Lu option is described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

title

The **#pragma title** directive places the text specified by *title* on all subsequent pages of the generated source listing.

Note: This directive is valid for C programs only.

```
▶—#—pragma—title—(—"title"—)→◀
```

The string *title* must be less than 255 characters.

You can also use the /Lt compiler option to specify the listing title. The /Lt option is described in the *IBM VisualAge for C++ for AS/400 C++ User's Guide*.

Examples of #pragma Directives

```
#pragma pagesize(55)  
#pragma map(ABC, "A$BC@")
```

Chapter 9. Classes

This chapter discusses the following topics:

C++ Classes Overview	239
Declaring Class Objects	241
Scope of Class Names	246

Related Information

- "C++ Support for Object-Oriented Programming" on page 2
- Chapter 10, "Class Members and Friends" on page 253
- Chapter 13, "Inheritance" on page 315

C++ Classes Overview

A C++ *class* is a mechanism for creating user-defined data types. It is similar to the C-language structure data type. In C, a structure is composed of a set of data members. In C++, a class type is like a C structure, except that a class is composed of a set of data members and an optional set of operations that can be performed on the class.

In C++, a class type can be declared with the keywords **union**, **struct**, or **class**. A union object can hold any one of a set of named members. Structure and class objects hold a complete set of members. Each class type represents a unique set of class members including data members, member functions, and other type names. The default access for members depends on the class key:

- The members of a class declared with the class key `class` are private by default. A class is inherited privately by default.
- The members of a class declared with the class key `struct` are public by default. A structure is inherited publicly by default.
- The members of a union (declared with the class key `union`) are public by default. A union cannot be used as a base class in derivation. Base classes and derivation are described in Chapter 13, "Inheritance" on page 315.

Once you create a class type, you can declare one or more objects of that class type.

For example:

```
class X
{ /* define class members here */ };
void main()
{
    X xobject1;        // create an object of class type X
    X xobject2;        // create another object of class type X
}
```

C++ Classes Overview

Classes and Polymorphic Functions

Classes are also used in C++ to support polymorphic functions through overloaded functions (static compile time binding) and virtual functions (dynamic binding). C++ allows you to redefine standard operators and functions through the concept of overloading. Operator overloading facilitates data abstraction by allowing you to use classes as easily as built-in types.

C++ overloading is described in Chapter 11, “Overloading” on page 277, and virtual functions are described in “Virtual Functions” on page 338.

Classes and Structures

The C++ class is an extension of the C-language structure. Because the only difference between a structure and a class is that structure members have public access by default and a class members have private access by default, you can use the keywords **class** or **struct** to define equivalent classes.

For example, in the following code fragment, the class X is equivalent to the structure Y:

```
// In this example, class X is equivalent to struct Y

class X
{
int a; // private by default
public:
    int f() { return a = 5; }; // public member function
};
struct Y
{
int f() { return a = 5; }; // public by default
private:
    int a; // private data member
};
```

Declaring Class Objects

If you define a structure and then declare an object of that structure using the keyword **class**, the members of the object are still public by default. In the following example, `main()` has access to the members of `X` even though `X` is declared as using the keyword **class**:

```
// This example declares a structure, then declares a class
// that is an object of the structure.

#include <iostream.h>

struct x {
    int a;
    int b;
};

class x X;

void main() {
    X.a = 0;
    X.b = 1;
    cout << "Here are a and b " << X.a << " " << X.b << endl;
}
```

Aggregate Classes

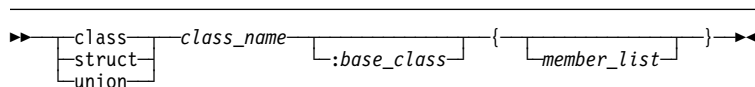
An *aggregate class* is a class that has no user-defined constructors, no private or protected members, no base classes, and no virtual functions.

Initialization of aggregate classes is described in "Initializers" on page 88.

Declaring Class Objects

A class declaration creates a unique type class name.

A *class specifier* is a type specifier used to declare a class. Once a class specifier has been seen and its members declared, a class is considered to be defined even if the member functions of that class are not yet defined. A class specifier has the following form:



The *member_list* is optional. It specifies the class members, both data and functions, of the class *class_name*. If the *member_list* of a class is empty, objects of that class have a nonzero size. You can use a *class_name* within the *member_list* of the class specifier itself as long as the size of the class is not required. For more information, see "Class Member Lists" on page 253.

Declaring Class Objects

The *base_class* is optional. It specifies the base class or classes from which the class *class_name* inherits members. If the *base_class* is not empty, the class *class_name* is called a *derived class*. See “Derivation” on page 319 for more information about derived classes.

The declarator for a class variable declared with the **class**, **struct**, or **union** keyword is an identifier. If the symbol * precedes the identifier, the identifier names a pointer to a class of the specified data type. If ** precedes the identifier, the identifier names a pointer to a pointer to a class of the specified data type.

If a constant expression enclosed in [] (brackets) follows the identifier, the identifier names an array of classes of the specified data type. If * precedes the identifier and a constant expression enclosed in [] follows the identifier, the identifier names an array of pointers to classes of the specified data type.

Class Names

A *class name* is a unique identifier that becomes a reserved word within its scope. Once a class name is declared, it hides other declarations of the same name within the enclosing scope.

If a class name is declared in the same scope as a function, enumerator, or object with the same name, that class can be referred to by using an *elaborated type specifier*. In the following example, the elaborated type specifier is used to refer to the class `print` that is hidden by the later definition of the function `print()`:

```
class print
{
    /* definition of class print */
};
void print (class print*);      // redefine print as a function
//      .                      // prefix class-name by class-key
//      .                      // to refer to class print
//      .
void main ()
{
    class print* paper;        // prefix class-name by class-key
                                // to refer to class print
    print(paper);              // call function print
}
```

You can use an elaborated type specifier with a class name to declare a class.

For more information on elaborated type specifiers, see “Incomplete Class Declarations” on page 247.

You can also qualify type names to refer to hidden type names in the current scope. You can reduce complex class name syntax by using a **typedef** to represent a nested class name.

Declaring Class Objects

In the following example, a **typedef** is used so that the simple name `nested` can be used in place of `outside::middle::inside`.

```
// This example illustrates a typedef used to simplify
// a nested class name.

#include <iostream.h>

class outside {
public:
    class middle {
    public:
        class inside {
        private:
            int a;
        public:
            inside(int a_init = 0): a(a_init) {}
            void printa();
        };
    };
};

typedef outside::middle::inside nested;

void nested::printa() {
    cout << "Here is a " << this->a << endl;
}

void main() {
    nested n(9);
    n.printa();
}
```

For more information on nested classes, see “Nested Classes” on page 248

Declaring Class Objects

Using Class Objects

You can use a class type to create instances or *objects* of that class type. For example, you can declare a class, structure, and union with class names X, Y, and Z respectively:

```
class X {      /* definition of class X */ };
struct Y {    /* definition of struct Y */ };
union Z {     /* definition of union Z */ };
```

You can then declare objects of each of these class types. Remember that classes, structures, and unions are all types of C++ classes.

```
void main()
{
    X xobj;      // declare a class object of class type X
    Y yobj;      // declare a struct object of class type Y
    Z zobj;      // declare a union object of class type Z
}
```

In C++, unlike C, you do not need to precede declarations of class objects with the keywords **union**, **struct**, and **class** unless the name of the class is hidden. For example:

```
struct Y { /* ... */ };
class X { /* ... */ };
void main ()
{
    int X;          // hides the class name X
    Y yobj;         // valid
    X xobj;         // error, class name X is hidden
    class X xobj;  // valid
}
```

For more information on hidden names, see “Scope of Class Names” on page 246.

When you declare more than one class object in a declaration, the declarators are treated as if declared individually. For example, if you declare two objects of class S in a single declaration:

```
class S { /* ... */ };
//      .
//      .
//      .
void main()
{
    S S,T; // declare two objects of class type S
}
```

Declaring Class Objects

this declaration is equivalent to:

```
class S { /* ... */ };
void main()
{
    S S;
    class S T;    // keyword class is required
                 // since variable S hides class type S
}

```

but is not equivalent to:

```
class S { /* ... */ };
//      .
//      .
//      .
void main()
{
    S S;
    S T;    // error, S class type is hidden
}

```

You can also declare references to classes, pointers to classes, and arrays of classes.

For example:

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
void main()
{
    X xobj;
    X &xref = xobj;    // reference to class object of type X
    Y *yptr;         // pointer to struct object of type Y
    Z zarray[10];    // array of 10 union objects of type Z
}

```

Objects of class types that are not copy restricted can be assigned, passed as arguments to functions, and returned by functions. For more information, see “Copy Restrictions” on page 312.

For more information on objects, see “Objects” on page 30. Initialization of classes is discussed in “Initialization by Constructor” on page 306.

Scope of Class Names

Scope of Class Names

A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden. If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared, you can only refer to the class by using the elaborated type specifier. The class key (**class**, **struct**, or **union**) must precede the class name to identify it.

For example:

```
// This example shows the scope of class names.

class x { int a; };           // declare a class type class-name

x xobject;                   // declare object of class type x

int x(class x*)              // redefine x to be a function
{return 0;}                  // use class-key class to define
                             // a pointer to the class type x
                             // as the function argument

void main()
{
    class x* xptr;           // use class-key class to define
                             // a pointer to class type x
    xptr = &xobject;        // assign pointer
    x(xptr);                // call function x with pointer to class x
}
```

An elaborated type specifier can be used in the declaration of objects and functions. See “Class Names” on page 242 for an example.

An elaborated type specifier can also be used in the incomplete declaration of a class type to reserve the name for a class type within the current scope.

Scope of Class Names

Incomplete Class Declarations

An *incomplete class declaration* is a class declaration that does not define any class members. You cannot declare any objects of the class type or refer to the members of a class until the declaration is complete. However, an incomplete declaration allows you to make specific references to a class prior to its definition as long as the size of the class is not required.

For example, you can define a pointer to the structure `first` in the definition of the structure `second`. Structure `first` is declared in an incomplete class declaration prior to the definition of `second`, and the definition of `oneptr` in structure `second` does not require the size of `first`:

```
struct first;           // incomplete declaration of struct first

struct second          // complete declaration of struct second
{
    first* oneptr;     // pointer to struct first refers to
                      // struct first prior to its complete
                      // declaration

    first one;        // error, you cannot declare an object of
                      // an incompletely declared class type

    int x, y;
};

struct first          // complete declaration of struct first
{
    second two;       // define an object of class type second
    int z;
};
```

If you declare a class with an empty member list, it is a complete class declaration. For example:

```
class X;               // incomplete class declaration
class Z {};           // empty member list
class Y
{
public:
    X yobj;           // error, cannot create an object of an
                      // incomplete class type
    Z zobj;           // valid
};
```

Class member lists are described on page 253.

Scope of Class Names

Nested Classes

A *nested class* is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class.

You can define member functions and static data members of a nested class in the global scope. For example, in the following code fragment, you can access the static members `x` and `y` and member functions `f()` and `g()` of the nested class `nested` by using a qualified type name. Qualified type names allow you to define a **typedef** to represent a qualified class name. You can then use the **typedef** with the `::` (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
class outside
{
public:
    class nested
    {
    public:
        static int x;
        static int y;
        int f();
        int g();
    };
};

int outside::nested::x = 5;
int outside::nested::f() { return 0; };

typedef outside::nested outnest;    // define a typedef
int outnest::y = 10;                // use typedef with ::
int outnest::g() { return 0; };
// . . .
```

Scope of Class Names

Local Classes

A *local class* is declared within a function definition. The local class is in the scope of the enclosing function scope. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                // global variable
void f()              // function definition
{
    static int y;     // static variable y can be used by
                    // local class
    int x;            // auto variable x cannot be used by
                    // local class
    extern int g();   // extern function g can be used by
                    // local class

    class local      // local class
    {
        int g() { return x; } // error, local variable x
                            // cannot be used by g
        int h() { return y; } // valid, static variable y
        int k() { return ::x; } // valid, global x
        int l() { return g(); } // valid, extern function g
    };
}

void main()
{
    local* z;         // error, local is undefined
    // .
    // .
    // .
}
```

Member functions of a local class have to be defined within their class definition. Member functions of a local class must be inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword **inline**.

For more information about inline functions, see “Inline Member Functions” on page 257.

Scope of Class Names

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
    class local
    {
        int f();           // error, local class has noninline
                          // member function
        int g() {return 0;} // valid, inline member function
        static int a;      // error, static is not allowed for
                          // local class
        int b;            // valid, nonstatic variable
    };
}
// . . .
```

An enclosing function has no special access to members of the local class.

Local Type Names

Local type names follow the same scope rules as other names. Scope rules are described in “Scope” on page 7. Type names defined within a class declaration have class scope and cannot be used outside their class without qualification.

If you use a class name, **typedef** name, or a constant name that is used in a type name, in a class declaration, you cannot redefine that name after it is used in the class declaration.

For example:

```
void main ()
{
    typedef double db;
    struct st
    {
        db x;
        typedef int db; // error
        db y;
    };
}
```

Scope of Class Names

The following declarations are valid:

```
typedef float T;
class s {
    typedef int T;
    void f(const T);
};
```

Here, function `f()` takes an argument of type `s::T`. However, the following declarations, where the order of the members of `s` has been reversed, cause an error:

```
typedef float T;
class s {
    void f(const T);
    typedef int T;
};
```

In a class declaration, you cannot redefine a name that is not a class name, or a **typedef** name to a class name or **typedef** name once you have used that name in the class declaration.

Scope of Class Names

Chapter 10. Class Members and Friends

This chapter describes class members and friends, including the following topics:

Class Member Lists	253
Data Members	254
Class-Type Class Members	255
Member Functions	256
Member Scope	258
Pointers to Members	260
The this Pointer	262
Static Members	265
Member Access	270
Friends	273

Related Information

- Chapter 9, “Classes” on page 239
- Chapter 13, “Inheritance” on page 315
- Chapter 12, “Special Member Functions” on page 293

Class Member Lists

An optional *member list* declares sub-objects called *class members*. Class members can be data, functions, classes, enumeration, bit fields, and typedef names. A member list is the only place you can declare class members. Friend declarations are not class members but must appear in member lists.

The member list follows the class name and is placed between braces. It can contain access specifiers, member declarations, and member definitions.

You can access members by using the class access `.` (dot) and `->` (arrow) operators. The class access operators are described in “Dot Operator `.`” on page 102 and “Arrow Operator `->`” on page 103.

A *member declaration* declares a class member for the class containing the declaration. For more information on declarations, see Chapter 3, “Declarations” on page 29, and “Declaring Class Objects” on page 241.

An *access specifier* is one of **public**, **private**, or **protected**. Access specifiers are described on page 270.

A member declaration that is a qualified name followed by a `;` (semicolon) is used to restore access to members of base classes and is described in “Access Declarations” on page 326.

A *member declarator* declares an object, function, or type within a declaration. It cannot contain an initializer. You can initialize a member by using a constructor or, if

Data Members

the member belongs to an aggregate class, by using a brace initializer list (a list surrounded by braces { }) in the declarator list. You must explicitly initialize a class containing constant or reference members with a brace initializer list or explicitly with a constructor.

A member declarator of the form:

[identifier] : constant-expression

specifies a bit field.

A *pure specifier* (= 0) indicates that a function has no definition. It is only used with virtual member functions and replaces the function definition of a member function in the member list. Pure specifiers are described in “Virtual Functions” on page 338.

You can use the storage-class specifier **static** (but not **extern**, **auto** or **register**) in a member list. For more information, see “Static Members” on page 265.

The order of mapping of class members in a member list is implementation dependent. For the VisualAge for C++ for AS/400 compiler, class members are allocated in the order they are declared.

Data Members

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition.

If an array is declared as a nonstatic class member, you must specify all of the dimensions of the array.

Class-Type Class Members

A class can have members that are of a class type or are pointers or references to a class type. Members that are of a class type must be of a class type that is previously declared. An incomplete class type can be used in a member declaration as long as the size of the class is not needed. For example, a member can be declared that is a pointer to an incomplete class type. For more information, see “Incomplete Class Declarations” on page 247.

A class *X* cannot have a member that is of type *X*, but it can contain pointers to *X*, references to *X*, and static objects of *X*. Member functions of *X* can take arguments of type *X* and have a return type of *X*. For example:

```
class X
{
    X();
    X *xptr;
    X &xref;
    static X xcount;
    X xfunc(X);
};
```

The bodies of member functions are always processed *after* the definition of their class is complete. For this reason, the body of a member function can refer to the name of the class that owns it, even if this requires information about the class definition.

The language allows member functions to refer to any class member even if the member function definition appears before the declaration of that member in the class member list. For example,

```
class Y
{
public:
    int a;
    Y ();
private:
    int f() {return sizeof(Y);};
    void g(Y yobj);
    Y h(int a);
};
```

In this example, it is permitted for the inline function `f()` to make use of the size of class *Y*. Inline member functions are described on page 257.

Member Functions

Member Functions

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the **friend** specifier. These are called *friends* of a class. For more information, see “Friends” on page 273.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body. When the function `add()` is called in the following example, the data variables `a`, `b`, and `c` can be used in the body of `add()`.

```
class x
{
public:
    int add()           // inline member function add
    {return a+b+c;};
private:
    int a,b,c;
};
```

For information on static member functions, see “Static Member Functions” on page 269. For more general information on functions, see Chapter 6, “Functions.”

const and volatile Member Functions

A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

Virtual Member Functions

Virtual member functions are declared with the keyword **virtual**. They allow dynamic binding of member functions. Because all virtual functions must be member functions, virtual member functions are simply called virtual functions.

If the definition of a virtual function is replaced by a pure specifier in the declaration of the function, the function is said to be declared pure. A class that has at least one pure virtual function is called an abstract class.

Virtual Functions are described in more detail on page 338. Pure virtual functions are described in “Abstract Classes” on page 343.

Special Member Functions

Special member functions are used to create, destroy, initialize, convert, and copy class objects. These include:

- Constructors
- Destructors
- Conversion constructors

Member Functions

- Conversion functions
- Copy constructors

Special member functions are described in Chapter 12, “Special Member Functions.”

Inline Member Functions

A member function that is both declared and defined in the class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline.

An equivalent way to declare an inline member function is to declare it outside of the class declaration using the keyword **inline** and the `::` (scope resolution) operator to identify the class the member function belongs to. For example:

```
class Y
{
    char* a;
public:
    char* f() {return a;};
};
```

is equivalent to:

```
class Z
{
    char* a;
public:
    char* f();
};
// .
// .
// .
inline char* Z::f() {return a;}
```

When you declare an inline function without the **inline** keyword and do not define it in the class member list, you cannot call the function before you define it. In the above example, you cannot call `f()` until after its definition.

Inline member functions have internal linkage. Noninline member functions have external linkage.

For more information, see “Inline Functions” on page 163.

Member Function Templates

Any member function (inlined or noninlined) declared within a class template is implicitly a function template. When a template class is declared, it implicitly generates template functions for each function defined in the class template. If a class template is instantiated, only the function templates whose instantiations will actually be used by the resulting template class are instantiated.

Member Scope

For more information about member function templates, see “Member Function Templates” on page 361.

Member Scope

Member functions and static members can be defined outside their class declaration if they have already been declared, but not defined, in the class member list. Nonstatic data members are defined when their class is instantiated. The declaration of a static data member is not a definition. The declaration of a member function is a definition if the body of the function is also given.

Whenever the definition of a class member appears outside of the class declaration, the member name must be qualified by the class name using the `::` (scope resolution) operator.

Member Scope

The following example defines a member function outside of its class declaration.

// This example illustrates member scope.

```
#include <iostream.h>
class X
{
public:
    int a, b ;      // public data members
    int add();     // member function declaration only
};
int a = 10;       // global variable
// define member function outside its class declaration
int X::add() {return a + b;};
//      .
//      .
//      .
void main()
{
    int answer;
    X xobject;
    xobject.a = 1;
    xobject.b = 2;
    answer = xobject.add();
    cout << xobject.a << " + " << xobject.b << " = " << answer<<endl;
}
```

The output for this example is: 1 + 2 = 3

All member functions are in class scope even if they are defined outside their class declaration. In the above example, the member function `add()` returns the data member `a`, not the global variable `a`.

The name of a class member is local to its class. Unless you use one of the class access operators, `.` (dot), or `->` (arrow), or `::` (scope resolution) operator, you can only use a class member in a member function of its class and in nested classes. You can only use types, enumerations and static members in a nested class without qualification with the `::` operator.

The order of search for a name in a member function body is:

1. Within the member function body itself
2. Within all the enclosing classes, including inherited members of those classes
3. Within the lexical scope of the body declaration

Pointers to Members

The search of the enclosing classes, including inherited members, is demonstrated in the following example:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class Z : A {
    class Y : B {
        class X : C { int f(); /* ... */ };
    };
};
int Z::Y::X f()
{
    // .
    // .
    // .
    j();
    // .
    // .
    // .
}
```

In this example, the search for the name `j` in the definition of the function `f` follows this order:

1. In the body of the function `f`
2. In `X` and in its base class `C`
3. In `Y` and in its base class `B`
4. In `Z` and in its base class `A`
5. In the lexical scope of the body of `f`. In this case, this is global scope.

Note: When the containing classes are being searched, only the definitions of the containing classes and their base classes are searched. The scope containing the base class definitions (global scope, in this example) is not searched.

Pointers to Members

Pointers to members allow you to refer to nonstatic members of class objects. You cannot use a pointer to member to point to a static class member because the address of a static member is not associated with any particular object. To point to a static class member, you must use a normal pointer.

You can use pointers to member functions in the same manner as pointers to functions. You can compare pointers to member functions, assign values to them, and use them to call member functions. Note that a member function does not have the same type as a nonmember function that has the same number and type of arguments and the same return type.

Pointers to Members

Pointers to members can be declared and used as shown in the following example:

// This example illustrates pointers to members.

```
#include <iostream.h>
class X
{
public:
    int a;
    void f(int b)
    {
        cout << "The value of b is "<< b << endl;
    }
};
// .
// .
// .
void main ()
{
    // declare pointer to data member
    int X::*ptiptr = &X::a;

    // declare a pointer to member function
    void (X::* ptfptr) (int) = &X::f;

    X xobject;           // create an object of class type X
    xobject.*ptiptr = 10; // initialize data member

    cout << "The value of a is " << xobject.*ptiptr << endl;
    (xobject.*ptfptr) (20); // call member function
}
```

The output for this example is:

```
The value of a is 10
The value of b is 20
```


The this Pointer

To reduce complex syntax, you can declare a **typedef** to be a pointer to a member. A pointer to a member can be declared and used as shown in the following code fragment:

```
typedef void (X::*ptfptr) (int);    // declare typedef
void main ()
{
    // .
    // .
    // .
    ptfptr ptf = &X::f;            // use typedef
    X xobject;
    (xobject.*ptf) (20);          // call function
}
```

The pointer to member operators `.*` and `->*` are used to bind a pointer to a member of a specific class object. Because the precedence of `()` (function call operator) is higher than `.*` and `->*`, you must use parentheses to call the function pointed to by `ptf`.

For more information, see “C++ Pointer to Member Operators `.* ->*`” on page 123.

The this Pointer

The keyword **this** identifies a special type of pointer. When a nonstatic member function is called, the **this** pointer identifies the class object which the member function is operating on. You cannot declare the **this** pointer or make assignments to it.

The type of the **this** pointer for a member function of a class type `X`, is `X* const`. If the member function is declared with the constant qualifier, the type of the **this** pointer for that member function for class `X`, is `const X* const`. If the member function is declared with the volatile qualifier, the type of the **this** pointer for that member function for class `X` is `volatile X* const`.

this is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

For example, you can refer to the particular class object that a member function is called for by using the **this** pointer in the body of the member function. The following code example produces the output `a = 5`:

The this Pointer

```
// This example illustrates the this pointer

#include <iostream.h>
class X
{
    int a;
public:
    // The 'this' pointer is used to retrieve 'xobj.a' hidden by
    // the automatic variable 'a'
    void Set_a(int a) { this->a = a; }
    void Print_a() { cout << "a = " << a << endl; }
};
void main()
{
    X xobj;
    int a = 5;
    xobj.Set_a(a);
    xobj.Print_a();
}
```

Unless a class member name is hidden, using the class member name is equivalent to using the class member name qualified with the **this** pointer.

The following example shows code using class members without the **this** pointer. The comments on each line show the equivalent code with the hidden use of the **this** pointer.

```
// This example uses class members without the this pointer.

#include <string.h>
#include <iostream.h>
class X
{
    int len;
    char *ptr;
public:
    int GetLen()           // int GetLen (X* const this)
    { return len; }       // { return this->len; }
    char * GetPtr()       // char * GetPtr (X* const this)
    { return ptr; }       // { return this->ptr; }
    X& Set(char *);
    X& Cat(char *);
    X& Copy(X&);
    void Print();
};
```

The this Pointer

```
X& X::Set(char *pc)           // X& X::Set(X* const this, char *pc)
{
    len = strlen(pc);        // this->len = strlen(pc);
    ptr = new char[len];     // this->ptr =
                             // new char[this->len];
    strcpy(ptr, pc);         // strcpy(this->ptr, pc);
    return *this;
}

X& X::Cat(char *pc)          // X& X::Cat(X* const this, char *pc)
{
    len += strlen(pc);      // this->len += strlen(pc);
    strcat(ptr,pc);         // strcat(this->ptr,pc);
    return *this;
}

X& X::Copy(X& x)             // X& X::Copy(X* const this, X& x)
{
    Set(x.GetPtr());        // this->Set(x.GetPtr(&x));
    return *this;
}

void X::Print()              // void X::Print(X* const this)
{
    cout << ptr << endl;    // cout << this->ptr << endl;
}

void main()
{
    X xobj1;
    xobj1.Set("abcd").Cat("efgh");
    // xobj1.Set(&xobj1, "abcd").Cat(&xobj1, "efgh");

    xobj1.Print();          // xobj1.Print(&xobj1);
    X xobj2;
    xobj2.Copy(xobj1).Cat("ijkl");
    // xobj2.Copy(&xobj2, xobj1).Cat(&xobj2, "ijkl");

    xobj2.Print();         // xobj2.Print(&xobj2);
}
```

This example produces the following output:

```
abcdefgh
abcdefghijkl
```

Static Members

Class members can be declared using the storage-class specifier **static** in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

The declaration of a static member in the member list of a class is not a definition. The definition of a static member is equivalent to an external variable definition. You must define the static member outside of the class declaration.

For example:

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
//      .
//      .
//      .
```

A static member can be accessed from outside of its class only if it is declared with the keyword **public**. You can then access the static member by qualifying the class name using the `::` (scope resolution) operator. In the following example:

```
class X
{
public:
    static int f();
};
//      .
//      .
//      .
void main ()
{
    X::f();
}
```

you can refer to the static member `f()` of class type `X` as `X::f()`.

For more information on the storage-class specifier **static**, see “static Storage Class Specifier” on page 40

Static Members

Using the Class Access Operators with Static Members

You can also access a static member from a class object by using the class access operators `.` (dot) and `->` (arrow). `x*`

The following example uses the class access operators to access static members.

```
// This example illustrates access to static
// members with class access operators.

#include <iostream.h>
class X
{
    static int cnt;
public:
    // The following routines all set X's static variable cnt
    // and print its value.
    void Set_Show (int i)
    {
        X::cnt = i;
        cout << "X::cnt = " << X::cnt << endl; }
    void Set_Show (int i, int j )
    {
        this->cnt = i+j;
        cout << "X::cnt = " << X::cnt << endl; }
    void Set_Show (X& x, int i)
    {
        x.cnt = i;
        cout << "X::cnt = " << X::cnt << endl; }
};
int X::cnt;
void main()
{
    X xobj1, xobj2;
    xobj1.Set_Show(11);
    xobj1.Set_Show(11,22);
    xobj1.Set_Show(xobj2, 44);
}
```

The above example produces the following output:

```
X::cnt = 11
X::cnt = 33
X::cnt = 44
```

When a static member is accessed through a class access operator, the expression on the left of the `.` or `->` operator is not evaluated.

A static member can be referred to independently of any association with a class object because there is only one static member shared by all objects of a class. A static member can exist even if no objects of its class have been declared.

Static Members

When you access a static member, the expression that you use to access it is not evaluated. In the following example, the external function `f()` returns class type `X`. The function `f()` can be used to access the static member `i` of class `X`. The function `f()` itself is not called.

```
// This example shows that the expression used to
// access a static member is not evaluated.

class X
{
public:
    static int i;
};
int X::i = 10;
X f() { /* ... */ }
void main ()
{
    int a;
    a = f().i;      // f().i does not call f()
}
```

Static Data Members

Static data members of global classes have external linkage and can be initialized in file scope like other global objects. Static data members follow the usual class access rules, except that they can be initialized in file scope. Static data members and their initializers can access other static private and protected members of their class. The initializer for a static data member is in the scope of the class declaring the member.

The following example shows how you can initialize static members using other static members, even though these members are private:

```
class C {
    static int i;
    static int j;
    static int k;
    static int l;
    static int m;
    static int n;
    static int p;
    static int q;
    static int r;
    static int s;
    static int f() { return 0; }
    int a;
public:
    C() { a = 0; }
};
```

Static Members

```
C c;
int C::i = C::f();    // initialize with static member function
int C::j = C::i;     // initialize with another static data member
int C::k = c.f();    // initialize with member function from an object
int C::l = c.j;      // initialize with data member from an object
int C::s = c.a;      // initialize with nonstatic data member
int C::r = 1;        // initialize with a constant value

class Y : private C { } y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;      // error
int C::q = y.f();    // error
```

The initializations of `C::p` and `C::x` cause errors because `y` is an object of a class that is derived privately from `C`, and its members are not accessible to members of `C`.

You can only have one definition of a static member in a program. If a static data member is not initialized, it is assigned a zero default value.

Local classes cannot have static data members.

The following example shows the declaration, initialization, use, and scope of the static data member `si` and static member functions `Set_si(int)` and `Print_si()`.

```
// This example shows the declaration, initialization,
// use, and scope of a static data member.

#include <iostream.h>
class X
{
    int i;
    static int si;
public:
    void Set_i(int i) { this->i = i; }
    void Print_i() { cout << "i = " << i << endl; }
    // Equivalent to:
    // void Print_i(X* this)
    // { cout << "X::i = " << this->i << endl; }
    static void Set_si(int si) { X::si = si; }

    static void Print_si()
    {
        cout << "X::si = " << X::si << endl;
    }
    // Print_si doesn't have a 'this' pointer
};
int X::si = 77;      // Initialize static data member
```

Static Members

```
void main()
{
    X xobj;
    // Non-static data members and functions belong to specific
    // instances (here xobj) of class X
    xobj.Set_i(11);
    xobj.Print_i();

    // static data members and functions belong to the class and
    // can be accessed without using an instance of class X
    X::Print_si();
    X::Set_si(22);
    X::Print_si();
}
```

This example produces the following output:

```
i = 11
X::si = 77
X::si = 22
```

Static Member Functions

You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.

A static member function does not have a **this** pointer. You can call a static member function using the **this** pointer of a nonstatic member function. In the following example, the nonstatic member function `printall()` calls the static member function `f()` using the **this** pointer:

// This example illustrates a static member function `f()`.

```
#include <iostream.h>
class c {
    static void f() { cout << "Here is i"
                    << i << endl;}

    static int i;
    int j;
public:
    c(int firstj): j(firstj) {}
    void printall();
};
void c::printall() {
    cout << "Here is j " << this->j << endl;
    this->f();
}
int c::i = 3;
void main() {
    class c C(0);
    C.printall();
}
```


Member Access

A static member function cannot be declared with the keyword **virtual**.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared.

Member Access

Member access determines if a class member is accessible in an expression or declaration. Note that accessibility and visibility are independent. Visibility is based on the scoping rules of C++. A class member can be visible and inaccessible at the same time. This section describes how you control the access to the individual nonderived class members by using access specifiers when you declare class members in a member list.

Classes and Access Control

C++ facilitates data abstraction and encapsulation by providing access control for members of class types.

For example, if you declare private data members and public member functions, a client program can only access the private members through the public member functions and friends of that class. Such a class would have *data hiding* because client programs do not have access to implementation details and are forced to use a public interface to manipulate objects of the class.

You can control access to class members by using access specifiers. In the following example, the class `abc` has three private data members `a`, `b`, and `c`, and three public member functions `add()`, `mult()`, and the constructor `abc()`. The `main()` function creates an object `danforth` of the `abc` class and then attempts to print the value of the member `a` for this object:

Member Access

```
// This example illustrates class member access specifiers

#include <iostream.h>

class abc
{
private:
    int a, b, c;
public:
    abc(int p1, int p2, int p3): a(p1), b(p2), c(p3) {}
    int add() { return a + b + c ; }
    int mult() { return a * b * c ; }
};

void main() {
    abc danforth(1,2,3);
    cout << "Here is the value of a " << danforth.a << endl;
        // This causes an error because a is not
        // a public member and cannot be accessed
        // directly
    }
```

Because class members are private by default, you can omit the keyword **private** in the definition of `abc`. Because `a` is not a public member, the attempt to access its value directly causes an error.

Access Specifiers

The three class member *access specifiers* have the following effect:

public class members

can be accessed by any function, file or class.

private class members

can be accessed only by member functions and friends of the class in which the member is declared.

protected class members

can be accessed only by member functions and friends of the class in which they are declared and by member functions and friends of classes derived with public or protected access from the class in which the protected members are declared. The access specifier **protected** can be used for nonbase class members, but it is equivalent to **private** unless it is used in a base class member declaration or in a base list. For more information, see “Protected Members” on page 324.

The default access for an individual class member depends on the class key used in the class declaration. Members of classes declared with the keyword **class** are private by default. Members of classes declared with the keyword **struct** or **union** are **public** by default.

Member Access

The access specifier **protected** is meaningful only in the context of derivation. You can control the access to inherited members (that is, base class members) by including access specifiers in the base list of the derived class declaration. You can also restore the access to an inherited member from a derived class by using an access declaration.

Access for inherited members is described in “Inherited Member Access” on page 324.

Member lists can include access specifiers as labels. Members declared after these labels have access as specified by the label they follow. An access specifier determines the access for members until another access specifier is used or until the end of the class declaration. You can use any number of access specifiers in any order.

The following example shows access specifiers in member lists.

```
class X
{
    int a;           // private data by default
public:
    void f(int);    // public function
    int b;         // public data
private:
    int c;         // private data
protected:
    void g(int);   // protected function
};
struct Y
{
    int a;         // public data by default
public:
    int b;         // public data
private:
    void g(int);   // private function
    int c;         // private data
};
```

Friends

A friend of a class *X* is a function or class that is granted the same access to *X* as the members of *X*. Functions declared with the **friend** specifier in a class member list are called *friend functions* of that class. Classes declared with the **friend** specifier in the member list of another class are called *friend classes* of that class.

A class *Y* must be defined before any member of *Y* can be declared a friend of another class.

In the following example, the friend function `print` is a member of class *Y* and accesses the private data members `a` and `b` of class *X*.

// This example illustrates a friend function.

```
#include <iostream.h>
class X;
class Y
{
public:
    void print(X& x);
};
class X
{
public:
    X() {a=1; b=2;}
private:
    int a, b;
    friend void Y::print(X& x);
};
void Y::print(X& x)
{
    cout << "A is " << x.a << endl;
    cout << "B is " << x.b << endl;
}
void main ()
{
    X xobj;
    Y yobj;
    yobj.print(xobj);
}
```

You can declare an entire class as a friend.

In the following example, the friend class *F* has a member function `print` that accesses the private data members `a` and `b` of class *X* and performs the same task as the friend function `print` in the above example. Any other members declared in class *F* also have access to all members of class *X*. In the example, the friend class *F* has not been previously declared, so an elaborated type specifier and a qualified type specifier are used to specify the class name.

Friends

```
// This example illustrates a friend class.

#include <iostream.h>
class X
{
public:
    X() {a=1; b=2;}          // constructor
private:
    int a, b;
    friend class F;        // friend class
};
class F
{
public:
    void print(X& x)
    {
        cout << "A is " << x.a << endl;
        cout << "B is " << x.b << endl;
    }
//    .
//    .
//    .
};
void main ()
{
    X xobj;
    F fobj;
    fobj.print(xobj);
}
```

Both the above examples produce the following output:

```
A is 1
B is 2
```

If the class has not been previously declared, use an elaborated type specifier and a qualified type specifier to specify the class name.

Friends

If the friend class has been previously declared, you can omit the keyword **class**, as shown in the following example:

```
class F;
class X
{
public:

    X() {a=1; b=2;}
private:
    int a, b;
    friend F; // elaborated-type-specifier not required
};
// .
// .
// .
```

Friend Scope

The name of a friend function or class first introduced in a friend declaration is not in the scope of the class granting friendship (also called the *enclosing class*) and is not a member of the class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

A class that is first declared in a friend declaration is equivalent to an **extern** declaration. For example:

```
class B {};
class A
{
    friend class B; // global class B is a friend of A
};
```

If the name of a friend class has been introduced before the friend declaration, the compiler searches for a class name that matches the name of the friend class beginning at the scope of the friend declaration. If the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

Friends

The scope of a friend class name is the first nonclass enclosing scope. For example:

```
class A {
    class B { // arbitrary nested class definitions
        friend class C;
    };
};
```

is equivalent to:

```
class C;
class A {
    class B { // arbitrary nested class definitions
        friend class C;
    };
};
```

If the friend function is a member of another class, you need to use the class member access operators. For example:

```
class A
{
public:
    int f() { /* ... */ }
};
class B
{
    friend int A::f();
};
```

Friends of a base class are not inherited by any classes derived from that base class.

For more information about friend scope, see “Scope of Class Names” on page 246.

Friend Access

A friend of a class can access the private and protected members of that class. Normally, you can only access the private members of a class through member functions of that class, and you can only access the protected members of a class through member functions of a class or classes derived from that class.

Friend declarations are not affected by access specifiers.

For more information on access, see also “Member Access” on page 270.

Chapter 11. Overloading

Overloading enables you to redefine functions and most standard C++ operators. Typically, you overload a function or operator if you want to extend the operations the function or operator performs to different data types.

This chapter discusses:

Overloading Functions	277
Argument Matching in Overloaded Functions	279
Overloading Operators	281
Overloading Unary Operators	285
Overloading Binary Operators	286
Special Overloaded Operators	287

Related Information

- Chapter 4, "Expressions and Operators" on page 93
- Chapter 6, "Functions" on page 137
- Chapter 9, "Classes" on page 239

Overloading Functions

You can overload a function by having multiple declarations of the same function name in the same scope. The declarations differ in the type and number of arguments in the argument list. When an overloaded function is called, the correct function is selected by comparing the types of the actual arguments with the types of the formal arguments.

Consider a function `print`, which displays an **int**. As shown in the following example, you can overload the function `print` to display other types, for example, **double** and **char***. You can have three functions with the same name, each performing a similar operation on a different data type.

```
// This example illustrates function overloading.

#include <iostream.h>

void print(int i) { cout << " Here is int " << i << endl; }
void print(double f) { cout << " Here is float "
                    << f << endl; }
void print(char* c) { cout << " Here is char* " << c << endl; }
void main() {
    print(10);           // calls print(int)
    print(10.10);       // calls print(double)
    print("ten");        // calls print(char*)
}
```


Overloading Functions

Declaration Matching

Two function declarations are identical if all of the following are true:

- They have the same function name
- They are declared in the same scope
- They have identical argument lists

When you declare a function name more than once in the same scope, the second declaration of the function name is interpreted by the compiler as follows:

- If the return type, argument types, and number of arguments of the two declarations are identical, the second declaration is considered a declaration of the same function as the first.
- If only the return types of the two function declarations differ, the second declaration is an error.
- If either the argument types or number of arguments of the two declarations differ, the function is considered to be overloaded.

Restrictions on Overloaded Functions

- Functions that differ only in return type cannot have the same name.
- Two member functions that differ only in that one is declared with the keyword **static** cannot have the same name.
- A **typedef** is a synonym for another type, not a separate type. The following two declarations of `spadina()` are declarations of the same function:

```
typedef int I;  
void spadina(float, int);  
void spadina(float, I);
```

- A member function of a derived class is not in the same scope as a member function in a base class with the same name. A derived class member hides a base class member with the same name.
- Argument types that differ only in that one is a pointer `*` and the other is an array `[]` are identical. The following two declarations are equivalent:

```
f(char*);  
f(char[10]);
```

Only the second and subsequent array dimensions are significant.

- The **const** and **volatile** type-specifiers are ignored in distinguishing argument types when they appear at the outermost level of the argument type specification. The following declarations are equivalent:

```
int f (int);  
int f (const int);  
int f (volatile int);
```

Pointers and references to types considered distinct parameter types.

For more information on functions, see Chapter 6, “Functions” on page 137.

Argument Matching in Overloaded Functions

When an overloaded function or overloaded operator is called, the compiler chooses the function declaration with the *best match* on all arguments from all the function declarations that are visible. The compiler compares the actual arguments of a function call with the formal arguments of all declarations of the function that are visible. For a best match to occur, the compiler must be able to distinguish a function that:

- Has at least as good a match on all arguments as any other function with the same name
- Has at least one better argument match than any other function with the same name

If no such function exists, the call is not allowed. A call to an overloaded function has three possible outcomes. The compiler can find:

- An exact match
- No match
- An ambiguous match

An ambiguous match occurs when the actual arguments of the function call match more than one overloaded function. The ANSI C++ working group has made the following clarifications of how and which built-in operators should participate in the operator overload resolution process:

- All appropriate built-in operators that are specified in section 13.6 of the September 1995 ANSI draft participate in the process.
- If you are using binary operators with pointer operands, the pointer operands must point to the same data type.
- The compiler distinguishes between integral/arithmetic and promoted integral/arithmetic type operands.
- The compiler only chooses the user-defined operator[] function for the subscripting expression x[y], if the overload resolution process chooses it as the best function match over the builtin operator.
- For builtin assignment operators, conversions of the left operand are restricted as follows:
 - No temporary objects are introduced to hold the left operand.
 - No user-defined conversions are applied to achieve a type match with the left operand.

Argument matching can include performing standard and user-defined conversions on the arguments to match the actual arguments with the formal arguments. Only a single user-defined conversion is performed in a sequence of conversions on an actual argument. In addition, the *best-matching* sequence of standard conversions is performed on an actual argument. The best-matching sequence is the shortest sequence of conversions between two standard types. For example, the conversion:

```
int -> float -> double
```

Argument Matching

can be shortened to the best-matching conversion sequence:

```
int -> double
```

because the conversion from **int** to **double** is allowed.

Trivial conversions, described on page 281, do not affect the choice of conversion sequence.

Sequence of Argument Conversions

Argument-matching conversions occur in the following order:

1. An exact match in which the actual arguments match exactly (including a match with one or more trivial conversions) with the type and number of formal arguments of one declaration of the overloaded function
2. A match with promotions in which a match is found when one or more of the actual arguments is promoted
3. A match with standard conversions in which a match is found when one or more of the actual arguments is converted by a standard conversion
4. A match with user-defined conversions in which a match is found when one or more of the actual arguments is converted by a user-defined conversion
5. A match with ellipses

Match through promotion follows the rules for “Integral Promotions” and “Standard Type Conversions” on page 132.

You can override an exact match by using an explicit cast. In the following example, the second call to `f()` matches with `f(void*)`:

```
void f(int);
void f(void*);
//      .
//      .
//      .
void main()
{
    f(0xaabb);           // matches f(int);
    f((void*) 0xaabb);  // matches f(void*)
}
```

The implicit first argument for a nonstatic member function or operator is the **this** pointer. It refers to the class object for which the member function is called. When you overload a nonstatic member function, the first implicit argument, the **this** pointer, is matched with the object or pointer used in the call to the member function. User-defined conversions are not applied in this type of argument matching for overloaded functions or operators.

When you call an overloaded member function of class `X` using the `.` (dot) or `->` (arrow) operator, the **this** pointer has type `X* const`. The type of the **this** pointer for a constant

Overloading Operators

object is `const X* const`. The type of the **this** pointer for a volatile object is `volatile X* const`.

The **this** pointer is described on page 262. The class-member access operators are described in “Dot Operator `.`” on page 102 and “Arrow Operator `->`” on page 103.

Trivial Conversions

Functions cannot be distinguished if they have the same name and have arguments that differ only in that one is declared as a reference to a type and the other is that type. You cannot have two functions with the same name and with arguments differing only in this respect. Because the following two declarations cannot be distinguished, the second one causes an error:

```
double f(double i); // declaration
//      .
//      .
//      .
double f(double &i); // error
```

However, functions with the same name having arguments that differ only in that one is a pointer or reference and the other is a pointer to **const** or **const** reference can be distinguished. Functions with the same name having arguments that differ only in that one is a pointer or reference and the other is a pointer to **volatile** or **volatile** reference can also be distinguished. For the purpose of finding a best match of arguments, functions that have a **volatile** or **const** match (not requiring a trivial conversion) are better than those that have a **volatile** or **const** mismatch.

For more information on conversions, see “Standard Type Conversions” on page 132 and “User-Defined Conversions” on page 304.

Overloading Operators

You can overload one of the standard C++ operators by redefining it to perform a particular operation when it is applied to an object of a particular class. Overloaded operators must have at least one argument that has class type. An overloaded operator is called an *operator function* and is declared with the keyword **operator** preceding the operator. Overloaded operators are distinct from overloaded functions, but, like overloaded functions, they are distinguished by the number and types of operands used with the operator.

You can overload any of the following operators:

+	-	*	/	%	^	&		~
!	=	<	>	+=	--	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete					

where `()` is the function call operator and `[]` is the subscript operator.

Overloading Operators

Consider the standard + (plus) operator. When this operator is used with operands of different standard types, the operators have slightly different meanings. For example, the addition of two integers is not implemented in the same way as the addition of two floating-point numbers. C++ allows you to define your own meanings for the standard C++ operators when they are applied to class types. In the following example, a class called `complx` is defined to model complex numbers, and the + (plus) operator is redefined in this class to add two complex numbers.

```
// This example illustrates overloading the plus (+) operator.

#include <iostream.h>
class complx
{
    double real,
           imag;
public:
    complx( double real = 0., double imag = 0.); // constructor
    complx operator+(const complx& c) const;    // operator+()
};
// define constructor
complx::complx( double r, double i )
{
    real = r; imag = i;
}
// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result;
}
void main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}
```

General Rules for Overloading Operators

You can overload both the unary and binary forms of:

+ - * &

When an overloaded operator is a member function, the first operand is matched against the class type of the overloaded operator. The second operand, if one exists, is matched against the argument in the overloaded operator call.

When an overloaded operator is a nonmember function, at least one operand must have class or enumeration type. The first operand is matched against the first argument in the overloaded operator call. The second operand, if one exists, is matched against the second argument in the overloaded operator call.

Overloading Operators

The argument-matching conventions and rules described in “Argument Matching in Overloaded Functions” on page 279 apply to overloaded operators.

Operands of Overloaded Operators

An overloaded operator must be either a member function, as shown in the following example:

```
class X
{
public:
    X operator!();
    X& operator=(X&);
    X operator+(X&);
};
X X::operator!() { /* ... */ }
X& X::operator=(X& x) { /* ... */ }
X X::operator+(X& x) { /* ... */ }
```

or take at least one argument of class, a reference to a class, an enumeration, or a reference to an enumeration, as shown below:

```
class Y;
{
// .
// .
// .
};
class Z;
{
// .
// .
// .
};
Y operator!(Y& y);
Z operator+(Z& z, int);
```

Overloading Operators

Usually, overloaded operators are invoked using the normal operator syntax. You can also call overloaded operators explicitly by qualifying the operator name. For example, for the class `complx`, described above, you can call the overloaded `+` (plus) operator either implicitly or explicitly as shown below.

```
// This example shows implicit and explicit calls
// to an overloaded plus (+) operator.

class complx
{
    double real,
          imag;
public:
    complx( double real = 0., double imag = 0.);
    complx operator+(const complx&) const;
};
//      .
//      .
//      .
void main()
{
    complx x(4,4);
    complx y(6,6);
    complx u = x.operator+(y); // explicit call
    complx z = x + y;         // implicit call to complx::operator+()
}
```

Restrictions on Overloaded Operators

- The following C++ operators cannot be overloaded:
`.` `.*` `::` `?:`
- You cannot overload the preprocessing symbols `#` and `##`.
- You cannot change the precedence, grouping, or number of operands of the standard C++ operators.
- An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.
- You must declare the overloaded `=`, `[]`, `()` and `->` operators as nonstatic member functions to ensure that they receive lvalues as their first operands.
- The operators **new** and **delete** do not follow the general rules described in this section. Overloading **new** and **delete** is described in “Overloaded new and delete” on page 291.
- All operators except the `=` operator are inherited. “Copy by Assignment” on page 312 describes the behavior of the assignment operator.
- Unless they are explicitly mentioned in “Special Overloaded Operators” on page 287, overloaded unary and binary operators follow the rules outlined in “Overloading Unary Operators” on page 285 and “Overloading Binary Operators” on page 286.

Overloading Unary Operators

For more information on standard C++ operators, see Chapter 4, “Expressions and Operators” on page 93.

Overloading Unary Operators

You can overload a prefix unary operator by declaring a nonmember function taking one argument or a nonstatic member function taking no arguments.

When you prefix a class object with an overloaded unary operator, for example:

```
class X
{
//      .
//      .
//      .
};
void main ()
{
    X x;
    !x;      // overloaded unary operator
}
```

the operator function call `!x` can be interpreted as:

`x.operator!()`

or

`operator!(x)`

depending on the declarations of the operator function. If both forms of the operator function have been declared, argument matching determines which interpretation is used.

For more information on standard unary operators, see “Unary Expressions” on page 103.

Overloading Binary Operators

Overloading Binary Operators

You can overload a binary operator by declaring a nonmember function taking two arguments or a nonstatic member function taking one argument.

When you use a class object with an overloaded binary operator, for example:

```
class X
{
//      .
//      .
//      .
};
void main ()
{
    X x;
    int y=10;
    x*y;      // overloaded binary operator
}
```

the operator function call `x*y` can be interpreted as:

`x.operator*(y)`

or

`operator*(x,y)`

depending on the declarations of the operator function. If both forms of the operator function have been declared, argument matching determines which interpretation is used.

For more information on standard binary operators, see "Binary Expressions" on page 114.

Special Overloaded Operators

Special Overloaded Operators

The following overloaded operators do not fully follow the rules for unary or binary overloaded operators:

- Assignment
- Function call
- Subscripting
- Class member access
- Increment and decrement
- **new** and **delete**

Overloaded Assignment

You can only overload an assignment operator by declaring a nonstatic member function. The following example shows how you can overload the assignment operator for a particular class:

```
class X
{
public:
    X();
    X& operator=(X&);
    X& operator=(int);

    // .
    // .
    // .
};
X& X::operator=(X& x) { /* ... */ }
X& X::operator=(int i) { /* ... */ }
// .
// .
// .
void main()
{
    X x1, x2;
    x1 = x2;    // call x1.operator=(x2)
    x1 = 5;    // call x1.operator=(5)
}
```

You cannot declare an overloaded assignment operator that is a nonmember function.

Overloaded assignment operators are not inherited.

If a copy assignment operator function is not defined for a class, the copy assignment operator function is defined by default as a memberwise assignment of the class members. If assignment operator functions exist for base classes or class members, these operators are used when the compiler generates default copy assignment operators. See “Copy by Assignment” on page 312 for more information.

For more information on standard assignment operators, see “Assignment Expressions” on page 126.

Special Overloaded Operators

Overloaded Function Calls

The operands are *function_name* and an optional *expression_list*. The operator function **operator()** must be defined as a nonstatic member function. You cannot declare an overloaded function call operator that is a nonmember function.

If you make the following call for the class object x:

```
x (arg1, arg2, arg3)
```

it is interpreted as

```
x.operator()(arg1, arg2, arg3)
```

Unlike all other overloaded operators, you can provide default arguments and ellipses in the argument list for the function call operator. For example:

```
class X
{
public:
    X& operator() (int = 5);
};
//      .
//      .
//      .
```

For more information on the standard function call operator, see “Function Calls ()” on page 100.

Overloaded Subscripting

An expression containing the subscripting operator has syntax of the form:

identifier [*expression*]

and is considered a binary operator. The operands are *identifier* and *expression*. The operator function `operator[]` must be defined as a nonstatic member function. You cannot declare an overloaded subscript operator that is a nonmember function.

A subscripting expression for the class object x:

```
x [y]
```

is interpreted as `x.operator[](y)`. It is not interpreted as `operator[](x,y)` because it is defined as a nonstatic member function.

For more information on the standard subscripting operator, see “Array Subscript [] (Array Element Specification)” on page 102.

Overloaded Class Member Access

An expression containing the class member access `->` (arrow) operator has syntax of the form:

identifier `->` *name-expression*

Special Overloaded Operators

and is considered a unary operator. The operator function `operator->()` must be defined as a nonstatic member function.

The following restrictions apply to class member access operators:

- You cannot declare an overloaded arrow operator that is a nonmember function.
- You cannot overload the class member access `.` (dot) operator.

Consider the following example of overloading the `->` (arrow) operator:

```
class Y
{
public:
    void f();
};
class X
{
public:
    Y* operator->();
};
X x;
//      .
//      .
//      .
x->f();
```

Here `x->f()` is interpreted as:

```
( x.operator->() )-> f()
```

`x.operator->()` must return either a reference to a class object or a class object for which the overloaded `operator->` function is defined or a pointer to any class. If the overloaded `operator->` function returns a class type, the class type must not be the same as the class declaring the function, and the class type returned must contain its own definition of an overloaded `->` operator function.

For more information on the standard class member access arrow operator, see “Arrow Operator `->`” on page 103.

Special Overloaded Operators

Overloaded Increment and Decrement

The prefix increment operator `++` can be overloaded for a class type by declaring a nonmember function operator with one argument of class type or a reference to class type, or by declaring a member function operator with no arguments.

In the following example, the increment operator is overloaded in both ways:

// This example illustrates an overloaded prefix increment operator.

```
class X
{
    int a;
public:
    operator++;           // member prefix increment operator
};
class Y { /* ... */ };
operator++(Y& y);       // nonmember prefix increment operator
//      .
//      .
//      .
// Definitions of prefix increment operator functions
//      .
//      .
//      .

void main()
{
    X x;
    Y y;
    ++x;                 // x.operator++
    x.operator++;        // x.operator++
    operator++(y);       // nonmember operator++
    ++y;                 // nonmember operator++
}
```

The postfix increment operator `++` can be overloaded for a class type by declaring a nonmember function operator `operator++()` with two arguments, the first having class type and the second having type `int`. Alternatively, you can declare a member function operator `operator++()` with one argument having type `int`. The compiler uses the `int` argument to distinguish between the prefix and postfix increment operators. For implicit calls, the default value is zero.

Special Overloaded Operators

For example:

// This example illustrates an overloaded postfix increment operator.

```
class X
{
    int a;
public:
    operator++(int); // member postfix increment operator
};
operator++(X x, int i); // nonmember postfix increment operator
// .
// .
// .
// Definitions of postfix increment operator functions
// .
// .
// .
void main()
{
    X x;
    x++; // x.operator++
        // default zero is supplied by compiler
    x.operator++(0); // x.operator++
    operator++(x,0); // nonmember operator++
}
```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

For more information on the standard postfix and prefix increment operators, see “Increment ++” on page 103. For more information on the standard postfix and prefix decrement operators, see “Decrement --” on page 104.

Overloaded new and delete

You can implement your own memory management scheme for a class by overloading the operators **new** and **delete**. The overloaded operator **new** must return a **void***, and its first argument must have type **size_t**. The overloaded operator **delete** must return a **void** type, and its first argument must be **void***. The second argument for the overloaded **delete** operator is optional and, if present, it must have type **size_t**. You can only define one **delete** operator function for a class.

Type **size_t** is an implementation dependent unsigned integral type defined in `<stddef.h>`.

The size argument is required because a class can inherit an overloaded **new** operator. The derived class can be a different size than the base class. The size argument ensures that the correct amount of storage space is allocated or deallocated for the object.

Special Overloaded Operators

When **new** and **delete** are overloaded within a class declaration, they are static member functions whether they are declared with the keyword **static** or not. They cannot be virtual functions.

You can access the standard, nonoverloaded versions of **new** and **delete** within a class scope containing the overloading **new** and **delete** operators by using the `::` (scope resolution) operator to provide global access.

For more information on the class member operators **new** and **delete**, see “Free Store” on page 299. For more information on the standard **new** and **delete** operators, see “C++ new Operator” on page 108 and “C++ delete Operator” on page 113.

Constructors and Destructors Overview

Chapter 12. Special Member Functions

This chapter introduces the special member functions that are used to create, destroy, convert, initialize, and copy class objects. This chapter discusses:

Constructors and Destructors Overview	293
Constructors	294
Destructors	297
Free Store	299
Temporary Objects	302
User-Defined Conversions	304
Initialization by Constructor	306
Copying Class Objects	311

Related Information

- Chapter 6, “Functions” on page 137
- Chapter 9, “Classes” on page 239
- Chapter 10, “Class Members and Friends” on page 253
- Chapter 11, “Overloading” on page 277
- Chapter 13, “Inheritance” on page 315

Constructors and Destructors Overview

Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword **virtual**.
- Constructors and destructors cannot be declared **static**, **const**, or **volatile**.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if a constructor is declared with the keyword **protected**, only derived classes and friends can use it to create class objects. Class member access is described in “Member Access” on page 270.

Constructors

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object its **this** pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the **new** operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the **delete** operator. The global **new** and **delete** operators are described in “C++ new Operator” on page 108 and “C++ delete Operator” on page 113.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

You can call member functions from constructors or destructors. You can call a virtual function, either directly or indirectly, from a constructor or destructor. In this case, the function called is the one defined in the class or base class containing the constructor (or destructor), but not a function defined in any class derived from the class being constructed. This avoids the possibility of accessing an unconstructed object from a constructor or destructor.

Constructors

A *constructor* is a member function with the same name as its class. For example:

```
class X
{
public:
    X();    // constructor for class X
    // .
    // .
    // .
};
```

Constructors are used to create, and can initialize, objects of their class type. Initialization of class objects using constructors is described in “Initialization by Constructor” on page 306.

Default Constructors

A *default constructor* is a constructor that either has no arguments, or, if it has arguments, *all* the arguments have default values. If no user-defined constructor exists for a class and one is needed, the compiler creates a default constructor, with public access, for that class. No default constructor is created for a class that has any constant or reference type members.

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor. For more information, see “Default Arguments in

Constructors

C++ Functions” on page 156. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

A *copy constructor* is used to make a copy of one class object from another class object of the same class type. A copy constructor is called with a single argument that is a reference to its own class type. You cannot use a copy constructor with an argument of the same type as its class; you must use a reference. You can provide copy constructors with additional default arguments. If a user-defined copy constructor does not exist for a class and one is needed, the compiler creates a copy constructor, with public access, for that class. It is not created for a class if any of its members or base classes have an inaccessible copy constructor.

The following code fragment shows two classes with constructors, default constructors, and copy constructors:

```
class X
{
public:
    X();                // default constructor, no arguments
    X(int, int , int = 0); // constructor
    X(const X&);        // copy constructor
    X(X);              // error, incorrect argument type
};
class Y
{
public:
    Y( int = 0);        // default constructor with one
                       // default argument
    Y(const Y&, int = 0); // copy constructor
};
```

Constructors

Construction Order of Class Objects

If a class has a base class or members with constructors when it is constructed, the constructor for the base class is called, followed by any constructors for members. The constructor for the derived class is called last. Virtual base classes are constructed before nonvirtual base classes. When more than one base class exists, the base class constructors are called in the order that their classes appear in the base list, as shown in the following example.

```
class B1 { public: B1(); };
class B2
{
public:
    B2();
    B1 b1obj;
};
class B3 { public: B3(); };
//      .
//      .
//      .
class D : public B1, public B2, public B3
{
public:
    D();
    ~D();
};
//      .
//      .
//      .
void main ()
{
    D object;
}
```

In the above example, the constructors for object are called in the following order:

```
B1();    // first base constructor declared
B1();    // member constructor for B2::b1obj
B2();    // second base constructor declared
B3();    // last base constructor declared
D();     // derived constructor called last
```

Note that the construction of class D involves construction of the base classes B1, B2, and B3. The construction of base class B2 involves the construction of its class B1 member object. When class B2 is constructed, the constructor for class B1 is called in addition to B2's own constructor.

As explained above, the second call to the constructor of B1 followed by the call to the constructor of B2 is part of the construction of B2.

For more information, see "Construction Order of Derived Class Objects" on page 311.

Explicitly Constructing Objects

You cannot call constructors directly. You use a function style cast to explicitly construct an object of the specified type. In the following example, a constructor is used as an initializer to create a named object.

```
#include <iostream.h>
class X
{
public:
    X (int, int , int = 0); // constructor with default argument
private:
    int a, b, c;
    int f();
};
X::X (int i, int j, int k) { a = i; b = j; c = k; }
// .
// .
// .
void main ()
{
X xobject = X(1,2,3); // explicitly create and initialize
                  // named object with constructor call
}
```

Destructors

A *destructor* is a member function with the same name as its class prefixed by a `~` (tilde).

For example:

```
class X
{
public:
    X(); // constructor for class X
    ~X(); // destructor for class X
// .
// .
// .
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared **const**, **volatile**, or **static**. A destructor can be declared **virtual** or pure **virtual**. A union cannot have as a member an object of a class with a destructor.

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

Destructors

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class or a member of a base class has a destructor and a class derived from that base class does not declare a destructor, a default destructor is generated. The default destructor calls the destructors of the base class and members of the derived class. Default destructors are generated with default public access.

Destructors are called in the reverse order to constructors:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block. For more information, see “Constructors and Destructors in Exception Handling” on page 376.

Destructors are implicitly called when an automatic or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the **delete** operator for objects created with the **new** operator.

For example:

```
#include <string.h>
class Y
{
private:
    char * string;
    int number;
public:
    Y(const char* n,int a); // constructor
    ~Y() { delete[] string; } // destructor
};
Y::Y(const char* n, int a) // define class Y constructor
{
    string = strcpy(new char[strlen(n) + 1 ], n);
    number = a;
}
void main ()
{
    Y yobj = Y("somestring", 10); // create and initialize
                                // object of class Y

    // .
    // .
    // .
    // destructor ~Y is called before control returns from main()
}
```

Free Store

You can use a destructor explicitly to destroy objects, although this practice is not recommended. If an object has been placed at a specific address by the **new** operator, you can call the destructor of the object explicitly to destroy it. An explicitly called destructor cannot delete storage.

Note: You can only call destructors for class types. You cannot call destructors for simple types. The call to the destructor in the following example causes the compiler to issue a warning:

```
int * ptr;
ptr -> int::~~int(); // warning
```

Free Store

Free store is used for dynamic allocation of memory. The **new** and **delete** operators are used to allocate and deallocate free store, respectively. You can define your own versions of **new** and **delete** for a class by overloading them. You can supply the **new** and **delete** operators with additional arguments. When **new** and **delete** operate on class objects, the class member operator functions **new** and **delete** are called, if they have been declared.

If you create a class object with the **new** operator, one of the operator functions **operator new()** or **operator new[]()** (if they have been declared) is called to create the object. An **operator new()** or **operator new[]()** for a class is always a static class member, even if it is not declared with the keyword **static**. It has a return type **void*** and its first argument must be the size of the object type and have type **size_t**. It cannot be virtual.

Type **size_t** is an implementation dependent unsigned integral type defined in `<stddef.h>`.

When you overload the **new** operator, you must declare it as a class member, returning type **void***, with first argument **size_t**, as described above. You supply additional arguments in the declaration of **operator new()** or **operator new[]()**. Use the placement syntax to specify values for these arguments in an allocation expression.

The following example shows two overloaded **new** operator functions.

Free Store

```
#include <stddef.h>
class X
{
public:
    void* operator new(size_t);
    void* operator new(size_t, int);
};
// .
// .
// .
void main ()
{
    X* ptr1 = new X;           // calls X::operator new(sizeof(X))
    X* ptr2 = new(10) X;      // calls X::operator
                             // new(sizeof(X),10)
}
```

The **delete** operator destroys an object created by the **new** operator. The operand of **delete** must be a pointer returned by **new**. If **delete** is called for an object with a destructor, the destructor is invoked before the object is deallocated.

If you destroy a class object with the **delete** operator, the operator function **operator delete()** or **operator delete[]()** (if they have been declared) is called to destroy the object. An **operator delete()** or **operator delete[]()** for a class is always a static member, even if it is not declared with the keyword **static**. Its first argument must have type **void***. Because **operator delete()** and **operator delete[]()** have a return type **void**, they cannot return a value. They cannot be virtual.

When you overload the **delete** operator, you must declare it as class member, returning type **void**, with first argument having type **void***, as described above. You can add a second argument of type **size_t** to the declaration. You can only have one **operator delete()** or **operator delete[]()** for a single class.

Overloading **new** and **delete** is described in “Overloaded new and delete” on page 291.

Free Store

The following example shows the declaration and use of the operator functions **operator new()** and **operator delete()**:

```
#include <stddef.h>
class X
{
public:
    void* operator new(size_t);
    void operator delete(void*);           // single argument
};
class Y
{
public:
    void operator delete(void*, size_t); // two arguments
};
// .
// .
// .
void main ()
{
    X* ptr = new X;
    delete ptr;    // call X::operator delete(void*)
    Y* yptr;
// .
// .
// .
    delete yptr;    // call Y::operator delete(void*, size_t)
                   // with size of Y as second argument
}
```

The result of trying to access a deleted object is undefined because the value of the object can change after deletion.

If **new** and **delete** are called for a class object that does not declare the operator functions **new** and **delete**, or they are called for a nonclass object, the global operators **new** and **delete** are used. The global operators **new** and **delete** are provided in the C++ library.

Note: The C++ operators for allocating and deallocating arrays of class objects are **operator new[]()** and **operator delete[]()**. They are described in “C++ new Operator” on page 108 and “C++ delete Operator” on page 113.

Temporary Objects

Temporary Objects

It is sometimes necessary for the compiler to create temporary objects. They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the **throw** expression.

When a temporary object is created to initialize a reference variable, the name of the temporary object has the same scope as that of the reference variable. When a temporary object is created during the evaluation of an expression, it exists until there is a break in the flow of control of the program.

If a temporary object is created for a class with constructors, the compiler calls the appropriate (matching) constructor to create the temporary object.

When a temporary object is destroyed and a destructor exists, the compiler calls the destructor to destroy the temporary object. When you exit from the scope in which the temporary object was created, it is destroyed. If a reference is bound to a temporary object, the temporary object is destroyed when the reference passes out of scope unless it is destroyed earlier by a break in the flow of control. For example, a temporary object created by a constructor initializer for a reference member is destroyed on leaving the constructor.

Temporary Objects

The following example shows two expressions in which temporary objects are constructed:

```
class Y
{
public:
    Y(int)={ };
    Y(Y&){ };
    ~Y()={ };
};
Y add(Y y) { return y; }
//      .
//      .
//      .
void main ()
{
    Y obj1(10);
    Y obj2 = add(Y(5));    // one temporary created
    obj1 = add(obj1);     // two temporaries created
}
```

In the above example, a temporary object of class type `Y` is created to construct `Y(5)` before it is passed to the function `add()`. Because `obj2` is being constructed, the function `add()` can construct its return value directly into `obj2`, so another temporary object is not created. A temporary object of class type `Y` is created when `obj1` is passed to the function `add()`. Because `obj1` has already been constructed, the function `add()` constructs its return value into a temporary object. This second temporary object is then assigned to `obj1` using an assignment operator.

Related Information

- “Initializing References” on page 91
- Chapter 4, “Expressions and Operators” on page 93
- “Standard Type Conversions” on page 132
- Chapter 6, “Functions” on page 137
- “Using Exception Handling” on page 368

User-Defined Conversions

User-Defined Conversions

User-defined conversions allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:

- Conversion by constructor
- Conversion functions.

For more information, see “Standard Type Conversions” on page 132.

Conversion by Constructor

You can call a class constructor with a single argument to convert from the argument type to the type of the class.

For example:

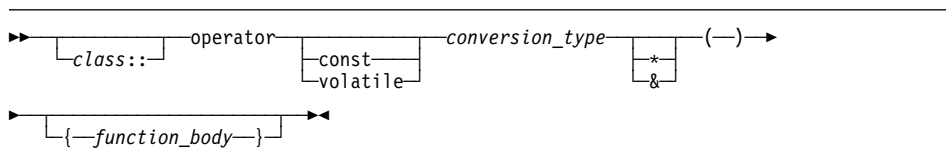
```
class Y
{
    int a,b;
    char* name;
public:
    Y(int i);
    Y(const char* n, int j = 0);
};
void add(Y);
// .
// .
// .
void main ()
{
    // code                equivalent code
    Y obj1 = 2;             // obj1 = Y(2)
    Y obj2 = "somestring"; // obj2 = Y("somestring",0)
    obj1 = 10;             // obj1 = Y(10)
    add(5);                // add(Y(5))
}
```

At most one user-defined conversion, either a constructor or conversion function, is implicitly applied to a class object. When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class.

User-Defined Conversions

Conversion Functions

You can define a member function of a class, called a *conversion function*, that converts from the type of its class to another specified type.



The conversion function specifies a conversion from the class type the conversion function is a member of, to the type specified by the name of the conversion function. Classes, enumerations, and **typedef** names cannot be declared or defined as part of the function name.

The following code fragment shows a conversion function called `operator int()`:

```
class Y
{
    int b;
public:
    operator int();
};
Y::operator int() {return b;}

void f(Y obj )
{
    // each value assigned is converted by Y::operator int()
    int i = int(obj);
    int j = (int)obj;
    int k = i + obj;
}
```

Conversion functions have no arguments, and the return type is implicitly the conversion type. Conversion functions can be inherited. You can have virtual conversion functions but not static ones.

Only one user-defined conversion is implicitly applied to a single value. User-defined conversions must be unambiguous, or they are not called.

If a conversion function is declared with the keyword **const**, the keyword has no effect on the function except for acting as a tie-breaker when there is more than one conversion function that could be applied. Specifically, if more than one conversion function could be applied, all of these functions are compared. If any of these functions is declared with the keyword **const**, **const** is ignored for the purposes of this comparison. If one of these functions is a best match, this function is applied. If there is no best match, the functions are compared again, but this time **const** is not ignored.

Initialization by Constructor

Initialization by Constructor

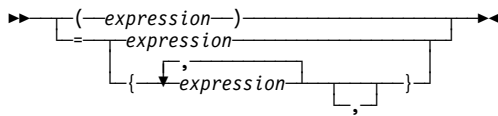
A class object with a constructor must be explicitly initialized or have a default constructor. Explicit initialization using a constructor is the only way, except for aggregate initialization, to initialize nonstatic constant and reference class members.

A class object that has no constructors, no virtual functions, no private or protected members, and no base classes is called an aggregate. Aggregates are described in “Structures” on page 70 and “Unions” on page 78.

Explicit Initialization

Class objects with constructors can be initialized with a parenthesized expression list. This list is used as an argument list for the call of a constructor that is used to initialize the class. You can also call a constructor with a single initialization value using the = operator. Because this type of expression is an initialization, not an assignment, the assignment operator function, if one exists, is not called. This value is used as a single argument for the call of a constructor. The type of the single argument must match the type of the first argument to the constructor. If the constructor has remaining arguments, these arguments must have default values.

The syntax for an initializer that explicitly initializes a class object with a constructor is:



Initialization by Constructor

The following example shows the declaration and use of several constructors that explicitly initialize class objects:

```
// This example illustrates explicit initialization
// by constructor.

#include <iostream.h>
class complx
{
    double re, im ;
public:
    complx(); // default constructor
    complx(const complx& c) {re = c.re; im = c.im;}
        // copy constructor
    complx( double r, double i = 0.0) {re = r; im = i;}
        // constructor with default trailing argument
    void display()
    {
        cout << "re = " << re << " im = " << im << endl;
    }
};
// .
// .
// .

void main ()
{
    complx one(1); // initialize with complx(double, double)
    complx two = one; // initialize with a copy of one
        // using complx::complx(const complx&)
    complx three = complx(3,4); // construct complx(3,4)
        // directly into three
    complx four; // initialize with default constructor
    complx five = 5; // complx(double, double) & construct
        // directly into five

    one.display();
    two.display();
    three.display();
    four.display();
    five.display();
}
```

Initialization by Constructor

The above example produces the following output:

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

Constructors can initialize their members in two different ways. A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

```
complex( double r, double i = 0.0) {re = r; im = i;}
```

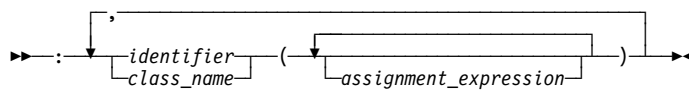
Or a constructor can have an initializer list within the definition but prior to the function body:

```
complex ( double r, double i = 0) : re(r), im(i) { /* ... */ }
```

Both methods assign the argument values to the appropriate data members of the class. The second method must be used to initialize base classes from within a derived class to initialize constant and reference members and members with constructors.

Initializing Base Classes and Members

You can initialize immediate base classes and derived class members that are not inherited from base classes by specifying initializers in the constructor definition prior to the function body. The syntax for a constructor initializer is:



In a constructor that is not inline, include the initialization list as part of the function definition, not as part of the class declaration.

Initialization by Constructor

For example:

```
class B1
{
    int b;
public:
    B1();
    B1(int i) : b(i) { /* ... */ } // inline constructor
};
class B2
{
    int b;
protected:
    B2();
    B2(int i); // noninline constructor
};
// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { /* ...*/ }
// .
// .
// .
class D : public B1, public B2
{
    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(), d1(i) {d2 = j;}
};
```

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor. In the above example, if you leave out the call `B2()` in the constructor of class `D` (as shown below), a constructor initializer with an empty expression list is automatically created to initialize `B2`. The constructors for class `D`, shown above and below, result in the same construction of an object of class `D`.

```
class D : public B1, public B2
{
    int d1, d2;
public:
    // call B2() generated by compiler
    D(int i, int j) : B1(i+1), d1(i) {d2 = j;}
};
```


Initialization by Constructor

Note: You must declare base constructors with the access specifiers **public** or **protected** to enable a derived class to call them.

For example:

```
class B1
{
    int b;
public:
    B1();
    B1(int i) : b(i) { /* ... */ }
};
class B2
{
    int b;
protected:
    B2();
    B2(int i);
};
B2::B2(int i) : b(i) { /* ... */ }
class B4
{
public:
    B4();           // public constructor for B4
    int b;
private:
    B4(int);       // private constructor for B4
};
// .
// .
// .

class D : public B1, public B2, public B4
{
    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(i+2) ,
                    B4(i) {d1 = i; d2 = j; }
                    // error, attempt to access private constructor B4()
};
```

To ensure a valid call, you can change the code as follows:

```
public:
    D(int i, int j) : B1(i+1), B2(i+2) {d1 = i; d2 = j;}
                    // valid, calls public constructor for B4
```

Copying Class Objects

Construction Order of Derived Class Objects

When a derived class object is created using constructors, it is created in the following order:

1. Virtual base classes are initialized, in the order they appear in the base list.
2. Nonvirtual base classes are initialized, in declaration order.
3. Class members are initialized in declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

In the following code fragment, the constructor for class B1 is called before the member d1 is initialized. The value passed to the constructor for class B1 is undefined.

```
class B1
{
    int b;
public:
    B1();
    B1(int i) {b = i;}
};
// .
// .
// .
class D : public B1
{
    int d1, d2;
public:
    D(int i, int j) : d1(i), B1(d1) {d2 = j;}
    // d1 is not initialized in call B1::B1(d1)
};
```

Copying Class Objects

You can copy one class object to another object of the same type by either assignment or initialization.

Copy by assignment is implemented with an assignment operator function. If you do not define the assignment operator, it is defined as *memberwise assignment*.

Copy by initialization is implemented with a copy constructor. If you do not define a copy constructor, it is defined as *memberwise initialization* of members of its class.

Memberwise assignment and memberwise initialization mean that, if a class has a member that is a class object, the assignment operator and copy constructor of that class object are used to implement assignment and initialization of the member.

Copying Class Objects

Copy Restrictions

A default assignment operator cannot be generated for a class that has:

- A nonstatic constant or reference data member
- A nonstatic data member or base class whose assignment operator is not accessible
- A nonstatic data member or base class with no assignment operator and for which a default assignment operator cannot be generated.

A default copy constructor cannot be generated for a class that has:

- A nonstatic data member or base class whose copy constructor is not accessible
- A nonstatic data member or base class with no copy constructor and for which a default copy constructor cannot be generated.

Copy by Assignment

If you do not define an assignment operator and one is required, a default assignment operator is defined. If you do not define an assignment operator and one is not required, a default assignment operator is declared but not defined. If an assignment operator that takes a single argument of a class type exists for a class, a default assignment operator is not generated.

Copy by assignment is used only in assignment.

You can define an assignment operator for a class with a single argument that is a constant reference to that class type, only if all its base classes and members have assignment operators that accept constant arguments.

For example:

```
class B1
{
public:
    B1& operator=(const B1&);
};
class D: public B1
{
public:
    D& operator=(const D&);
};
D& D::operator=(const D& dobj) {D dobj2 = dobj;
                               return dobj2;}
```

Otherwise, you can define an assignment operator for a class with a single argument that is a reference to that class type. For example:

Copying Class Objects

```
class Z
{
public:
    Z& operator=( Z&);
};
Z& Z::operator=(Z& zobj) {Z zobj2 = zobj;
                        return zobj2;}
```

The default assignment operator for a class is a public class member. The return type is a reference to the class type it is a member of.

For more information on standard C and C++ assignment operators, see “Assignment Expressions” on page 126. For more information on assignment operator functions, see “Overloaded Assignment” on page 287.

Copy by Initialization

You can define a copy constructor for a class. If you do not define a copy constructor and one is required, a default copy constructor is defined. If you do not define a copy constructor, and one is not required, a default copy constructor is declared but not defined. If a class has a copy constructor defined, a default copy constructor is not generated.

Copy by initialization is used only in initialization.

You can define a copy constructor for a class with a single argument that is a constant reference to a class type only if all its base classes and members have copy constructors that accept constant arguments. For example:

```
class B1
{
public:
    B1(const B1&) { /* ... */ }
};

class D: public B1
{
public:
    D(const D&);
};
D::D(const D& dobj):B1(dobj) { /* ... */ }
```

Otherwise, you can define a copy constructor with a single reference to a class type argument. For example:

```
class Z
{
public:
    Z(Z&);
};
Z::Z(Z&) { /* ...*/ }
```

Copying Class Objects

The default copy constructor for a class is a public class member. For more information on copy constructors, see “Constructors” on page 294, and “Initialization by Constructor” on page 306.

Chapter 13. Inheritance

In C++, you can create classes from existing classes using the object-oriented programming technique called *inheritance*. Inheritance allows you to define an *is a* relationship between classes. When members are inherited, they can be used as if they are members of the class that inherits them.

This chapter discusses:

Inheritance Overview	315
Derivation	319
Inherited Member Access	324
Multiple Inheritance	332
Virtual Functions	338
Abstract Classes	343

Related Information

- Chapter 6, “Functions” on page 137
- Chapter 9, “Classes” on page 239
- Chapter 10, “Class Members and Friends” on page 253

Inheritance Overview

C++ implements inheritance through the mechanism of *derivation*. Derivation allows you to reuse code by creating new classes, called *derived classes*, that inherit properties from one or more existing classes, called *base classes*. A derived class inherits the properties, including data and function members, of its base class. You can also add new data members and member functions to the derived class. You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

Suppose you have defined a `shape` class to describe and operate on geometric shapes. Now suppose you want to define a `circle` class. Because you have existing code that operates on the `shape` class, you can use inheritance to create the `circle` class. You can redefine operations in the derived `circle` class that were originally defined in the `shape` base class. When you manipulate an object of the `circle` class, these redefined function implementations are used.

Inheritance

For example:

```
class shape
{
    char* name;
    int xpoint, ypoint;
public:
    virtual void rotate(int);
    virtual void draw();
    void display() const;
};

class circle: public shape    // derive class circle from
class shape
{
    int xorigin, yorigin;
    int radius;
public:
    void rotate(int);
    void draw();
    void display() const;
};
// .
// .
// .
```

In the above example, class `circle` inherits the data members `name`, `xpoint` and `ypoint`, as well as the member functions `display()`, `rotate()`, and `draw()` from class `shape`. Because the member functions `rotate()` and `draw()` are declared in class `shape` with the keyword **virtual**, you can provide an alternative implementation for them in class `circle`.

You can also provide an alternative implementation for the nonvirtual member function `display()` in class `circle`. When you manipulate an argument of type `circle` using a pointer to `shape`, and call a virtual member function, the member function defined in the derived class overrides the base-class member function. A similar call to a nonvirtual member function will call the member function defined in the base class. In addition to inheriting the members of class `shape`, class `circle` has declared its own data members, `xorigin`, `yorigin`, and `radius`.

The key difference between virtual and nonvirtual member functions is that, when you treat the `circle` class as if it were a `shape`, the implementations of the virtual functions `rotate()` and `draw()` defined in class `circle` are used, rather than those originally defined in class `shape`. Because `display()` is a nonvirtual member function, the original implementation of `display()` defined in class `shape` is used.

Multiple Inheritance

Multiple inheritance allows you to create a derived class that inherits properties from more than one base class.

For example, in addition to the `shape` class, described above, you could also have a `symbol` class. Because a `circle` is both a `shape` and a `symbol`, you can use multiple inheritance to reflect this relationship. If the `circle` class is derived from both the `shape` and `symbol` classes, the `circle` class inherits properties from both classes.

```
class symbol
{
    char* language;
    char letter;
    int number;
public:
    virtual void write();
    virtual void meaning();
};
class shape
{
    char* name;
    int xpoint, ypoint;
public:
    virtual void rotate(int);
    virtual void draw();
    void display() const;
};
class circle: public symbol, public shape
{
    int xorigin, yorigin;
    int radius;
public:
    void rotate(int);
    void draw ();
    void write();
    void meaning();
    void display() const;
};
// .
// .
// .
```

In the above example, class `circle` inherits the members `name`, `xpoint`, `ypoint`, `display()`, `rotate()`, and `draw()` from class `shape` and also inherits the members `language`, `letter`, `number`, `write()`, and `meaning()` from class `symbol`.

Because a derived class inherits members from all its base classes, ambiguities can result. For example, if two base classes have a member with the same name, the derived class cannot implicitly differentiate between the two members. Note that, when you are using multiple inheritance, the access to names of base classes may be ambiguous.

Inheritance

The Inheritance Design Process

Multiple inheritance allows you to have more than one base class for a single derived class. You can create an interconnected *inheritance graph* of inherited classes by using derived classes as base classes for other derived classes. You can build an inheritance graph through the process of specialization, in which derived classes are more specialized than their base classes. You can also work in the reverse direction and build an inheritance graph through generalization. If you have a number of related classes that share a group of properties, you can generalize and build a base class to embody them. The group of related classes becomes the derived classes of the new base class.

Direct and Indirect Base Classes

A *direct base class* is a base class that appears directly as a base specifier in the declaration of its derived class. A direct base class is analogous to a parent in a hierarchical graph. In the above example, both `shape` and `symbol` are direct base classes of class `circle`.

An *indirect base class* is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through one of its base classes. An indirect base class is analogous to a grandparent or great grandparent or great-great grandparent in a hierarchical graph. For a given class, all base classes that are not direct base classes are indirect base classes.

Polymorphism

Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

- Overloaded functions are statically bound at compile time, as discussed in “Overloading Functions” on page 277.
- C++ provides virtual functions. A *virtual function* is a function that can be called for a number of different user-defined types that are related through derivation. Virtual functions are bound dynamically at run time.

Typically, a base class has several derived classes, each requiring its own customized version of a particular operation. It is difficult for a base class to implement member functions that are useful for all of its derived classes. A base class would have to determine which derived class an object belonged to before it could execute the applicable code for that object. When a virtual function is called, the compiler executes the function implementation associated with the object that the function is called for. The implementation of the base class is only a default that is used when the derived class does not contain its own implementation.

Derivation

Inheritance is implemented in C++ through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

In the declaration of a derived class, you list the base classes of the derived class. The derived class inherits its members from these base classes. All classes that appear in the list of base classes must be previously defined classes.

Incompletely declared classes are not allowed in base lists.

For example:

```
class X; // incomplete declaration of class X
class Y: public X    // error
{
//      .
//      .
//      .
};
```

When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class.

Derivation

For example:

```
// This example illustrates references  
// to base class members.
```

```
class base  
{  
public:  
    int a,b;  
};  
class derived : public base  
{  
public:  
    int c;  
};  
void main()  
{  
    derived d;  
    d.a = 1;    // base::a  
    d.b = 2;    // base::b  
    d.c = 3;    // derived::c  
}
```

The derived class can also add new class members and redefine existing base class members. In the above example, the two inherited members, a and b, of the derived class d, in addition to the derived class member c, are assigned values. If you redefine base class members in the derived class, you can still refer to the base class members by using the `::` (scope resolution) operator.

For example:

Derivation

```
// This example illustrates references to base class
// members with the scope resolution (::) operator.

#include <iostream.h>
class base
{
public:
    char* name;
    void display(char* i) {cout << i << endl;}
};
class derived : public base
{
public:
    char* name;
    void display(char* i){cout << i << endl;}
};
void main()
{
    derived d;                // create derived class object
    d.name = "Derived Class"; // assignment to derived::name
    d.base::name = "Base Class"; // assignment to base::name

    // call derived::display(derived::name)
    d.display(d.name);

    // call base::display(base::name)
    d.base::display(d.base::name);
}
```

The :: (scope resolution) operator is described on page 97.

You can manipulate a derived class object as if it were a base class object. You can use a pointer or a reference to a derived class object in place of a pointer or reference to its base class. For example, you can pass a pointer or reference to a derived class object D to a function expecting a pointer or reference to the base class of D. You do not need to use an explicit cast to achieve this; a standard conversion is performed. You can implicitly convert a pointer to a derived class to point to a base class. You can also implicitly convert a reference to a derived class to a reference to a base class.

In the following example, d, a pointer to a derived class object is assigned to bptr, a pointer to a base class object. A call is made to display() using bptr. Even though bptr has a type of pointer to base, in the body of display() the name member of derived is manipulated:

Derivation

```
// This example illustrates how to make a pointer
// to a derived class point to a base class.

#include <iostream.h>
class base
{
public:
    char* name;
    void display(char* i) {cout << i << endl;}
};
class derived : public base
{
public:
    char* name;
    void display(char* i){cout << i << endl;}
};
void main()
{
    derived d;

    // standard conversion from derived* to base*
    base* bptr = &d;

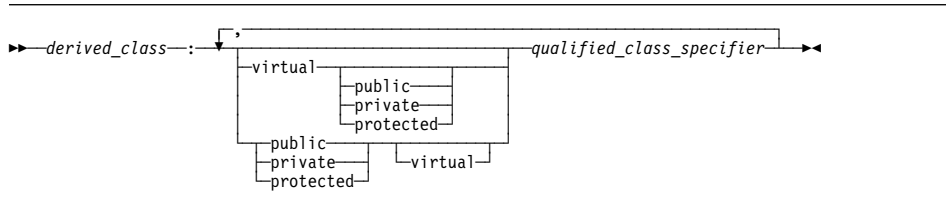
    // call base::display(base::name)
    bptr->display(bptr->name);
}
```

The reverse case is not allowed. You cannot implicitly convert a pointer or a reference to a base class object to a pointer or reference to a derived class.

If a member of a derived class and a member of a base class have the same name, the base class member is hidden in the derived class. If a member of a derived class has the same name as a base class, the base class name is hidden in the derived class. In both cases, the name of the derived class member is called the *dominant* name.

Syntax of a Derived Class Declaration

The syntax for the list of base classes is:



The *qualified class specifier* must be a class that has been previously declared in a class declaration as described in “Class Names” on page 242.

An *access specifier* is one of:

- public**
- private**
- protected**

Access specifiers are described on page 271.

The **virtual** keyword can be used to declare virtual base classes. For more information, see “Virtual Base Classes” on page 333.

The following example shows the declaration of the derived class D and the base classes V, B1, and B2. The class B1 is both a base class and a derived class because it is derived from class V and is a base class for D.

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

Inherited Member Access

Inherited Member Access

Access specifiers, as described on page 271, control the level of access to noninherited class members. The access for an inherited member is controlled in three ways:

- When you declare a member in a base class, you can specify a level of access using the keywords **public**, **private**, and **protected**.
- When you derive a class, you can specify the access level for the base class in the base list.
- You can also restore the access level of inherited members. See “Derivation Access of Base Classes” on page 325 for an example.

Resolution of member names does not depend on the level of access associated with each class member.

Consider the following example:

```
class A {
    private:
        int a;
};
class B {
    public:
        int a;
};
class C : public A, public B {
    void f() { a = 0; } // ambiguous - is it A::a or B::a?
};
```

In this example, class A has a private member `a`, and class B has a public member `a`. Class C is derived from both A and B. C does not have access to `A::a`, but `a` in the body of `f()` can still resolve to either `A::a` or `B::a`. For this reason, `a` is ambiguous in the body of `f()`.

Protected Members

If a class is derived publicly from a base class, a protected static base class member can be accessed by members and friends of any classes derived from that base class. A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:

- A pointer to a directly or indirectly derived class
- A reference to a directly or indirectly derived class
- An object of a directly or indirectly derived class

If a class is derived privately from a base class, all protected base class members become private members of the derived class.

The access specifier **protected** is also described in “Access Specifiers” on page 271.

Inherited Member Access

Derivation Access of Base Classes

When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class, but allows the derived class to restore the access attributes of the members of a base class.

You can derive classes using any of the three access specifiers:

- In a **public** base class, public and protected members of the base class remain public and protected members of the derived class.
- In a **private** base class, public and protected members of the base class become private members of the derived class.
- In a **protected** base class, public and protected members of the base class are protected members of the derived class.

In all cases, private members of the base class remain private. Private members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them.

In the following example, class `d` is derived publicly from class `b`. Class `b` is declared a public base class by this declaration.

```
class b
{
//    .
//    .
//    .
};
class d : public b // public derivation
{
//    .
//    .
//    .
};
```

You can use both a structure and a class as base classes in the base list of a derived class declaration. If the base class is declared with the keyword **class**, its default access specifier in the base list of a derived class is **private**. If the base class is declared with the keyword **struct**, its default access specifier in the base list of a derived class is **public**.

In the following example, private derivation is used by default because no access specifier is used in the base list:

Inherited Member Access

```
struct bb
{
//      .
//      .
//      .
};
class dd : bb // private derivation
{
//      .
//      .
//      .
};
```

Members and friends of a class can implicitly convert a pointer to an object of that class to a pointer to either:

- A direct private base class
- A protected base class (either direct or indirect)

Access Declarations

You can restore access to members of a base class using an *access declaration*. It allows you to change the access of a public member of a private or protected base class back to public. You can also change the access of a protected member of a private base class back to protected. Access is *adjusted* by using the base class member qualified name in the public or protected declarations of the derived class.

You only use access declarations to restore base class access. You cannot change the access to a member to give it more access than it was originally declared with. You cannot change the access of a private member to public or to protected. You cannot change the access of a protected member to public.

An access declaration cannot be used to restrict access to a member that is accessible in a base class.

It is redundant to use an access declaration to change the access to a public member of a public base class to public, or to change the access to a protected member of a protected base class to protected.

Inherited Member Access

In the following example, the member `b` of the base class `base` is declared `public` in its base class declaration. Class `derived` is derived privately from class `base`. The access declaration in the `public` section of class `derived` restores the access level of the member `b` back to **public**.

```
// This example illustrates using access declarations
// to restore base class access.

#include <iostream.h>
class base
{
    char a;
public:
    char c, b;
    void bprint();
};

class derived: private base
{
    char d;
public:
    char e;
    base::b;          // restore access to b in derived
    void dprint();
    derived(char ch) { base::b = ch; }
};

void print(derived& d)
{
    cout << " Here is d " << d.b << endl;
}

void main()
{
    derived obj('c');
    print(obj);
}
```

The external function `print(derived&)` can use the member `b` of `base` because the access of `b` has been restored to `public`. The external function `print(derived&)` can also use the members `e` and `dprint()` because they are declared with the keyword **public** in the derived class. The derived class member `dprint()` can use the members of its own class, `d` and `e`, in addition to the inherited members, `b`, `c`, and `bprint()`, that are declared with the keyword **public** in the base class. The base class member `bprint()` can use all the members of its own class, `a`, `b`, and `c`.

Access declarations can only be used to adjust the access of a member of a base class. The base class that an access declaration appears in can be directly or indirectly inherited by the derived class.

Inherited Member Access

You can also use an access declaration in a nested class. For example:

```
class B
{
public:
    class N          // nested class
    {
    public:
        int i;      // public member
    };
};
class D: private B::N // derive privately
{
public:
    B::N::i;        // restores access to public
};
```

You cannot adjust the access to a base class member if a member with the same name exists in a class derived from that base class.

You cannot convert a pointer to a derived class object to a pointer to a base class object if the base class is private or protected. For example:

```
class B { /* ... */ };
class D : private B { /* ... */ };    // private base class

void main ()
{
    D d;
    B* ptr;
    ptr = &d;    // error
}
```

If you use an access declaration to adjust the access to an overloaded function, the access is adjusted for all functions with that name in the base class.

Inherited Member Access

Access Resolution

Access resolution is the process by which the accessibility of a particular class member is determined. Accessibility is dependent on the context. For example, a class member can be accessible in a member function but inaccessible at file scope. The following describes the access resolution procedure used by the compiler.

In general, two *scopes* must be established before access resolution is applied. These scopes reduce an expression or declaration into a simplified construct to which the access rules are applied. Access rules are described in “Member Access” on page 270. These scopes are:

Call scope The scope that encloses the expression or declaration that uses the class member.

Reference scope The scope that identifies the class.

For example, in the following code:

```
// This example illustrates access resolution.
```

```
class B { public: int member; };            // declaration
class A : B {}                              // declaration
void main()
{
    A aobject;                              // declaration
    aobject.member = 10;                  // expression
}
```

the reference scope for member is the type of aobject, that is class type A.

Reference scope is chosen by simplifying the expression (or declaration) containing the member. An expression can be thought of as being reduced to a simple expression of the form obj.member where obj is the reference scope. Reference scope is selected as follows:

- If the member is qualified with . (dot) or -> (arrow), the reference scope is the type of the object that is immediately to the left of the . or -> operator closest to the member. Unqualified members are treated as if they are qualified with **this->**.
- If the member is a type member or a static member and is qualified with :: (the scope resolution operator), the reference scope is the type immediately to the left of the :: operator closest to the member.
- Otherwise, the reference scope is the call scope.

The call scope and the reference scope determine the accessibility of a class member. Once these scopes are resolved, the *effective access* of the member is determined. Effective access is the access of the member as it is seen from the reference scope. It is determined by taking the original access of the member in its scope as the effective access and changing it as the class hierarchy is traversed from the member's class to the reference scope. Effective access is altered as the class hierarchy is traversed for each derivation by the following:

Inherited Member Access

- The derivation access of a base class (see “Derivation Access of Base Classes” on page 325)
- Access declarations that are applied to the members (see “Access Declarations” on page 326)
- Friendships that are granted to the call scope (see “Member Access” on page 270)

After effective access is determined for a member, the access rules are applied as if the effective access were the original access of the member. A member is only accessible if the access rules say that it is.

The following example demonstrates the access resolution procedure.

```
class A
{
public:
    int a;
};
class B : private A
{
    friend void f (B*);
};
void f(B* b)
{
    b->a = 10; // is 'a' accessible to f(B*) ?
}
//      .
//      .
//      .
```

The following steps occur to determine the accessibility of `A::a` in `f(B*)`:

1. The call scope and reference scope of the expression `b->a` are determined:
 - a. The call scope is the function `f(B*)`.
 - b. The reference scope is class `B`.
2. The effective access of member `a` is determined:
 - a. Because the original access of the member `a` is public in class `A`, the initial effective access of `a` is public.
 - b. Because `B` inherits from `A` privately, the effective access of `a` inside class `B` is private.
 - c. Because class `B` is the reference scope, the effective access procedure stops here. The effective access of `a` is private.
3. The access rules are applied. The rules state that a private member can be accessed by a friend or a member of the member's class. Because `f(B*)` is a friend of class `B`, `f(B*)` can access the private member `a`.

Inherited Member Access

Access Summary

The following example demonstrates inherited member access rules.

// This example illustrates inherited member access rules.

```
class B
{
    int a;
public:
    int b,c;
    void f(int) {}
protected:
    int d;
    void g(int) {}
};

class D1 : public B
{
    int a;
public:
    int b;
    void h(int i )
    {
        g(i);           // valid, protected B::g(int)
        B::b = 10;      // valid, B::b (not local b)
        d = 5 ;        // valid, protected B::d
    }
};

class D2 : private B
{
    int e;
public:
    B::c;               // modify access to B::c
    void h(int i) { d = 5; } // valid,protected B::d
};
```

Multiple Inheritance

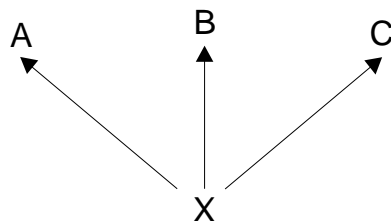
```
void main( )
{
    int i= 1;    // declare and initialize local variable
    D1 d1;      // create object of class d1
    D2 d2;      // create object of class d2

    d1.a = 5;   // error, D1::a is private in class D1
    d2.b = 10;  // error, B::b is inherited private to
                // derived class D2
    d2.c = 5;   // valid, modified access from private to public
    d2.B::c = 5; // valid, public B::c
    d1.c = 5;   // valid, B::c is inherited publicly
    d1.d = 5;   // error, B::d is protected in base class
    d2.e = 10;  // error, private D2::e
    d1.g(i);    // error, g(int) is protected in base class
    d1.h(i);    // valid
    d2.h(i);    // valid
}
```

Multiple Inheritance

You can derive a class from more than one base class. Deriving a class from more than one direct base class is called *multiple inheritance*.

In the following example, classes A, B, and C are direct base classes for the derived class X:



```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

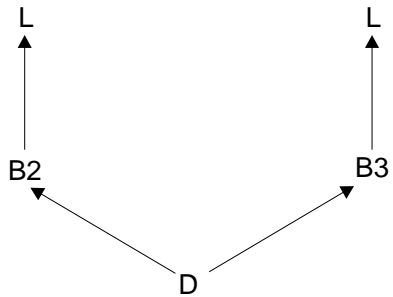
The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors. For more information, see “Initialization by Constructor” on page 306.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ }; // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

Multiple Inheritance

However, a derived class can inherit an indirect base class more than once, as shown in the following example:



```
class L { /* ... */ }; // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two objects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

```
B2::L
```

or

```
B3::L.
```

You can also avoid this ambiguity by using the base specifier **virtual** to declare a base class.

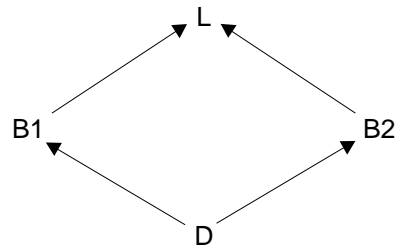
Virtual Base Classes

If you have an inheritance graph in which two or more derived classes have a common base class, you can use a virtual base class to ensure that the two classes share a single instance of the base class.

In the following example, an object of class D has two distinct objects of class L, one through class B1 and another through class B2. You can use the keyword **virtual** in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one class L, shared by class B1 and class B2, exists.

Multiple Inheritance

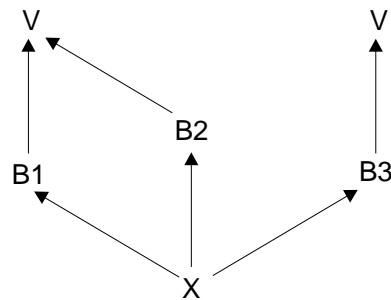
For example:



```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword **virtual** in this example ensures that an object of class D inherits only one object of class L.

A derived class can have both virtual and nonvirtual base classes. For example:



```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class D : public B1, public B2, public B3 { /* ... */ };
};
```

In the above example, class D has two objects of class V, one that is shared by classes B1 and B2 and one through class B3.

Multiple Inheritance

Multiple Access

In an inheritance graph containing virtual base classes, a name that can be reached through more than one path is accessed through the path that gives the most access.

For example:

```
class L { public: void f(); };
class B1 : private virtual L { /* ... */ };
class B2 : public virtual L { /* ... */ };
class D : public B1, public B2
{
public:
    void f() {L::f();} // L::f() is accessed through B2
                       // and is public
};
```

In the above example, the function `f()` is accessed through class `B2`. Because class `B2` is inherited publicly and class `B1` is inherited privately, class `B2` offers more access.

Accessible Base Classes

An *accessible base class* is a publicly derived base class that is neither hidden nor ambiguous in the inheritance hierarchy.

Ambiguous Base Classes

When you derive classes, ambiguities can result if base and derived classes have members with the same names. Access to a base class member is ambiguous if you use a name or qualified name that does not refer to a unique function, object, type, or enumerator. The declaration of a member with an ambiguous name in a derived class is not an error. The ambiguity is only flagged as an error if you use the ambiguous member name.

Multiple Inheritance

For example, if two base classes have a member of the same name, an attempt to access the member by the derived class is ambiguous. You can resolve ambiguity by qualifying a member with its class name using the `::` (scope resolution) operator.

// This example illustrates ambiguous base classes.

```
class B1
{
public:
    int i;
    int j;
    int g( );
};
class B2
{
public:
    int j;
    int g( );
};
// .
// .
// .
class D : public B1, public B2
{
public:
    int i;
};
void main ()
{
    D dobj;
    D *dptr = &dobj;
    dptr -> i = 5;           // valid, D::i
    dptr -> j = 10;         // error, ambiguous reference to j
    dptr->B1::j = 10;       // valid, B1::j
    dobj.g( );             // error, ambiguous reference to g( )
    dobj.B2::g( );         // valid, B2::g( )
}
```

The compiler checks for ambiguities at compile time. Because ambiguity checking occurs before access control or type checking, ambiguities may result even if only one of several members with the same name is accessible from the derived class.

Multiple Inheritance

Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object. For example:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
void main ()
{
    Z z;
    X* xptr = &z;      // valid
    Y* yptr = &z;      // valid
    W* wptr = &z;      // error, ambiguous reference to class W
                       // X's W or Y's W ?
}
```

You can use virtual base classes to avoid ambiguous reference. For example:

```
class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
void main ()
{
    Z z;
    X* xptr = &z;      // valid
    Y* yptr = &z;      // valid
    W* wptr = &z;      // valid, W is virtual therefore only one
                       // W subobject exists
}
```

For more information, see “Virtual Base Classes” on page 333.

Virtual Functions

Virtual Functions

In C++, dynamic binding is supported by the mechanism of virtual functions. Virtual functions must be members of a class. Use virtual functions when you expect a class to be used as a base class in a derivation and when the implementation of the function may be overridden in the derived class. You can declare a member function with the keyword **virtual** in its class declaration. For example:

```
class B
{
    int a,b,c;
public:
    virtual int f();
};
// .
// .
// .
```

You can reimplement a virtual member function, like any member function, in any derived class. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called. If a virtual member function is called for a derived class object and the function is redefined in the derived class, the definition in the derived class is executed. In this case, the redefined derived class function is said to *override* the base class function. This occurs even if the access to the function is through a pointer or reference to the base class. If you call a virtual function with a pointer that has base class type but points to a derived class object, the member function of the derived class is called. However, if you call a nonvirtual function with a pointer that has base class type, the member function of the base class is called regardless of whether the pointer points to a derived class object.

For example:

Virtual Functions

```
class B
{
public:
    virtual int f();
    virtual int g();
    int h();
};
class D : public B
{
public:
    int f();
    int g(char*);    // hides B::g()
    int h();
};
// .
// .
// .
void main ()
{
    D d;
    B* bptr = &d;

    bptr->f();        // calls D::f() because f() is virtual
    bptr->h();        // calls B::h() because h() is nonvirtual
    bptr->g();        // calls B::g()
    d.g();           // error, wrong number and type of arguments
    d.g("string");  // calls D::g(char*)
}
```

If the argument types or the number of arguments of the two functions are different, the functions are considered different, and the function in the derived class does not override the function in the base class. The function in the derived class hides the function in the base class.

The return type of an overriding virtual function can differ from the return type of the overridden virtual function provided that:

- The overridden function returns a pointer or a reference to a class T
- AND
- The overriding virtual function returns a pointer or a reference to a class derived from T.

An error does result when a virtual function that returns D^* overrides a virtual function that returns B^* where B is an ambiguous base class of D. The reason is that two or more instances of class B will exist within class D, and the compiler will not know which base B to return. For more information, see "Function Return Values" on page 159.

A virtual function cannot be global or static because, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which

Virtual Functions

implementation of the function is called. You can declare a virtual function to be a friend of another class. Friends are described on page 273.

If a function is declared virtual in its base class, it can still be accessed directly using the `::` (scope resolution) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not redefine a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A base class containing one or more pure virtual member functions is called an *abstract class*. For more information, see “Abstract Classes” on page 343.

Ambiguous Virtual Function Calls

It is an error to override one virtual function with two or more ambiguous virtual functions. This can happen in a derived class that inherits from two nonvirtual bases that are derived from a virtual base class.

For example:

```
class V
{
public:
    virtual void f() { /* ... */ };
};
class A : virtual public V
{
    void f() { /* ... */ };
};
class B : virtual public V
{
    void f() { /* ... */ };
};
class D : public B, public A { /* ... */ }; // error
void main ()
{
    D d;
    V* vptr = &d;
    vptr->f();           // which f(), A::f() or B::f()?
}
```

In class A, only `A::f()` will override `V::f()`. Similarly, in class B, only `B::f()` will override `V::f()`. However, in class D, both `A::f()` and `B::f()` will try to override `V::f()`. This attempt is not allowed because it is not possible to decide which function to call if a D object is referenced with a pointer to class V, as shown in the above

Virtual Functions

example. Because only one function can override a virtual function, the compiler flags this situation as an error.

A special case occurs when the ambiguous overriding virtual functions come from separate instances of the same class type. In the following example, there are two objects (instances) of class L. There are two data members L::count, one in class A and one in class B. If the declaration of class D is allowed, incrementing L::count in a call to L::f() with a pointer to class V is ambiguous.

```
class V
{
public:
    virtual void f();
};
class L : virtual public V
{
    int count;
    void f();
};
void L::f() {++count;}
class A : public L
{ /* ... */ };
class B : public L
{ /* ... */ };
class D : public A, public B { /* ... */ }; // error
void main ()
{
    D d;
    V* vptr = &d;
    vptr->f();
}
```

In the above example, the function L::f() is expecting a pointer to an L object; that is, the **this** pointer for class L, as its first implicit argument. Because there are two objects of class L in a D object, there are two **this** pointers that could be passed to L::f(). Because the compiler cannot decide which **this** pointer to pass to L::f(), the declaration of class D is flagged as an error.

Virtual Functions

Virtual Function Access

The access for a virtual function is specified when it is declared. The access rules for a virtual function are not affected by the access rules for the function that later overrides the virtual function. In general, the access of the overriding member function is not known.

If a virtual function is called with a pointer or reference to a class object, the type of the class object is not used to determine the access of the virtual function. Instead, the type of the pointer or reference to the class object is used.

In the following example, when the function `f()` is called using a pointer having type `B*`, `bptr` is used to determine the access to the function `f()`. Although the definition of `f()` defined in class `D` is executed, the access of the member function `f()` in class `B` is used. When the function `f()` is called using a pointer having type `D*`, `dptr` is used to determine the access to the function `f()`. This call produces an error because `f()` is declared private in class `D`.

```
class B
{
public:
    virtual void f();
};
class D : public B
{
private:
    void f();
};
// .
// .
// .
void main ()
{
    D dobj;
    B *bptr = &dobj;
    D *dptr = &dobj;
    bptr->f();    // valid, virtual B::f() is public,
                // D::f() is called
    dptr->f();    // error, D::f() is private
}
```

Abstract Classes

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. Pure virtual functions are inherited. You can declare a function to be pure by using a pure specifier in the declaration of the member function in the class declaration.

For example:

```
class AB          // abstract class
{
public:
    virtual void f()= 0; // pure virtual member function
};
class D: public AB
{
public:
    void f();
};
//      .
//      .
//      .
void main ()
{
    D d;
    d.f() ;    // calls D::f()
    AB ab;     // error, cannot create an object of an
               // abstract class type
}
```

A function that is declared pure typically has no definition and cannot be executed. Attempting to call a pure virtual function that has no implementation is undefined; however, such a call does not cause an error. No objects of an abstract class can be created, as shown in the above example.

Note: Because destructors are not inherited, a virtual destructor that is declared pure must have a definition.

Virtual member functions are inherited. If a base class contains a pure virtual member function and a class derived from that base class does not redefine that pure virtual member function, the derived class itself is an abstract class. Any attempt to create an object of the derived class type produces an error.

Abstract Classes

For example:

```
class AB // abstract class
{
public:
    virtual void f()= 0; // pure virtual member function
};
class D2: public AB
{
    int a,b,c;
public:
    void g();
};
//      .
//      .
//      .
void main ()
{
    D2 d;
    // error, cannot declare an object of abstract class D2
}
```

To avoid the error in the above example, provide a declaration of `D2::f()`.

You cannot use an abstract class as the type of an explicit conversion, as an argument type, or as the return type for a function. You can declare a pointer or reference to an abstract class.

Chapter 14. Templates

This chapter describes the C++ template facility. A *template* specifies how an individual class, function, or static data member can be constructed by providing a blueprint description of classes or functions within the template.

Unlike an ordinary class or function definition, a template definition contains the **template** keyword, and uses a *type argument*, instead of a type, in one or more of the constructs used to define the class or function template. Individual classes or functions can then be generated simply by specifying the template name and by naming the type for the particular class or function as the type argument of the template. You can use templates to define a family of types or functions.

This chapter discusses:

Templates Overview	346
Structuring Your Program Using Templates	349
Class Templates	351
Function Templates	356
Differences between Class and Function Templates	360
Member Function Templates	361
Friends and Templates	364
Static Data Members and Templates	365

Related Information

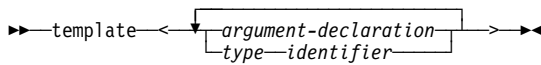
- Chapter 6, “Functions” on page 137
- Chapter 9, “Classes” on page 239
- “Type Specifiers” on page 45
- “define” on page 216
- “implementation” on page 220

See the *VisualAge for C++ for AS/400 C++ Programming Guide* for programming hints on using templates in C++ programs.

Templates Overview

Templates Overview

The syntax for a template is:



The *declaration* in a template declaration must define or declare one of the following:

- A class
- A function
- A static member of a template class

The *identifier* of a *type* is defined to be a *type_name* in the scope of the template declaration. A template declaration can appear as a global declaration only.

The template arguments (within the < and > delimiters) specify the types and the constants within the template that must be specified when the template is instantiated.

Given the following template:

```
template<class L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

The following table shows what the classes `Key<int>`, `Key<char*>`, and `Key<mytype>` look like:

<code>class Key<int> i;</code>	<code>class Key<char*> c;</code>	<code>class Key<mytype> m;</code>
<pre>class Key<int> { int k; int * kptr; int length; public: Key(int); // ... };</pre>	<pre>class Key<char*> { char* k; char** kptr; int length; public: Key(char*); // ... };</pre>	<pre>class Key<mytype> { mytype k; mytype* kptr; int length; public: Key(mytype); // ... };</pre>

Templates Overview

The declarations create the following objects:

- `i` of type `Key<int>`
- `c` of type `Key<char*>`
- `m` of type `Key<mytype>`

Note that these three classes have different names. The types contained within the angle braces are not arguments to the class names, but part of the class names themselves. `Key<int>` and `Key<char*>` are class names. Within the context of the example, a class called `Key` (with no template argument list) is undefined.

Default initializers are permitted in template arguments, under the following conditions:

- They can only be applied to nontype template arguments.
- Like functions, they can only be applied to trailing arguments.
- Subsequent template declarations can add default initializers but cannot redefine existing default initializers.
- They can only be applied to class template declarations, not to function template declarations.

Note: A template that defines a member function of a class template is treated as a function template. Such a template cannot have default initializers.

Templates Overview

The following example shows a valid template declaration with default initializers:

```
// This example shows a template declaration
// with default initializers.

#include <stdio.h>

template <class T, int i=1> class X
{
public:
    T s;
    X(int j=4);
    int val(T&)
    {
        return i;
    };
};

template <class T, int i> X<T,i>::X(int j):s(i){
    printf("i=%d   j=%d\n",i,j);
}

void main()
{
    X<int>   myX(2);
    X<int,3> myX2(4);
}
```

Structuring Your Program Using Templates

Structuring Your Program Using Templates

You can structure your program three ways using templates:

1. Include the function template definition (both the .h and .c files) in all files that may reference the corresponding template functions.
2. Include the function template declaration (the .h file only) in all files that may reference the corresponding template functions, but include the function definition (both the .h and .c files) in one file only.
3. Include the declaration of the function templates in a header file and the definition in a source file that has the same name. When you include the header file in your source, the compiler automatically generates the template functions.

The following examples use two files to illustrate all three methods:

File stack.h

```
#ifndef _STACK_TPL_H
#define _STACK_TPL_H

template<class T>
class stack
{
private:
    T* v;
    T* p;
    int sz;

public:
    stack( int );
    ~stack();
    void push( T );
};
#endif
```


Structuring Your Program Using Templates

File `stackdef.h`

```
#include "stack.h"

template<class T> stack<T>::stack( int s )
{
    v = p = new T[sz=s];
}

template<class T> stack<T>::~~stack()
{
    delete [] v;
}

template<class T> void stack<T>::push( T a )
{
    *p++ = a;
}
```

To instantiate a stack of 50 ints, you would declare the following in each source file that requires it:

```
stack<int> intStack(50);
```

For method 1, each source file using the template should include both `stack.h` and `stackdef.h`.

For method 2, every source file should include `stack.h`, but only one of the files needs to include `stackdef.h`.

Class Templates

The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

Note the distinction between the terms *class template* and *template class*:

Class template is a template used to generate template classes. A class template can be only a declaration, or it can be a definition of the class.

Template class is an instance of a class template.

A template definition is identical to any valid class definition that the template might generate, except for the following:

- The class template definition is preceded by

```
template < template-argument-list >
```

where *template-argument-list* can include zero or more *type-arguments* and zero or more *argument-declarations*. The *template-argument-list* must contain at least one argument.

- Types, variables, constants and objects within the class template can be declared with arguments of user-defined type as well as with explicit types (for example, **int** or **char**).
- The *template-argument-list* can include *argument-declarations* (for example, `int a` or `char* b`), which are generally used to define constant values within the created class.

A class template can declare a class without defining it by using an elaborated type specifier. For example:

```
template <class L,class T> class key;
```

This reserves the name as a class template name. All template declarations for a class template must have the same types and number of template arguments. Only one template declaration containing the class definition is allowed.

You can instantiate the class template by declaring a template class. If the definitions of the member functions of the template class are not inlined, then you have to define them. When you instantiate a template class, its argument list must match the argument list in the class template declaration.

Note: When you have nested template argument lists, you must have a separating space between the > at the end of the inner list and the one at the end of the outer list. Otherwise, there is an ambiguity between the output operator >> and two template list delimiters >.

Class Templates

```
template <class L,class T> class key
{
//      .
//      .
//      .
};
template <class L> class vector
{
//      .
//      .
//      .
};

void main ()
{
class key <int, vector<int> >; // instantiate template
}
```

Objects and functions of individual template classes can be accessed by any of the techniques used to access ordinary class member objects and functions. Given a class template:

```
template<class T> class vehicle
{
public:
    vehicle() { /* ... */ } // constructor
    ~vehicle() {}; // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

and the declaration:

```
vehicle<char> bicycle; // instantiates the template
```

the constructor, the constructed object, and the member function drive() can be accessed with any of the following (assuming the standard header file <string.h> is included in the program file):

constructor	vehicle<char> bicycle; // constructor called automatically // object bicycle created
object bicycle	strcpy (bicycle.kind, "10 speed"); bicycle.kind[0] = '2';
function drive()	char* n = bicycle.drive();
function roadmap()	vehicle<char>::roadmap();

Class Templates

Class Template Declarations and Definitions

A class template must be declared before any declaration of a corresponding template class. A class template definition can only appear once in any single compilation unit. A class template must be defined before any use of a template class that requires the size of the class or refers to members of the class.

In the following example, the class template `key` is declared before it is defined. The declaration of the pointer `keyiptr` is valid because the size of the class is not needed. The declaration of `keyi`, however, causes an error.

```
template <class L> class key;      // class template declared,
                                  // not defined yet
                                  //
class key<int> *keyiptr;          // declaration of pointer
                                  //
class key<int> keyi;              // error, cannot declare keyi
                                  // without knowing size
                                  //
template <class L> class key      // now class template defined
{
//      .
//      .
//      .
};
```

If a template class is used before the corresponding class template is defined, the compiler issues an error. A class name with the appearance of a template class name is considered to be a template class. In other words, angle brackets are valid in a class name only if that class is a template class.

The definition of a class template is not compiled until the definition of a template class is required. At that point, the class template definition is compiled using the argument list of the template class to instantiate the template arguments. Any errors in the class definition are flagged at this time. If the definition of a class template is never required, it is not compiled. In this case, some errors in the definition might not be flagged by the compiler.

Class Templates

Reference and Uniqueness

A class template can only be defined once within a compilation unit, and the class template name cannot be declared to refer to any other template, class, object, function, value, or type in the same scope.

Nontype Template Arguments

A nontype template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members. Nontype template arguments are normally used to initialize a class or to specify the sizes of class members.

For nontype integral arguments, the instance argument matches the corresponding template argument as long as the instance argument has a value and sign appropriate to the argument type.

For nontype address arguments, the type of the instance argument must be of the form `identifier` or `&identifier`, and the type of the instance argument must match the template argument exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of nontype template arguments within a template argument list form part of the template class's type. If two template class names have the same template name and if their arguments have identical values, they are the same class.

In the following example, a class template is defined that requires a nontype template **int** argument as well as the type argument:

```
template<class T, int size> class myfilebuf
{
    T* filepos;
    static int array[size];
public:
    myfilebuf() { /* ... */ }
    ~myfilebuf();
    advance(); // function defined elsewhere in program
};
```

In this example, the template argument `size` becomes a part of the template class name. An object of such a template class is created with both the type arguments of the class and the values of any additional template arguments.

An object `x`, and its corresponding template class with arguments **double** and `size=200`, can be created from this template with a value as its second template argument:

```
myfilebuf<double,200> x;
```

`x` can also be created using an arithmetic expression:

```
myfilebuf<double,10*20> x;
```

Class Templates

The objects created by these expressions are identical because the template arguments evaluate identically. The value 200 in the first expression could have been represented by an expression whose result at compile time is known to be equal to 200, as shown in the second construction.

Note: Arguments that contain the < symbol or the > symbol must be enclosed in parentheses to prevent it from being parsed as a template argument list delimiter when it is being used as a relational operator or a nested template delimiter. For example, the arguments in the following definition are valid:

```
myfilebuf<double, (20>10)> x;      // valid
```

The following definition, however, is not valid because the greater than operator (>) is interpreted as the closing delimiter of the template argument list:

```
myfilebuf<double, 20>10> x;      // error
```

If the template arguments do not evaluate identically, the objects created are of different types:

```
myfilebuf<double,200> x;          // create object x of class
                                  // myfilebuf<double,200>
myfilebuf<double,200.0> y;        // error, 200.0 is a double,
                                  // not an int
```

The instantiation of `y` fails because the value `200.0` is of type **double**, and the template argument is of type **int**.

The following two objects:

```
myfilebuf<double, 128> x
myfilebuf<double, 512> y
```

belong to separate template classes, and referencing either of these objects later with `myfilebuf<double>` is an error.

A class template does not need to have a type argument if it has nontype arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
    public:
        int k;
        C() { k = i; }
};
```

This class template can be instantiated by declarations such as:

```
class C<100>;
class C<200>;
```

Again, these two declarations refer to distinct classes because the values of their nontype arguments differ.

Function Templates

Explicitly Defined Template Classes

You can override the definition of a class template of a particular template class by providing a class definition for the type of class required. For example, the following class template creates a class for each type for which it is referenced, but that class may be inappropriate for a particular type:

```
template<class M> class portfolio
{
    double capital;
    M arr;
    // ...
};
```

The type for which the template class is inappropriate can be defined by using the applicable template class name. Assuming the inappropriately defined type is `stocks`, you can redefine the class `portfolio<stocks>` as follows:

```
class portfolio<stocks>
{
    double capital;
    stocks yield;
    // ...
};
```

An explicit specialization of a template class can be defined before the class template is declared. In particular, a template class such as `portfolio<stocks>` can be defined before its class template has been defined.

Function Templates

A function template allows you to define a group of functions that are the same except for the types of one or more of their arguments or objects. All type arguments in a function template must be used in the argument list or in the class qualifier for the function name. The type of a template function argument need not be explicitly specified when the template function is called. In this respect, a template function differs from a template class.

Note the distinction between the terms *function template* and *template function*:

Function template is a template used to generate template functions. A function template can be only a declaration, or it can define the function.

Template function is a function generated by a function template.

Example of a Function Template

If you want to create a function `approximate()`, which determines whether two values are within 5% of each other, you can define the following template:

Function Templates

```
#include <math.h>
template <class T> int approximate (T first, T second)
{
    double aptemp=double(first)/double(second);
    return int(abs(aptemp-1.0) <= .05);
};
```

Assuming you have two values of type **float** you want to compare, you can use the approximate function template:

```
float a=3.24, b=3.35;
if (approximate(a,b))
    cout << "a and b are pretty close" << endl;
```

A template function `int approximate(float,float)` is generated to resolve the call.

Overloading Resolution for Template Functions

Resolution of overloaded template functions is done in the following order:

1. Look for a function with an exact type match. This does not include template functions, unless such functions were explicitly declared using a function declaration. Trivial conversions are performed if they produce an exact type match.
2. Look for a function template that allows generation of a function with an exact type match. Trivial conversions are performed if they produce an exact type match.
3. Try ordinary overloading resolution for functions already present. This does not include template functions, unless such functions were explicitly declared using a function declaration.

A call to a template function causes an error, and no overloading is done if the following conditions are true:

- The only available functions for a call are template functions.
- These functions would require nontrivial conversions for the call to succeed.
- These functions have not been explicitly declared.

In the case of the `approximate()` function template, if the two input values are of different types, overloading resolution does not take place:

```
float a=3.24;
double b=3.35;
if (approximate(a,b)) // error, different types
{ /* ... */ }
```

The solution is to force a conversion to one of the available function types by explicitly declaring the function for the chosen type. To resolve the **float/double** example, include the following function declaration:

```
int approximate(double a, double b);
// force conversion of the float to double
```


Function Templates

This declaration creates a function `approximate()` that expects two arguments of type **double**, so that when `approximate(a,b)` is called, the overloading is resolved by converting variable `a` to type **double**.

For more information on argument matching and conversions, see “Trivial Conversions” on page 281, “Standard Type Conversions” on page 132, and “Integral Promotions” on page 131.

Defining Template Functions

Because template functions can be generated in all compilation units that contain function template definitions, you may want to group function template definitions into one or two compilation units.

Explicitly Defined Template Functions

In some situations, a function template can define a group of functions in which, for one function type, the function definition would be inappropriate. For instance, the function template:

```
template<class T> int approximate(T first, T second);
```

defined in “Example of a Function Template” on page 356, determines whether two values are within 5% of each other. The algorithm used for this function template is appropriate for numerical values, but for **char*** values, it would indicate whether the *pointers* to two character strings are within 5% of one another, not whether the strings themselves are approximately equal. Whether two pointers are within 5% of each other is not useful information. You can define an explicit template function for **char*** values to compare the two strings themselves, character by character.

The following explicitly defined template function compares two strings and returns a value indicating whether more than 5% of the characters differ between the two strings:

```
#include <string.h>
int approximate(char *first, char *second)
{
    if (strcmp(first,second) == 0)
        return 1; // strings are identical

    double difct=0;
    int maxlen=0;

    if (strlen(first)>strlen(second))
        maxlen=strlen(first);

    else maxlen=strlen(second);
    for (int i=0; i<=maxlen ; ++i)
        if ( first[i] != second[i] ) difct++;
    return int((difct / maxlen) <= .05 );
}
```

Given this definition, the function call:

```
approximate("String A","String B");
```

Function Templates

invokes the explicitly defined function above, and no template function is generated.

Explicit definition has the same effect on template overloading resolution as explicit declaration (See “Overloading Resolution for Template Functions” on page 357 for more information.) If a template function is explicitly defined for:

```
int approximate(double a, double b) { /* ... */ }
```

then a call of:

```
double a=3.54;
float b=3.5;
approximate(a,b);
```

resolves in a call to

```
approximate(double a, double b)
```

and variable `b` is converted to type `double`.

Function Template Declarations and Definitions

When a template function is defined explicitly within a compilation unit, this definition is used in preference to any instantiation from the function template. For example, if one compilation unit contains the code:

```
#include <iostream.h>
template <class T> T f(T i) {return i+1;}
void main()
{
    cout << f(2) << endl;
}
```

and another contains:

```
int f(int i) {return i+2;}
```

when compiled and run, the program prints the number 4 to standard output, indicating that the explicitly defined function was used to resolve the call to `f()`.

Each template, whether of a class or of a function, must be defined at most once within a compilation unit. The same applies to an explicitly defined template class or function. Function templates and class templates can be declared many times.

A template class is considered declared if its name is used. A template function is considered declared if any of the following applies:

- A function whose name matches a function template's name is *declared*, and an appropriate template function can be generated.
- A function whose name matches a function template's name is *called*, and an appropriate template function can be generated.
- A function whose name matches a function template's name is called, and the template function has been explicitly defined.

Differences between Class and Function Templates

- The address of a template function is taken in such a way that instantiation can occur. This means the pointer to function must supply a return type and argument types that can be used to instantiate the template function.

A template function is instantiated or generated if the function is referenced in any of the following ways, provided that function is not explicitly defined elsewhere in the program:

- The function is declared.
- A call to the function is made.
- The address of the function is taken.

When a template function is instantiated, the body of the function template is compiled using the template argument list of the template class to instantiate the template arguments. Any errors in the function definition are flagged at this time. If a template function is never generated from a function template, it is not compiled. In this case, some errors in the function definition might not be flagged by the compiler.

Differences between Class and Function Templates

The name of a template class is a compound name consisting of the template name and the full template argument list enclosed in angle braces. Any references to a template class must use this complete name. For example:

```
template <class T, int range> class ex
{
    T a;
    int r;
    // ...
};
//...
ex<double,20> obj1;    // valid
ex<double> obj2;     // error
ex obj3;             // error
```

C++ requires this explicit naming convention to ensure that the appropriate class can be generated.

A template function, on the other hand, has the name of its function template and the particular function chosen to resolve a given template function call is determined by the type of the calling arguments. In the following example, the call `min(a,b)` is effectively a call to `min(int a, int b)`, and the call `min(af, bf)` is effectively a call to `min(float a, float b)`:

Member Function Templates

```
// This example illustrates a template function.

template<class T> T min(T a, T b)
{
    if (a < b)
        return a;
    else
        return b;
}

void main()
{
    int a = 0;
    int b = 2;
    float af = 3.1;
    float bf = 2.9;
    cout << "Here is the smaller int " << min(a,b) << endl;
    cout << "Here is the smaller float " << min(af, bf) << endl;
}
```

Member Function Templates

In “Function Templates” on page 356, a function template was defined outside of any template class. However, functions in C++ are often member functions of a class. If you want to create a class template and a set of function templates to go with that class template, you do not have to create the function templates explicitly, as long as the function definitions are contained within the class template. Any member function (inlined or noninlined) declared within a class template is implicitly a function template. When a template class is declared, it implicitly generates template functions for each function defined in the class template.

Member Function Templates

You can define template member functions three ways:

- Explicitly at file scope for each type used to instantiate the template class. For example:

```
template <class T> class key
{
public:
    void f(T);
};
void key<char>::f(char) { /* ... */ }
void key<int>::f(int ) { /* ... */ }

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

- At file scope with the template arguments. For example:

```
template <class T> class key
{
public:
    void f(T);
};
template <class T> void key <T>::f(T) { /* ... */ }

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

Member Function Templates

- Inlined in the class template itself. For example:

```
template <class T> class key
{
public:
    void f(T) { /* ... */ }
};

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

Member function templates are used to instantiate any functions that are not explicitly generated. If you have both a member function template and an explicit definition, the explicit definition is used.

The template argument is not used in a constructor name. For example:

```
template<class L> class Key
{
    Key();           // default constructor
    Key( L );       // constructor taking L by value
    Key<L>( L );    // error, <L> implicit within class template
};
```

The declaration `Key<L>(L)` is an error because the constructor does not use the template argument. Assuming this class template was corrected by removing the offending line, you can define a function template for the class template's constructor:

```
// Constructor contained in function template:
template<class L>
    Key<L>::Key(int) { /* ... */ }
    // valid, constructor template argument assumed template<class L>

    Key<L>::Key<L>(int) { /* ... */ }
    /* error, constructor template argument <L> implicit
       in class template argument */
```

A template function name does not include the template argument. The template argument does, however, appear in the template class name if a member function of a template class is defined or declared outside of the class template. The definition:

```
Key<L>::Key(int) { /* ... */ }
```

is valid because `Key<L>` (with template argument) refers to the class, while `Key(int) { /* ... */ }` refers to the member function.

Friends and Templates

Friends and Templates

A friend function can be declared in a class template either as a single function shared by all classes created by the template or as a template function that varies from class to class within the class template. For example:

```
template<class T> class portfolio
{
    //...
    friend void taxes();
    friend void transact(T);
    friend portfolio<T>* invest(portfolio<T>*);
    friend portfolio* divest(portfolio*);          //error
    // ...
};
```

In this example, each declaration has the following characteristics:

- `taxes()` is a single function that can access private and protected members of any template class generated by the class template. Note that `taxes()` is not a template function.
- `transact(T)` is a function template that declares a distinct function for each class generated by the class template. The only private and protected members that can be accessed by functions generated from this template are the private and protected members of their template class.
- `invest(portfolio<T>*)` is a function template whose return and argument types are pointers to objects of type `portfolio<T>`. Each class generated by the class template will have a friend function of this name, and each such function will have a pointer to an object of its own class as both its return type and its argument type.
- `divest(portfolio*)` is an error because `portfolio*` attempts to point to a class template. A pointer to a class template is undefined and produces an error. This statement can be corrected by using the syntax of the `invest()` function template instead.

Because all friend functions in this example are declared but not defined, you could create a set of function templates to define those functions that are implicitly template functions (that is, all the valid functions except `taxes()`). The function templates would then be used to instantiate the template functions as required.

Static Data Members and Templates

Static Data Members and Templates

A **static** declaration within a class template declares a static data member for each template class generated from the template. The static declaration can be of template argument type or of any defined type.

Like member function templates, you can explicitly define a static data member of a template class at file scope for each type used to instantiate a template class. For example:

```
template <class T> class key
{
public:
    static T x;
};
int key<int>::x;
char key<char>::x;
void main()
{
    key<int>::x = 0;
}
```

You can also define a static data member of a template class using a template definition at file scope. For example:

```
template <class T> class key
{
public:
    static T x;
};
template <class T> T key<T> ::x; // template definition
void main()
{
    key<int>::x = 0;
}
```

In the following example:

```
template<class L> class Key
{
    static L k;
    static L* kptr;
    static int length;
    // ...
}
```

The definitions of static variables and objects must be instantiated at file scope. If the classes `Key<int>` and `Key<double>` are instantiated from this template, and no template definitions exist, the following static data members must be explicitly defined at file scope, or an error occurs:

Static Data Members and Templates

```
int Key<int>::k, Key<int>::length, Key<double>::length;  
int* Key<int>::kptr;  
double Key<double>::k;  
double* Key<double>::kptr = 0;
```

Chapter 15. Exception Handling

This chapter describes the VisualAge for C++ for AS/400 implementation of C++ exception handling. It discusses:

C++ Exception Handling Overview	367
Formal and Informal Exception Handling	368
Using Exception Handling	368
Transferring Control	370
Constructors and Destructors in Exception Handling	376
Exception Specifications	379
Special Exception Handling Functions	381

Related Information

- Chapter 4, “Expressions and Operators” on page 93
- Chapter 12, “Special Member Functions” on page 293

C++ Exception Handling Overview

Exception handling provides a way for a function that encounters an unusual situation to throw an exception and pass control to a direct or indirect caller of that function. The caller may or may not be able to handle the exception. Code that intercepts an exception is called a handler. Regardless of whether or not the caller can handle an exception, it may rethrow the exception so it can be intercepted by another handler.

C++ provides three language constructs to implement exception handling:

- Try blocks
- Catch blocks
- Throw expressions

Within a function, any unusual situation can be flagged with a throw expression. The throw expression is of type void. Your program can throw an object to pass information back to the caller. Any object can be thrown, including the object that caused the exception or an object constructed when the exception occurred.

A throw expression, or a call to a function that may throw an exception, should be enclosed within a try block. If the called function throws an exception and an exception handler is defined to catch the type of the object thrown, the exception handler is executed. In C++, a catch block implements an exception handler. A try block must be accompanied by one or more catch clauses, otherwise the compiler will flag it as an error.

A catch block follows immediately after a try statement or immediately after another catch block. A catch block includes a parenthesized exception declaration containing optional qualifiers, a type, and an optional variable name. The declaration specifies the type of object that the exception handler may catch. Once an exception is caught, the

Using Exception Handling

body of the catch block is executed. If no handler catches an exception, the program is terminated.

Exception handling is not strictly synonymous with error handling, because the implementation allows the passing of an exception whether or not an error actually occurred. You can use exception handlers for things other than handling errors. For example, you can transfer control back to the original caller of a function. You might use this if you wanted to process the Quit key in a program and transfer control back to the driver program when the user types Quit. To do this exception handlers could be used to throw an object back to the driver.

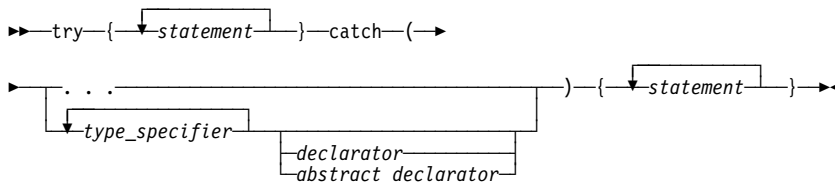
Formal and Informal Exception Handling

While the exception handling features of C++ offer a formal mechanism for handling exceptions (language implemented), in many situations informal exception handling (logic implemented) is more appropriate. Generally speaking, formal exception handling should be implemented in libraries, classes, and functions likely to be accessed by several programs or programmers. It should also be used in classes and functions that are repeatedly accessed within a program but are not well-suited to handling their exceptions themselves. Because formal exception handling is designed for exceptional circumstances, it is not guaranteed to be efficient. Program performance is usually not affected when you do not invoke formal exception handling, although it can inhibit some optimizations.

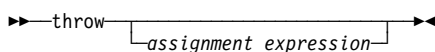
Informal exception handling, in which an appropriate action is defined if an error or exception occurs, is often more suitable for handling errors. For example, a simple error, such as entering incorrect input, can more easily and clearly be handled by testing the input for validity and by requesting the input again if the original input is incorrect.

Using Exception Handling

The three keywords designed for exception handling in C++ are **try**, **catch**, and **throw**.



The syntax for the **throw** keyword is:



Using Exception Handling

The steps required to implement an exception handler are:

1. Functions that are expected to be used by many programs are coded so that, when an error is detected, an exception is thrown. The throw expression generally throws an object. It may be created explicitly for purposes of exception handling, or it may be the object that caused the error to be detected. An example of throwing the problem object:

```
    :
int input=0;
cout << "Enter a number between 1 and 10:";
cin >> input;

if (input < 1 || input >> 10);
    throw(input); //throw the actual problem object
    :
```

The following is an example of throwing an object for the purpose of exception handling:

```
    :
int input=0;
cout << "Enter a number between 1 and 10:";
cin >> input;

if (input < 1 || input >> 10)
    throw(out_of_range_object); //throw object to tell handler
                                //what happened
```

2. Exceptions are anticipated in a caller by means of a try statement. Function calls that you anticipate might produce an exception must be enclosed in braces and preceded by the keyword **try**.
3. Immediately following the try block, you must code one or more catch blocks. Each catch block identifies what type or class of objects it can catch:
 - a. If the object thrown matches the type of a catch expression, control passes to that catch block.
 - b. If the object thrown does not match the first catch block, subsequent catch blocks are searched for a matching type.
 - c. If no match is found, the search continues in all enclosing try blocks and then in the code that called the current function.
 - d. If no match is found after all try blocks are searched, a call to **terminate()** is made.

For information on the default handlers of uncaught exceptions, see “Special Exception Handling Functions” on page 381.

Transferring Control

Notes:

1. Any object can be thrown if it can be copied and destroyed in the function from which the throw occurs.
2. Exceptions should never be thrown from a C language signal handler. The result is undefined, and can cause program termination.

A catch argument causes an error if it is a value argument, and a copy of it cannot be generated. For example:

```
class B {
public:
    B();
    B(B&);
};

// the following catch block will cause an error
//
catch(const B x)
{
    // .
    // .
    // .
}
```

The catch block causes an error because the compiler does not know the type of the object thrown at compile time. It assumes that the type of the thrown object is the same as the type of the catch argument. In the above example, the thrown object is assumed to be of type `const B`. The compiler uses a copy constructor on the thrown argument to create the catch argument. Because there is no copy constructor for class `B` that accepts `const B` as an input argument, the compiler cannot perform the construction and an error occurs. Similarly, a throw expression causes an error if a copy of the value of the expression being thrown cannot be generated.

Transferring Control

C++ implements the *termination model* of exception handling. In the termination model, when an exception is thrown, control never returns to the *throw point*. The throw point is the point in program execution where the exception occurred.

C++ exception handling does not implement the *resumption model* of exception handling, which allows an exception handler to correct the exception and then return to the throw point.

When an exception is thrown, control is passed out of the throw expression and out of the try block that anticipated the exception. Control is passed to the catch block whose exception type matches the object thrown. The catch block handles the exception as appropriate. If the catch block ends normally, the flow of control passes over all subsequent catch blocks.

Transferring Control

When an exception is not thrown from within a try block, the flow of control continues normally through the block, and passes over all catch blocks following the try block.

An exception handler cannot return control to the source of the error by using the **return** statement. A **return** issued in this context returns from the function containing the catch block.

If an exception is thrown and no try block is active, or if a try block is active and no catch block exception declaration matches the object thrown, a call to **terminate()** is issued. A call to **terminate()** in turn calls **abort()** to terminate the program. The **abort()** C library function is defined in the standard header file `<stdlib.h>`.

For more information on **terminate()**, see “Special Exception Handling Functions” on page 381.

The following example illustrates the basic use of **try**, **catch**, and **throw**. The program prompts for numerical input and determines the input's reciprocal. Before it attempts to print the reciprocal to standard output, it checks that the input value is nonzero, to avoid a division by zero. If the input is zero, an exception is thrown, and the catch block catches the exception. If the input is nonzero, the reciprocal is printed to standard output.

```
// This example illustrates the basic use of
// try, catch, and throw.

#include <iostream.h>
#include <stdlib.h>
class IsZero { /* ... */ };
void ZeroCheck( int i )
{
    if (i==0)
        throw IsZero();
}
void main()
{
    double a;

    cout << "Enter a number: ";
    cin >> a;
    try
    {
        ZeroCheck( a );
        cout << "Reciprocal is " << 1.0/a << endl;
    }
    catch ( IsZero )
    {
        cout << "Zero input is not valid" << endl;
        exit(1);
    }
    exit(0);
}
```

Transferring Control

This example could have been coded more efficiently by using informal exception handling. However, it provides a simple illustration of formal exception handling.

Catching Exceptions

You can declare a handler to catch many types of exceptions. The allowable objects that a function can catch are declared in the parentheses following the **catch** keyword (the *catch argument*). You can catch objects of the fundamental types, base and derived class objects, references, and pointers to all of these types. You can also catch **const** and **volatile** types.

You can also use the **catch(...)** form of the handler to catch all thrown exceptions that have not been caught by a previous catch block. The ellipsis in the catch argument indicates that any exception thrown can be handled by this handler.

If an exception is caught by a **catch(...)** block, there is no direct way to access the object thrown. Information about an exception caught by **catch(...)** is very limited.

You can declare an optional variable name if you want to access the thrown object in the catch block.

A catch block can only catch accessible objects. The object caught must have an accessible copy constructor. For more information on access, see “Member Access” on page 270; on copy constructors, see “Copy by Initialization” on page 313.

Matching between Exceptions Thrown and Caught

An argument in the catch argument of a handler matches an argument in the *expression* of the throw expression (throw argument) if any of the following conditions is met:

- The catch argument type matches the type of the thrown object.
- The catch argument is a public base class of the thrown class object.
- The catch specifies a pointer type, and the thrown object is a pointer type that can be converted to the pointer type of the catch argument by standard pointer conversion. Pointer conversion is described on page 133.

Note: If the type of the thrown object is **const** or **volatile**, the catch argument must also be a **const** or **volatile** for a match to occur. However, a **const**, **volatile**, or reference type catch argument can match a nonconstant, nonvolatile, or nonreference object type. A nonreference catch argument type matches a reference to an object of the same type.

Order of Catching

Always place a catch block that catches a derived class before a catch block that catches the base class of that derived class (following a try block). If a catch block for objects of a base class is followed by a catch block for objects of a derived class of that base class, the latter block is flagged as an error.

Transferring Control

A catch block of the form **catch(...)** must be the last catch block following a try block or an error occurs. This placement ensures that the **catch(...)** block does not prevent more specific catch blocks from catching exceptions intended for them.

Nested Try Blocks

When try blocks are nested and a **throw** occurs in a function called by an inner try block, control is transferred outward through the nested try blocks until the first catch block is found whose argument matches the argument of the throw expression.

For example:

```
try
{
    func1();
    try
    {
        func2();
    }
    catch (spec_err) { /* ... */ }
    func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

In the above example, if `spec_err` is thrown within the inner try block (in this case, from `func2()`), the exception is caught by the inner catch block, and, assuming this catch block does not transfer control, `func3()` is called. If `spec_err` is thrown after the inner try block (for instance, by `func3()`), it is not caught and the function **terminate()** is called. If the exception thrown from `func2()` in the inner try block is `type_err`, the program skips out of both try blocks to the second catch block without invoking `func3()`, because no appropriate catch block exists following the inner try block. If the entire try block in the example is in a function that has a throw list and does not include `spec_err` on its throw list, **unexpected()** is called.

The function **unexpected()** is discussed in “Special Exception Handling Functions” on page 381.

You can also nest a try block within a catch block.

Rethrowing an Exception

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (**throw** with no argument) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks following the dynamically enclosing try block have an opportunity to catch the exception.

Transferring Control

In the following example, `catch(FileIO)` catches any object of type `FileIO` and any objects that are public base classes of the `FileIO` class. It then checks for those exceptions it can handle. For any exception it cannot handle, it issues a `rethrow` expression to rethrow the exception and allow another handler in a dynamically enclosing `try` block to handle the exception.

// This example illustrates rethrowing an exception.

```
#include <iostream.h>
class FileIO
{
public:
    int notfound;
    int endfile;
    FileIO(); // initialize data members
    // the following member functions throw an exception
    // if an input error occurs
    void advance(int x);
    void clear();
    void put(int x, int y);
};
// .
// .
// .
void f()
{
    FileIO fio;
    try
    {
        // call member functions of FileIO class
        fio.advance (1);
        fio.clear();
        fio.put(1,-1);
    }
}
```

Transferring Control

```
catch(FileIO fexc)
{
    if (fexc.notfound)
        cout << "File not Found" << endl;
    else if (fexc.endfile)
        cout << "End of File" << endl;
    else
        throw;           // rethrow to outer handler
}
catch(...) { /* ... */ } // catch other exceptions
}
main()
{
    try
    {
        f();
    }
    catch(FileIO) { cout << "Outer Handler" << endl; }
}
```

The rethrow expression can be caught by any **catch** whose argument matches the argument of the exception originally thrown. Note that, in this example, the **catch(...)** will not catch the rethrow expression because, when the rethrow expression is issued, control passes out of the scope of the function `f()` into the next dynamically enclosing block.

Using a Conditional Expression in a Throw Expression

You can use a conditional expression as a throw expression. as shown in the following example:

```
// This example illustrates a conditional expression
// used as a throw expression.

#include <iostream.h>
void main() {
    int doit = 1;
    int dont = 0;
    float f = 8.9;
    int i = 7;
    int j = 6;
    try { throw(doit ? i : f); }
    catch (int x)
    {
        cout << "Caught int " << x << endl;
    }
    catch (float x)
    {
        cout << "Caught float " << x << endl;
    }
    catch (double x)
    {
        cout << "Caught double " << x << endl;
    }
}
```

Constructors and Destructors

```
    }  
    catch (...)  
    {  
        cout << "Caught something " << endl;  
    }  
}
```

This example produces the following output because `j` is of type **int**:

```
Caught float 7
```

At first glance, it looks as if the block that catches integer values should do the catch, but `i` is converted to a **float** value in the try block because it is in a conditional expression with the float value `f`. If the try block in the example is replaced with the following try block:

```
    try { throw doit ? i : j; }
```

The following output is produced:

```
Caught int 7
```

Constructors and Destructors in Exception Handling

When an exception is thrown and control passes to a catch block following a try block, destructors are called for all automatic objects constructed since the beginning of the try block directly associated with that catch block. If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

For more information on constructors and destructors, see “Constructors and Destructors Overview” on page 293.

If a destructor detects an exception and issues a throw, the exception can be caught if the caller of the destructor was contained within a try block and an appropriate catch is coded.

If an exception is thrown by a function called from an inner try block, but caught by an outer try block (because the inner try block did not have an appropriate handler), all objects constructed within both the outer and all inner try blocks are destroyed. If the thrown object has a destructor, the destructor is not called until the exception is caught and handled.

Because a throw expression throws an object and a catch statement can catch an object, the object thrown enables error-related information to be transferred from the point at which an exception is detected to the exception's handler. If you throw an object with a constructor, you can construct an object that contains information relevant to the catch expression.

Constructors and Destructors

In the following example, an object of class `DivideByZero` is thrown by the function `divide()`. The constructor copies the string "Division by zero" into the **char** array `errmsg`. Because `DivideByZero` is a derived class of class `Matherr`, the catch block for `Matherr` catches the thrown exception. The catch block can then access information provided by the thrown object, in this case the text of an error message.

```
// This example illustrates constructors and
// destructors in exception handling.

#include <string.h>           // needed for strcpy
#include <iostream.h>
class Matherr { public: char errmsg[30]; };
class DivideByZero : public Matherr
{
public:
    DivideByZero() {strcpy (errmsg, "Division by zero");}
};
double divide(double a, double b)
{
    if (b == 0) throw DivideByZero();
    return a/b;
}
void main()
{
    double a=7,b=0;
    try {divide (a,b);}
    catch (Matherr xx)
    {
        cout << xx.errmsg << endl;
    }
}
```

Exception handling can be used in conjunction with constructors and destructors to provide resource management that ensures that all locked resources are unlocked when an exception is thrown.

Constructors and Destructors

```
class data
{
    public:
        void lock();        // prevent other users from
                           // changing the object
        void unlock();     // allow other users to change
                           // the object
};
void q(data&), bar(data&);
// .
// .
// .
main()
{
    data important;
    important.lock();
    q(important);
    bar(important);
    important.unlock();
}
```

If `q()` or `bar()` throw an exception, `important.unlock()` will not be called and the data will stay locked. This problem can be corrected by using a helper class to write an *exception-aware* program for resource management.

```
class data
{
    public:
        void lock();        // prevent other users from
                           // changing the object
        void unlock();     // allow other users to change
                           // the object
};
class locked_data          // helper class
{
    data& real_data;
    public:
        locked_data(data& d) : real_data(d)
                               {real_data.lock();}
        ~locked_data() {real_data.unlock();}
};
void q(data&), bar(data&);
// .
// .
// .
main()
{
    data important;
    locked_data my_lock(important);
    q(important);
    bar(important);
}
```

Exception Specifications

In this case, if `q()` or `bar()` throws an exception, the destructor for `my_lock` will be called, and the data will be unlocked.

Exception Specifications

C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function's caller that the function will not directly or indirectly throw any exception not contained in the exception specification. For example, a function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

explicitly states that it will not throw any exception other than `unknown_word` or `bad_grammar`. The function `translate()` must handle any exceptions thrown by functions it might call, unless those exceptions are specified in the exception specification of `translate()`. If an exception is thrown by a function called by `translate()` and the exception is not handled by `translate()` or contained in the exception specification of `translate()`, **unexpected()** is called. The function **unexpected()** is discussed in “Special Exception Handling Functions” on page 381.

There are qualifications to the rule about throwing only a specified list of exceptions. If a class `A` is included in the exception specification of a function, the function will also allow exception objects of any classes that are publicly derived from class `A`. Also, if a pointer type `B*` is included in the exception specification of a function, the function will allow exceptions of type `B*` or of pointers to any type publicly derived from `B*`.

Exception Specification Syntax

The syntax of the exception specification is:

```
▶▶ throw ( [ type ] ) ▶▶
```

The syntax of a function definition that includes an exception specification is:

```
▶▶ return_type function_name ( [ argument ] ) throw ( [ type ] ) { ▶▶  
▶▶ function_body ▶▶ }
```

An exception specification is not part of a function's type. If an exception is thrown from a function that has not specified the thrown exception in its exception specification, the result is a call to the function **unexpected()**, which is discussed in “Special Exception Handling Functions” on page 381.

Exception Specifications

Empty Exception Specifications

A function with an empty **throw()** specification guarantees that the function will not throw any exceptions.

Functions without an Exception Specification

A function without an exception specification allows any object to be thrown from the function.

Other Exception Specifications

The compiler does not prevent an exception specification from defining a more limited set of valid exceptions than the set of exceptions the function may actually throw. Such an error is detected only at run time, and only if the unspecified exception is thrown.

In the following example, `NameTooShort` is thrown from within a function that explicitly states that it will only throw `NameTooLong`. This is a valid function, although at run time, if `NameTooShort` is thrown, a call to **unexpected()** will be made.

```
#include <string.h>           // needed for strlen
class NameTooLong {};
class NameTooShort {};

void check(char* fname) throw (NameTooLong)
{
    if ( strlen(fname)<4 ) throw NameTooShort();
}
```

If a function with an exception specification calls a subfunction with a less restrictive exception specification (one that contains more objects than the calling function's exception specification), any thrown objects from within the subfunction that are not handled by the subfunction, and that are not part of the outer function's specification list, must be handled within the outer function. If the outer function fails to handle an exception not in its exception specification, a call to **unexpected()** is made.

Special Exception Handling Functions

Special Exception Handling Functions

Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are **unexpected()** and **terminate()**.

unexpected()

When a function with an exception specification throws an exception that is not listed in its exception specification, the function **void unexpected()** is called. Next, **unexpected()** calls a function specified by the **set_unexpected()** function. By default, **unexpected()** calls the function **terminate()**. In turn, **terminate()** calls **abort()** by default, terminating the program.

Although **unexpected()** cannot return, it may throw an exception. The search for a handler starts at the call of the function whose exception specification was violated. For more information, see “set_unexpected() and set_terminate().”

terminate()

In some cases, the exception handling mechanism fails and a call to **void terminate()** is made. This **terminate()** call occurs in any of the following situations:

- When **terminate()** is explicitly called
- When no catch can be matched to a thrown object
- When the stack becomes corrupted during the exception-handling process
- When a system defined **unexpected()** is called

The **terminate()** function calls a function specified by the **set_terminate()** function. By default, **terminate** calls **abort()**, which exits from the program.

A terminate function cannot return to its caller, either by using **return** or by throwing an exception.

set_unexpected() and set_terminate()

The function **unexpected()**, when invoked, calls the function most recently supplied as an argument to **set_unexpected()**. If **set_unexpected()** has not yet been called, **unexpected()** calls **terminate()**.

The function **terminate()**, when invoked, calls the function most recently supplied as an argument to **set_terminate()**. If **set_terminate()** has not yet been called, **terminate()** calls **abort()**, which ends the program.

You can use **set_unexpected()** and **set_terminate()** to register functions you define to be called by **unexpected()** and **terminate()**. **set_unexpected()** and **set_terminate()** are included in the standard header files. Each of these functions has as its return type and its argument type a pointer to function with a **void** return type and no arguments. The pointer to function you supply as the argument becomes the function called by the corresponding special function: the argument to **set_unexpected()** becomes the

Special Exception Handling Functions

function called by **unexpected()**, and the argument to **set_terminate()** becomes the function called by **terminate()**.

Both **set_unexpected()** and **set_terminate()** return a pointer to the function that was previously called by their respective special functions (**unexpected()** and **terminate()**). By saving the return values, you can restore the original special functions later so that **unexpected()** and **terminate()** will once again call **terminate()** and **abort()**.

If you use **set_terminate()** to register your own function, the final action of that program should be to exit from the program. If you attempt to return from the function called by **terminate()**, **abort()** is called instead and the program ends.

Note: Providing a call to **longjmp()** inside a user-defined **terminate** function can transfer execution control to some other desired point. When you call **longjmp**, objects existing at the time of a **setjmp** call will still exist, but some objects constructed after the call to **setjmp** might not be destructed.

The **longjmp** and **setjmp** functions are described in the *VisualAge for C++ for AS/400 C Library Reference*.

Example of Using the Exception Handling Functions

The following example shows the flow of control and special functions used in exception handling:

```
#include <iostream.h>
class X { /* ... */ };
class Y { /* ... */ };
class A { /* ... */ };
// pfv type is pointer to function returning void
typedef void (*pfv)();
void my_terminate()
{
    cout << "Call to my terminate" << endl; }
void my_unexpected()
{
    cout << "Call to my unexpected" << endl; }
void f() throw(X,Y)      // f() is permitted to throw objects of class
                        // types X and Y only
{
    A aobj;
    throw(aobj); // error, f() throws a class A object
}
main()
{
    pfv old_term = set_terminate(my_terminate);
    pfv old_unex = set_unexpected(my_unexpected);
    try{ f(); }
    catch(X)      { /* ... */ }
    catch(Y)      { /* ... */ }
    catch (...)   { /* ... */ }

    set_unexpected(old_unex);
    try { f();}
```

Special Exception Handling Functions

```
    catch(X)      { /* ... */ }
    catch(Y)      { /* ... */ }
    catch (...)   { /* ... */ }
}
```

At run time, this program behaves as follows:

1. The call to **set_terminate()** assigns to `old_term` the address of the function last passed to **set_terminate()** when **set_terminate()** was previously called.
2. The call to **set_unexpected()** assigns to `old_unex` the address of the function last passed to **set_unexpected()** when **set_unexpected()** was previously called.
3. Within a try block, function `f()` is called. Because `f()` throws an unexpected exception, a call to **unexpected()** is made. **unexpected()** in turn calls `my_unexpected()`, which prints a message to standard output and returns.
4. The second call to **set_unexpected()** replaces the user-defined function `my_unexpected()` with the saved pointer to the original function (**terminate()**) called by **unexpected()**.
5. Within a second try block, function `f()` is called once more. Because `f()` throws an unexpected exception, a call to **unexpected()** is again made. **unexpected()** automatically calls **terminate()**, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message. It returns, and the system calls **abort()**, which terminates the program.

At run time, the following information is displayed, and the program ends:

```
Call to my_unexpected
Call to my_terminate
```

Note: The catch blocks following the try block are not entered, because the exception was handled by `my_unexpected()` as an unexpected throw, not as a valid exception.

Special Exception Handling Functions

Constructs Found in Both C++ and ANSI/ISO C

Appendix A. C and C++ Compatibility

The differences between ANSI/ISO C and C++ fall into two categories:

- Constructs found in C++ but not in ANSI/ISO C
- Constructs found in both C++ and ANSI/ISO C, but treated differently in the two languages

C++ Constructs Not Found in ANSI/ISO C

C++ contains many constructs that are not found in ANSI/ISO C:

- Single-line comments beginning with `//` (See “Comments” on page 16)
- Scope operator (See “C++ Scope Resolution Operator `::`” on page 97)
- Free store management using the operators **new** and **delete** (See “C++ new Operator” on page 108 and “C++ delete Operator” on page 113)
- Linkage specification for functions (See “Linkage Specifications — Linking to non-C++ Programs” on page 12)
- Reference types (See “References” on page 91)
- Default arguments for functions (See “Default Arguments in C++ Functions” on page 156)
- Inline functions (See “Inline Functions” on page 163)
- Classes (See Chapter 9, “Classes” on page 239)
- Anonymous unions (See “Anonymous Unions in C++” on page 81)
- Overloaded operators and functions (See Chapter 11, “Overloading” on page 277)
- Class templates and function templates (See “Class Templates” on page 351 and “Function Templates” on page 356)
- Exception handling (See Chapter 15, “Exception Handling” on page 367)

Constructs Found in Both C++ and ANSI/ISO C

Because C++ is based on ANSI/ISO C, the two languages have many constructs in common. The use of some of these shared constructs differs, as shown here.

Character Array Initialization

In C++, when you initialize character arrays, a trailing `'\0'` (zero of type **char**) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

In ANSI/ISO C, space for the trailing `'\0'` can be omitted in this type of initialization.

The following initialization, for instance, is not valid in C++:

```
char v[3] = "asd"; // not valid in C++, valid in ANSI/ISO C
```

because four elements are required. This initialization produces an error because there is no space for the implied trailing `'\0'` (zero of type **char**).

For more information, see “Initializing Arrays” on page 63.

Constructs Found in Both C++ and ANSI/ISO C

Character Constants

A character constant has type **char** in C++ and **int** in ANSI/ISO C.

For more information, see “Character Constants” on page 23.

Class and typedef Names

In C++, a class and a **typedef** cannot both use the same name to refer to a different type within the same scope (unless the **typedef** is a synonym for the class name). In C, a **typedef** name and a **struct** tag name declared in the same scope can have the same name because they have different name spaces. For example:

```
void main ()
{
    typedef double db;
    struct db; // error in C++, valid in ANSI/ISO C

    typedef struct st st; // valid ANSI/ISO C and C++
}
```

For more information on **typedef**, see “typedef” on page 44. For information on class types, see Chapter 9, “Classes” on page 239. For information on structures, see “Structures” on page 70.

Class and Scope Declarations

In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function, or other declaration of that name in an enclosing scope. In ANSI/ISO C, an inner scope declaration of a **struct** name does not hide an object or function of that name in an outer scope. For example:

```
double db;
void main ()
{
    struct db // hides double object db in C++
    { char* str; };
    int x = sizeof(db); // size of struct in C++
                       // size of double in ANSI/ISO C
}
```

For more information, see “Scope of Class Names” on page 246. For general information about scope, see “Scope” on page 7.

const Object Initialization

In C++, **const** objects must be initialized. In ANSI/ISO C, they can be left uninitialized.

For more information, see “volatile and const Qualifiers” on page 84.

Constructs Found in Both C++ and ANSI/ISO C

Definitions

An object declaration, for example:

```
int i;
```

is a definition in C++. In ANSI/ISO C, it is a tentative definition.

In C++, a global data object must be defined only once. In ANSI/ISO C, a global data object can be declared several times without using the **extern** keyword.

In C++, multiple definitions for a single variable cause an error. A C compilation unit can contain many identical tentative definitions for a variable.

For more information, see Chapter 3, “Declarations” on page 29.

Definitions within Return or Argument Types

In C++, types may not be defined in return or argument types. ANSI/ISO C allows such definitions. For example, the declarations:

```
void print(struct X { int i;} x); // error in C++
enum count{one, two, three} counter(); // error in C++
```

produce errors in C++, but are valid declarations in ANSI/ISO C.

For more information, see “Function Declarations” on page 138 and “Calling Functions and Passing Arguments” on page 150.

Enumerator Type

An enumerator has the same type as its enumeration in C++. In ANSI/ISO C, an enumeration has type **int**.

For more information on enumerators, see “Enumerations” on page 50.

Enumeration Type

The assignment to an object of enumeration type with a value that is not of that enumeration type produces an error in C++. In ANSI/ISO C, an object of enumeration type can be assigned values of any integral type.

For more information, see “Enumerations” on page 50.

Function Declarations

In C++, all declarations of a function must match the unique definition of a function. ANSI/ISO C has no such restriction.

For more information, see “Function Declarations” on page 138.

Constructs Found in Both C++ and ANSI/ISO C

Functions with an Empty Argument List

Consider the following function declaration:

```
int f();
```

In C++, this function declaration means that the function takes no arguments. In ANSI/ISO C, it could take any number of arguments, of any type.

For more information, see “Function Declarations” on page 138.

Global Constant Linkage

In C++, an object declared **const** has internal linkage, unless it has previously been given external linkage. In ANSI/ISO C, it has external linkage.

For more information, see “Program Linkage” on page 8.

Jump Statements

C++ does not allow you to jump over declarations containing initializations. ANSI/ISO C does allow you to use jump statements for this purpose.

For more information, see “Initializers” on page 88.

Keywords

C++ contains some additional keywords not found in ANSI/ISO C. C programs that use these keywords as identifiers are not valid C++ programs:

Table 9. C++ Keywords

asm	inline	public	virtual
catch	new	template	wchar_t
class	operator	this	
delete	private	throw	
friend	protected	try	

For more information, see “Keywords” on page 19.

main() Recursion

In C++, **main()** cannot be called recursively and cannot have its address taken. ANSI/ISO C allows recursive calls and allows pointers to hold the address of **main()**.

For more information, see “The main() Function” on page 148.

Names of Nested Classes

In C++, the name of a nested class is local to its enclosing class. In ANSI/ISO C, the name of the nested structure belongs to the same scope as the name of the outermost enclosing structure.

For more information, see “Nested Classes” on page 248.

Constructs Found in Both C++ and ANSI/ISO C

Pointers to void

C++ allows **void** pointers to be assigned only to other **void** pointers. In ANSI/ISO C, a pointer to **void** can be assigned to a pointer of any other type without an explicit cast.

For more information, see “void Type” on page 60 and “Pointers” on page 55.

Prototype Declarations

C++ requires full prototype declarations. ANSI/ISO C allows nonprototyped functions.

For more information, see “Function Declarator” on page 145.

Return without Declared Value

In C++, a return (either explicit or implicit) from **main()** that is declared to return a value results in an error if no value is returned. A return (either explicit or implicit) from all other functions that is declared to return a value *must* return a value. In ANSI/ISO C, a function that is declared to return a value can return with no value, with unspecified results.

For more information, see “Function Return Values” on page 159.

__STDC__ Macro

The predefined macro variable `__STDC__` is not defined for C++. It has the integer value 0 when used in a **#if** statement, indicating that the C++ language is not a proper superset of C, and that the compiler does not conform to ANSI/ISO C. In ANSI/ISO C, `__STDC__` has the integer value 1.

For more information on macros, see “Predefined Macro Names” on page 203.

typedefs in Class Declarations

In C++, a **typedef** name may not be redefined in a class declaration after being used in the declaration. ANSI/ISO C allows such a declaration. For example:

```
void main ()
{
    typedef double db;
    struct st
    {
        db x;
        double db; // error in C++, valid in ANSI/ISO C
    };
}
```

For more information, see “typedef” on page 44.

Constructs Found in Both C++ and ANSI/ISO C

Bibliography

This bibliography lists the publications that make up the IBM VisualAge for C++ for AS/400 library and publications of related IBM products referenced in this book. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ for AS/400 users.

The IBM VisualAge for C++ for AS/400 Library

The following books are part of the IBM VisualAge for C++ for AS/400 library.

- *VisualAge for C++ for AS/400 Licensed Programming Specification*, SC09-2414
- *VisualAge for C++ for AS/400 Installation Guide and Product Overview*, SC09-2415
- *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416
- *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440
- *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441
- *VisualAge for C++ for AS/400 C++ Programming Guide*, SC09-2417
- *ILE C/C++ MI Library Reference*, SC09-2418
- *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442
- *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443
- *IBM Access Class Library for OS/400 Reference*, SC41-4620
- *IBM Access Class Library for Windows Reference*, SC41-4622
- *IBM Access Class Library User's Guide*, SC41-4623
- *ILE Concepts*, SC41-4606

C and C++ Related Publications

- *Portability Guide for IBM C*, SC09-1405
- *American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])*
- *Draft Proposed American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)*

Other Books You Might Need

The following list contains the titles of IBM books that you might find helpful. These books are not part of the VisualAge for C++ for AS/400 or operating system libraries.

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.
- *C++ Primer* by Stanley B. Lippman, Addison-Wesley Publishing Company.
- *Object-Oriented Design with Applications* by Grady Booch, Benjamin/Cummings.
- *Object-Oriented Programming Using SOM and DSOM* by Christina Lau, Van Nostrand Reinhold.

Index

Special Characters

~ bitwise negation operator 105
|| logical OR operator 122, 129
!= not equal to operator 119
! logical negation operator 105
?: conditional operators 123
?? trigraphs 16
/= compound assignment operator 128
/ division operator 115
|= assignment operator 128
[] array subscript operators 102
" double quotation mark 24
<= less than or equal to operator 118
< less than operator 118
<<= compound assignment operator 128
<< left-shift operator 117
*= compound assignment operator 128
* indirection operator 106
* multiplication operator 115
\ continuation character 24, 27, 192
\ escape character 27
&= compound assignment operator 128
& address operator 106
& bitwise AND operator 120
&& logical AND operator 121
| bitwise inclusive OR operator 121
preprocessor directive character 192
preprocessor operator 198
-- decrement operator 104
+= compound assignment operator 128
+ addition operator 116
++ increment operator 103
= simple assignment operator 126
== equal to operator 119
^= compound assignment operator 128
>= greater than or equal to operator 118
^ bitwise exclusive OR operator 120
> greater than operator 118
>>= compound assignment operator 128
>> right-shift operator 117

A

abort function 381
abstract classes 343

access
 base classes 325
 constructors 293
 declarations 253, 326
 derived class 324
 effective 329
 exception handling 372
 friends 276
 inherited member 324
 members 270
 multiple 335
 private 325
 protected members 324
 public 325
 resolution 329
 specifiers 271, 323, 325
 summary example 331
 virtual function 342
accessibility 270, 329, 335
accessibility 270, 329, 335
additive operators
 addition + 116
 subtraction - 116
address operator & 106
aggregate classes
 constructors and destructors 293
 description 241
aggregate operands 96
alarm escape sequence \a 27
alert escape sequence \a 27
alias
 See references
allocation expressions 108
ambiguities
 base classes 333, 335
 conversions 133
 resolving 175
 virtual functions 340
AND operator (bitwise) & 120
AND operator (logical) && 121
anonymous
 unions 81
ANSI flagging 221
__ANSI__ macro 205
argc (argument count) 149
argument count (argc) 149
argument vector (argv) 149

- argument-matching conversions 280
- arguments
 - best-matching 279
 - const 281
 - default 156, 294
 - default initializers in templates 347
 - matching 279
 - pass by reference 154
 - template
 - matching 357, 359
 - nontype 354
 - type 351
 - trivial conversions 281
 - virtual functions 339
 - volatile 281
- arguments in a function call 150
- argv (argument vector) 149
- arithmetic
 - conversions 135
 - operands 96
- arrays
 - class members 254
 - declaring 61
 - operands 96
 - subscripting operator
 - overloading 288
 - subscripting operators 102
 - unsized 66
 - zero-sized 66
- arrow operator 103
- ASCII character codes 27
- assignment
 - memberwise 287
 - operators
 - copying classes 312
 - default for classes 312
 - overloading 287
- assignment expression
 - compound 128
 - description 126
 - simple 126
- associativity of operators 93
- auto storage class specifier 32

B

- backslash escape sequence `\\` 27
- backspace escape sequence `\b` 27
- base classes
 - abstract 343
- base classes (*continued*)
 - access 324, 325
 - ambiguities 333, 335
 - description 319
 - direct 318, 332
 - indirect 318, 333
 - initialization 306, 308
 - multiple 332
 - multiple access 335
 - pointers to 321, 322, 328, 338
 - virtual 333, 337
- base list 319, 323, 332
- base specifier 323
- bell escape sequence `\a` 27
- best-matching arguments 279
- binary expression 114
- binary operators
 - addition operator `+` 116
 - bitwise AND operator `&` 120
 - bitwise inclusive OR operator `|` 121
 - bitwise OR operator `^` 120
 - description 114
 - division operator `/` 115
 - equality operator `==` 119
 - greater than operator `>` 118
 - inequality operator `!=` 119
 - left-shift operator `<<` 117
 - less than operator `<` 118
 - logical AND operator `&&` 121
 - logical OR operator `||` 122
 - multiplication operator `*` 115
 - overloading 286
 - remainder operator `%` 115
 - right-shift operator `>>` 117
 - subtraction operator `-` 116
- binding
 - dynamic 318
 - static 318
 - virtual functions 338
- bit fields 74, 254
- block
 - scope 7
 - visibility 7
- block statement 166
- brackets `[]` 102
- break statement 168

C

- C and C++ differences 385
- C++ overview 1
- call scope 329
- call, function 150
- cancel_handler pragma 214
- carriage return escape sequence `\r` 27
- case label 184
- cast
 - argument-matching conversions 280
 - expressions 107
 - function style 297
- catch
 - argument matching 372
 - exceptions 372
 - keyword 368
 - no match 381
- char constant 23
- char type specifier 45
- character
 - constant 23
 - data types 45
 - set 15
 - string constant 24
- chars pragma 215
- class key 241
- class member access operators
 - argument-matching conversions 280
 - description 102
 - overloading 288
- class member lists 253
- class members 253
- class names
 - description 242
 - scope 246
- class object 30
- class operands 96
- class scope 8
- class templates
 - declarations 353
 - definitions 353
 - description 351
 - friends 363
 - instantiation 351
 - member functions 361
 - static members 365
- classes
 - abstract 343
 - aggregate 241, 293
- classes (*continued*)
 - base
 - See base classes
 - class templates 351
 - class-type members 255
 - constructors 293
 - conversions 304
 - copying
 - by assignment 312
 - by initialization 313
 - restrictions 312
 - declarations 241
 - derivation 319
 - derived
 - See derived classes
 - destructors 293
 - friends 273
 - incomplete declarations 247, 255, 319
 - inheritance 315
 - initialization 306
 - local 249
 - member access 270
 - member functions 256
 - member lists 253
 - member scope 258
 - members 253
 - nested 248, 276, 327
 - objects 30
 - overloading
 - functions 277
 - operators 281, 284
 - overview 239
 - scope 246
 - special member functions 293
 - static members 265
 - this pointer 262
 - virtual 338
 - virtual base 333
 - virtual member functions 338
- comma operator 129
- comment pragma 215
- comments 16
- `__COMPAT__` macro 205
- compatible types 124
- complete class name 323
- compound statement 166
- conditional compilation
 - description 207
 - elif preprocessor directive 208
 - if preprocessor directive 208

- conditional compilation (*continued*)
 - ifdef preprocessor directive 209
 - ifndef preprocessor directive 209
- conditional expression ? : 123
- const
 - arguments 281
 - qualifier 84
- constant
 - character 23
 - member functions 256
 - string 24
- constant expression 99
- constants
 - description 20
- construction order
 - of class objects 296
 - of derived class objects 311
- constructors
 - construction order 296, 311
 - conversion by 304
 - copy 295, 312, 313
 - default 294, 309
 - derived class objects 311
 - description 294
 - exception handling 376
 - explicitly constructing objects 297
 - initialization by 306
 - initializer 308
 - overview 293
 - templates 352, 363
 - temporary objects 302
 - virtual 293
- continuation character 24, 27, 192
- continue statement 171
- conversion functions 305
- conversions
 - ambiguous 133
 - arguments 279
 - arithmetic 135
 - cast 107
 - derived class 337
 - sequence 280
 - standard 132
 - trivial 281
 - user-defined
 - by constructor 304
 - conversion functions 305
- copy constructors 295, 312, 313
- copy restrictions 312

- copying class objects 311
- __cplusplus macro 204

D

- data
 - abstraction 2
 - hiding 2
- data members
 - description 254
 - scope 258
 - static 267
- __DATE__ macro 203
- deallocation expression 113
- decimal constant 21
- declarations
 - access 326
 - class
 - description 241
 - incomplete 247, 255, 319
 - syntax 241
 - class member 253
 - class templates 353
 - description 29
 - friend 273, 276
 - function
 - matching 278
 - resolving ambiguities 175
 - function templates 359
 - matching 278
 - overview 5
 - pointers to members 260
 - resolving ambiguous statements 175
 - template functions 359
 - unsubscripted arrays 63
- declarators
 - description 83
 - member 253
- decrement operator -- 104, 290
- default
 - arguments 293, 294
 - assignment operator 312
 - constructors 294, 309
 - copy constructors 313
 - initializers in templates 347
 - member access 271
- default clause 184, 185
- default label 185
- define pragma 216

- #define preprocessor directive 192
- defined unary operator 208
- defined, preprocessor operator 208
- definitions
 - class templates 353
 - description 29
 - function templates 359
 - macro 192
 - member function 256
 - overview 5
 - template classes 353
 - template functions 359
- delete operator
 - description 113
 - free store 299
 - overloading 291
- dereferencing operator 106
- derivation
 - See inheritance*
- derived classes
 - access 324
 - access declarations 326
 - base list 319, 323
 - catch block 372
 - construction order 311
 - description 319
 - initialization 306
 - pointers to 321, 322, 328, 338
 - syntax 323
- destructors
 - description 297
 - destruction order 298
 - exception handling 376
 - overview 293
 - thrown objects 298
 - virtual 293, 297
- differences between C and C++ 385
- direct base class 318, 332
- disable_handler pragma 216
- division operator / 115
- do statement 173
- dominant names 322
- dot operator 102
- double type specifier 48
- dynamic binding
 - in object-oriented programming 3
 - virtual functions 318

E

- EBCDIC character codes 27
- effective access 329
- elaborated type specifier 246
- #elif preprocessor directive 208
- ellipsis
 - argument-matching conversions 280
 - function call operator 288
 - in overloaded operator 284
 - type checking 146
 - user-defined conversions 280
- else clause 180
- #else preprocessor directive 210
- empty argument list 146
- encapsulation 2
- end of string 24
- #endif preprocessor directive 210
- enum 50
- enumeration operands 96
- enumerator 50
- enumsize pragma 217
- equal to operator == 119
- equality operators
 - See also relational operators*
 - equal to == 119
 - not equal to != 119
- error handling
 - See exception handling*
- #error preprocessor directive 201
- escape character \ 27
- escape sequence 27
- evaluation, expression 93
- exception handling
 - access 372
 - catching exceptions 372
 - constructors 376
 - destructors 376
 - exception specifications
 - empty 380
 - unexpected 380
 - order of catching 372
 - overview 367
 - resumption model 370
 - rethrowing exceptions 373
 - special functions 381
 - syntax 368
 - termination model 370
 - throw 298
 - transferring control 370

- exception specification syntax 379
- exception_handler pragma 218
- exclusive OR operator (bitwise) ^ 120
- explicit definitions
 - member function templates 361
 - template classes 356, 365
 - template functions 358
- explicit initialization 306
- explicit type conversions 107
- exponent 22
- expressions
 - allocation 108
 - assignment 126
 - binary 114
 - cast 107
 - comma 129
 - conditional 123
 - constant 99
 - deallocation 113
 - description 93
 - evaluation of 93
 - list 306
 - lvalue 96
 - parenthesized 98
 - pointer to member 123
 - primary 97
 - resolving ambiguous statements 175
 - statement 175
 - throw 114
 - unary 103
- __EXTENDED__ macro 205
- extern declaration 35
- extern storage class specifier 35
- external identifier 18
- external linkage 9
- extraction operator 10

F

- field, bit 74
- file inclusion 201
- __FILE__ macro 204
- file scope 7, 361
- file scope data declarations
 - unsubscripted arrays 63
- float type specifier 48, 50
- floating-point
 - constant 22
 - conversions 133
- for statement 177
- form feed escape sequence \f 27
- formal exception handling 368
- free store
 - delete operator 113
 - description 299
 - new operator 108
- friends
 - access 276
 - description 273
 - member functions 256
 - nested classes 276
 - scope 275
 - templates 363
 - virtual functions 340
- function call operator 288
- function declarator 145
- function scope 7
- function style cast
 - constructing an object 297
- function templates
 - description 356
 - friends 363
 - members 361
- function-like macro 194
- functions
 - argument conversions 281
 - arguments 279
 - body 146
 - calling functions 100, 150
 - conversion 305
 - declarations 138
 - declarator 145
 - default arguments 156
 - definitions 143
 - exception specifications 379
 - friend 273
 - inline 163
 - main 148
 - operator delete() 299
 - operator new() 299
 - overloading 277
 - overview 137
 - parameter 150
 - pointers to 161
 - polymorphic 318
 - prototypes 143
 - prototyping 6
 - return statements 182
 - return values 159

functions (*continued*)
specifiers 90, 163
template 356
virtual 256, 318, 338, 340
void 140

G

global variables 35
goto statement 179
greater than operator > 118
greater than or equal to operator >= 118

H

handler list 368
handling errors
 See exception handling
hexadecimal constant 21
hexadecimal numbers as escape sequences 27
hidden
 names 97, 244, 246
 virtual functions 339
horizontal tab escape sequence `\t` 27

I

identifier linkage 8
identifiers 18
#if preprocessor directive 208
if statement 180
#ifdef preprocessor directive 209
#ifndef preprocessor directive 209
implementation dependency
 allocation of floating-point types 48
 allocation of integral types 49
 bit field length 74
 class member allocation 254
 sign of char 46
 size_t 291, 299
implementation pragma 220
implicit conversions 131
implicit declaration 138
#include preprocessor directive 201
inclusive OR operator (bitwise) | 121
incomplete class declarations 247, 255, 319
increment operator ++ 103, 290
indentation of code 192
indirect base class 318, 333

indirection operator * 106
info pragma 221
inheritance
 See also base classes
 See also derived classes
 design using 318
 graph 318, 333, 335
 in object-oriented programming 3
 multiple 317, 332
 overview 315
 single 315
initial expression 88
initialization
 by constructor
 base classes 308
 explicit 306
 members 308
 by copying 313
 member lists 293
 members 254
 static data members 268
initializers
 constructors 306
 description 88
inline
 functions
 constructors 293
 description 163
 specifiers 90
 keyword 163
 member functions
 description 257
 template 363
_Inline function specifier 163
input operator 10
input/output overview 10
insertion operator 10
instance
 See objects
instantiation
 member function templates 361
 template classes 351, 365
 template functions 360
int constant 20
int type specifier 49
integer
 conversions 132
 data types 49
integral
 operands 96

integral (*continued*)
 promotions 131, 280
internal identifier 18
internal linkage 8

K

keywords
 description 19

L

label statement 165
langlvl pragma 221
left-shift operator << 117
less than operator < 118
less than or equal to operator <= 118
line feed escape sequence \r 27
__LINE__ macro 204
#line preprocessor directive 211
linkage of identifiers 8
linkage specifications 12
linking to non-C++ programs 12
literal 24
local
 classes 249
 scope 7
 type names 250
logical AND operator && 121
logical negation operator ! 105
logical OR operator || 122
long double type specifier 48
long type specifier 49
lvalue 96

M

macro
 definition 192, 194
 invocation 194
macros, predefined 203
main function 148
map pragma 222
mapinc pragma 222
matching arguments
 description 279
 exception handling 372
 template functions 357, 359
member functions
 constant 256

member functions (*continued*)
 constructors 293
 definition 256
 description 256
 destructors 293
 inline 257
 local classes 249
 overloading operators 283
 special 256, 293
 static 269
 templates 361
 this pointer 262, 341
 volatile 256

member lists 241, 253
member of a structure 71

members
 access
 default 271
 inherited 324
 public, private, and protected 271
 arrays 254
 class member access operators 102
 class type 255
 data 254
 declaration 253
 declarator 253
 inherited 319
 initialization 306, 308
 initializer list 293
 of classes 253
 overloading class access operators 288
 pointers to 123, 260
 protected 324
 scope 258
 static 248, 265
 virtual functions 256
memberwise assignment 287
methods
 See member functions
minus, unary operator 105
modifying access 326
modulo operator % 115
multiple
 access 335
 inheritance 317, 332
multiplicative operators
 division / 115
 multiplication * 115
 remainder % 115

N

- names
 - class 242, 246
 - dominant 322
 - hidden 97, 244, 246
 - local type 250
 - types 379
 - nested classes
 - access declarations 327
 - friend scope 276
 - scope 248
 - nested template arguments 352
 - nested try blocks 373
 - new operator
 - description 108
 - free store 299
 - overloading 291
 - newline escape sequence `\n` 27
 - not equal to operator `!=` 119
 - null character `\0` 24
 - NULL pointer 57, 133
 - null statement 182
 - number sign (`#`)
 - preprocessor directive character 192
 - preprocessor operator 198
- ## O
- object-like macro 193
 - objects
 - base class 333
 - class
 - copying 311
 - declarations 244
 - initialization 306
 - construction order 296
 - of class objects 296
 - of derived classes 311
 - constructors 294
 - conversion to 304
 - data abstraction 2
 - description 30
 - destruction order 298
 - destructors 294
 - explicitly constructing 297
 - temporary 160, 294, 298, 302
 - octal constant 22
 - octal numbers as escape sequences 27
 - one's complement operator `~` 105
 - operational descriptor pragma 225
 - `#` operator 198
 - `##` operator 199
 - operator `delete()` function 299
 - operator `new()` function 299
 - operators
 - `->` (arrow) 103
 - `->*` (pointer to member) 123
 - `::` (scope resolution) 97
 - `.` (dot) 102
 - `.*` (pointer to member) 123
 - additive
 - addition operator `+` 116
 - subtraction `-` 116
 - assignment
 - compound 128
 - copying classes 312
 - default 312
 - description 126
 - overloading 287
 - simple `=` 126
 - binary 286
 - bitwise shift
 - left-shift `<<` 117
 - right-shift `>>` 117
 - delete 113
 - equality
 - equal to `==` 119
 - not equal to `!=` 119
 - multiplicative
 - division `/` 115
 - multiplication `*` 115
 - remainder `%` 115
 - new 108
 - overloading
 - arrow 288
 - binary 286
 - description 281
 - dot 288
 - general rules 282
 - increment 290
 - restrictions 284
 - subscripting 288
 - unary 285
 - pointer to member 123, 260
 - preprocessor
 - `#` 198
 - `##` 199
 - primary
 - array subscripting `[]` 102

- operators (*continued*)
 - relational
 - greater than > 118
 - greater than or equal to >= 118
 - less than < 118
 - less than or equal to <= 118
 - scope resolution 320, 329, 336, 340
 - unary
 - address operator & 106
 - bitwise negation operator ~ 105
 - decrement operator -- 104
 - increment operator ++ 103
 - indirection operator * 106
 - logical negation operator ! 105
 - overloading 285
 - sizeof operator 107
 - unary minus operator - 105
 - unary plus operator (+) 104
- OR operator (logical) || 122
- order
 - of catching exceptions 372
 - template declaration 353, 359
- output operator 10
- overloading
 - functions
 - access declarations 328
 - argument matching 279
 - arguments 280
 - declaration matching 278
 - restrictions 278
 - operators
 - argument matching 279
 - assignment 287
 - class member access 288
 - decrement 290
 - delete 291, 300
 - description 281
 - function call 288
 - general rules 282
 - increment 290
 - member functions 283
 - new 291, 299
 - operands 283
 - restrictions 284
 - subscript 288
 - resolution for template functions 357
 - special operators 287
- overriding virtual functions 338, 341
- overview of C++ 1

P

- pack pragma 227
- packed
 - assignments and comparisons 126
 - structures 76, 151
 - unions 80, 151
- _Packed qualifier 86
- page pragma 235
- pagesize pragma 235
- parameter passing 150
- pass by reference 154
- passing a value 151
- passing an address 151
- placement syntax 110, 299
- plus, unary operator 104
- pointer pragma 235
- pointer to member
 - conversions 134
 - declarations 260
 - operators 123
- pointers
 - conversions 133
 - description 55
 - this 262
 - to functions 161
 - to members 123, 260
- polymorphism 318, 321
 - in object-oriented programming 3
- pound sign (#)
 - preprocessor directive character 192
 - preprocessor operator 198
- pragmas
 - cancel_handler 214
 - chars 215
 - comment 215
 - define 216
 - disable_handler 216
 - enumsize 217
 - exception_handler 218
 - implementation 220
 - info 221
 - langlvl 221
 - map 222
 - mapinc 222
 - operational descriptor 225
 - pack 227
 - page 235
 - pagesize 235
 - pointer 235

- pragmas (*continued*)
 - priority 236
 - skip 237
 - strings 237
 - subtitle 238
 - title 238
- precedence of operators 93
- predefined macros 205
 - `__ANSI__` 205
 - `__COMPAT__` 205
 - `__cplusplus` 204
 - `__DATE__` 203
 - description 203
 - `__EXTENDED__` 205
 - `__FILE__` 204
 - `__LINE__` 204
 - `__STDC__` 204
 - `__TEMPINC__` 205
 - `__TIME__` 204
 - `__TIMESTAMP__` 205
- `#` preprocessor directive character 192
- preprocessor operator
 - `#` 198
 - `##` 199
- primary expression 97
- priority pragma 236
- private keyword 271, 325
- program entry point 148
- program linkage 8
- promotions (integral) 131, 280
- protected keyword 271, 310
- protected member access 324
- prototype 143
- public derivation 325
- public keyword 271, 310, 325
- pure specifier 254, 256, 297, 340, 343

Q

- qualified
 - class name 97
 - type name 248
- qualifiers
 - `_Packed` 86
 - `const` 84
 - `volatile` 84
- question mark escape sequence `\?` 27
- quotation mark
 - double quotation escape sequence `\"` 27
 - single quotation escape sequence `\'` 27

R

- reference scope 329
- references
 - conversions 134
 - description 91
 - initialization 91
 - pass by reference 154
 - return types 160
 - temporary objects 302
- register storage class specifier 39
- relational operators
 - See also* equality operators
 - greater than `>` 118
 - greater than or equal to `>=` 118
 - less than `<` 118
 - less than or equal to `<=` 118
- remainder operator `%` 115
- restoring access 326
- resumption model 370
- rethrowing exceptions 373
- return statement 159, 182
- return types
 - description 159
 - references 160
- return values 140, 159
- right-shift operator `>>` 117

S

- scalar operands 96
- scope
 - call 329
 - class names 246
 - description 7
 - friend 275
 - local classes 249
 - member 258
 - nested classes 248
 - reference 329
- scope resolution operator
 - ambiguous base classes 336
 - class member access 329
 - description 97
 - inheritance 320
 - virtual functions 340
- `set_new_handler()` library function 112
- `set_terminate()` library function 381
- `set_unexpected()` library function 381
- shift operators `<<` and `>>` 117

- short type specifier 49
- signal handler 370
- signed char type specifier 45
- signed int 49
- signed long 49
- simple assignment operator = 126
- simple I/O 10
- single inheritance 315
- single quotation escape sequence \ 27
- sizeof operator 107
- skip pragma 237
- space character 192
- special functions
 - member functions 293
 - used in exception handling 381
- special member functions 256
- specifications
 - exception 379
 - linkage 12
- specifiers
 - access 271, 323, 325
 - base 323
 - class 241
 - declaration 253
 - inline 90, 163
 - pure 254, 256
 - virtual 90
- splice preprocessor directive ## 199
- standard conversions 131
- statements
 - block 166
 - break 168
 - continue 171
 - do 173
 - expression 175
 - for 177
 - goto 179
 - if 180
 - labels 165
 - null 182
 - overview 165
 - resolving ambiguities 175
 - return 159, 182
 - switch 184
 - while 189
- static
 - binding 3, 318
 - data members 267
 - initialization of data members 268
 - member functions 269
- static (*continued*)
 - members 248, 265
 - storage class specifier 40
- __STDC__ macro 204
- string
 - constants 24
 - literals 24
- stringize preprocessor directive # 198
- strings pragma 237
- struct type specifier 71
- structures
 - packing
 - using _Packed qualifier 76
 - using #pragma pack 227
 - unsized arrays in 66
 - zero-sized arrays in 66
- subdeclarator 84
- subscript declarator
 - description 84
 - in arrays 62
- subscript operator
 - overloading 288
- subscripts 102
- subtitle pragma 238
- subtraction operator - 116
- switch statement 184

T

- __TEMPINC__ macro 205
- template classes
 - declaration 353
 - definition 353
 - description 351
 - explicit definition 356, 365
 - instantiation 365
- template functions
 - declarations 359
 - definitions 359
 - description 356
 - explicit definition 358
 - grouping definitions of 358
 - instantiation 360
 - overloading resolution 357
- templates
 - argument
 - list 346, 351
 - nested list 351
 - nontype 354
 - class templates 351

- templates (*continued*)
 - constructors 352, 363
 - declaration 346
 - default initializers 347
 - friends 363
 - function templates 356
 - identifier 346
 - member functions 361
 - pragma define 216
 - pragma implementation 220
 - static data members 365
 - syntax 346
- temporary objects 160, 294, 298, 302
- terminate function 381
- termination model 370
- ternary expression ? : 123
- this pointer 262, 280, 341
- throw
 - argument matching 372
 - expression 114, 368, 373
 - keyword 368
 - point 370
 - rethrowing exceptions 373
- __TIME__ macro 204
- __TIMESTAMP__ macro 205
- title pragma 238
- tokens 15, 191
- trigraphs 16
- trivial conversions 281, 357
- try
 - blocks 368
 - keyword 368
 - nested blocks 373
- type checking 146
- type conversions 132
- type names
 - exception specification syntax 379
 - local 250
 - scope 7
- type qualifiers
 - _Packed 86
 - const 84
 - volatile 84
- type specifier
 - (long) double 48
 - char 45
 - enumeration 50
 - float 48
 - int 49
 - long 49

- type specifier (*continued*)
 - short 49
 - union 80
 - unsigned 49
- typedef specifier
 - class declaration 250
 - description 44
 - local type names 250
 - pointers to members 261
 - qualified type name 248
 - restrictions on overloaded functions 278
- types
 - aggregate classes 241
 - conversions 107
 - data abstraction 2

U

- unary expression 103
- unary minus operator - 105
- unary operators
 - address operator & 106
 - bitwise negation operator ~ 105
 - decrement operator -- 104
 - increment operator ++ 103
 - indirection operator * 106
 - logical negation operator ! 105
 - minus 105
 - overloading 285
 - plus 104
 - sizeof operator 107
- unary plus operator (+) 104
- #undef preprocessor directive 197
- unexpected function 381
- unexpected() library function 379
- union specifier 78
- unions
 - anonymous in C++ 81
 - constructors and destructors 293
 - member destructors 297
 - packing
 - using _Packed qualifier 80
 - using #pragma pack 227
 - unsized arrays in 66
 - zero-sized arrays in 66
- unsigned char type specifier 45
- unsigned int type specifier 49
- unsigned long type specifier 49
- unsigned short type specifier 49

- unsigned type specifier 49
- unsized arrays 66
- unsubscripted arrays
 - description 63
 - in structures and unions 66
- user-defined
 - conversions 280, 304
 - types 239

V

- variable arguments 146
 - See also ellipsis
- vertical tab escape sequence `\v` 27
- virtual
 - base classes 333, 337
 - destructors 293
 - function specifier 90
 - functions
 - access 342
 - ambiguous calls to 340
 - description 338
 - dynamic binding 318
 - hidden 339
 - overriding 338, 341
 - pure 343
 - keyword 323
 - member functions 256
- visibility
 - block 7
 - class members 270
 - static functions 8
- void 60
- void function 140
- volatile
 - keyword 281
 - member functions 256
 - qualifier 84

W

- wchar_t 23
- while statement 189
- white space 16, 191, 192, 198
- wide character constant 23

Z

- zero-sized arrays 66