

Note!

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

First Edition (November 1996)

This edition applies to Version 3, Release 7, Modification Level 0, of IBM VisualAge for C++ for AS/400 (Program 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. 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	xi
Programming Interface Information	xi
Trademarks and Service Marks	xi
About This Book	xiii
Who Should Use This Book	xiii
How to Use This Book	xiii
How to Read the Syntax Diagrams	xiii
Highlighting Conventions	xvi
C++ Language Standards and Portability	xvi
How to Get Help	xvi
Getting Help Inside VisualAge for C++ for AS/400	xvii
Getting Help from the Command Line	xvii
Getting Help for a Keyword or Construct	xviii
Online Documents Available in VisualAge for C++ for AS/400	xviii

Part 1. Introducing ILE C++	1
Chapter 1. Introducing the C++ Programming Language	3
Main Features of the C++ Language	3
Introducing the Program Development Process	4
Preparing a Program	4
Compiling Source Files	4
Binding	5
Running	5
Using C++ for Object-Oriented Programming	5
Coding a C++ Program	5
Program Description	6
Program Structure	6
Program Files	8
Chapter 2. Programming in ILE C++	19
Programming Languages Supported by the OS/400 Operating System	19
Creating a Program	20
Managing a Program	21
Calling a Program	22
Debugging a Program	22
Using Bindable APIs	22
Chapter 3. Creating Different Types of ILE Programs	23
Creating an ILE Program	23
Understanding the Effects of the Integrated Language Environment	24
Program calls	24
Data	24

Files	26
Designing ILE Programs	26
Single-Language ILE Programs	26
Mixed-Language ILE Programs	27
ILE Programs that Contain Bound Service Programs	28
OPM—ILE Programs	29

Part 2. Performing I/O and Optimizing Your Programs 31

Chapter 4. Using C or C++ Stream Input and Output	33
Introducing the Integrated File System	33
Root	34
Open Systems	35
Library	35
Document Library Services	36
LAN Server/400	37
Optical Support	38
File Server	38
Introducing Stream Files in the Integrated File System	39
Streams and Database Files	40
Text Streams	41
Binary Streams	41
Working with Text and Binary Streams in the Integrated File System	42
Storing Data as a Text Stream or as a Binary Stream	42
Open Modes for Dynamically Creating Stream Files	44
Understanding Session Input and Output	44
Understanding Stream Buffering	45
Enabling Integrated File System Stream Input and Output	45
Using Stream Input and Output with Other File Systems	46
Chapter 5. Using C Stream I/O or Record I/O Functions	49
Introducing the AS/400 Data Management File System	49
C Streams and File Types	50
File-Naming Conventions	50
Introducing Record Files	51
Introducing Stream Files in the Data Management File System	52
Streams and Database Files	53
Text Streams	53
Binary Streams	53
Working with Text and Binary Streams in the Data Management File System	54
Open Modes for Dynamically Created Stream Files	54
Understanding Session Input and Output	55
Understanding Stream Buffering	56
Opening Text Stream Files	57
Opening Binary Stream Files (character-at-a-time processing)	59
Opening Binary Stream Files (record-at-a-time processing)	63
Using the Open Feedback Area	65
Using the Input and Output Feedback Area	65

Obtaining Additional Information About Program Devices	65
Chapter 6. Improving Program Performance	69
Data Types	69
Avoid Using the volatile Qualifier	70
Replace Bit Fields with Other Data Types	70
Minimize the Use of Static and Global Variables	70
Use the Register Storage Class	70
Classes	70
Performance Measurement	70
Use a Compiler Option to Enable Performance Measurement	71
Exception Handling	71
Reduce Exceptions	71
Turn Off C2M Messages during Record Input and Output	72
Use a Direct Monitor Handler	72
Avoid Percolating Exceptions	72
Function Call Performance	72
Reduce the Number of Function Calls and Function Arguments	72
Input and Output Considerations	73
Use Record Input and Output Functions	74
Use Input and Output Feedback Information	74
Block Records	75
Manipulate the System Buffer	75
Reduce the Number of File Opens and Closes	76
Process Tape Files	76
Use Stream Input and Output Functions	76
Use C++ Input and Output Stream Classes	76
Use Physical Files Instead of Source Physical Files	76
Specify Library Names	76
Pointers	77
Open Pointers	77
Pointer Comparisons	77
Reduce Indirect Access	79
Shallow Copy and Deep Copy	80
Space Considerations	80
Use the Fast Heap for Dynamic Storage	80
Choose Appropriate Data Types	81
Reduce Dynamic Memory Allocation Calls	81
Reduce Space Used for Padding	82
Activation Groups	84
Calling Functions in Other Activation Groups	84
Reducing Program Start-Up Time	84
Program Control	84
Avoid Virtual Functions	84
Use Cheaper Operators	85
Compile-Time Performance Tips	85
AS/400 Back-End Compile-Time Performance Tips	85
Windows Front-End Compile-Time Performance Tips	85

Choosing Compiler Options	86
Run-Time Limits	86

Part 3. Working with AS/400 Files, Devices, Pointers, and Locales 89

Chapter 7. Using Externally Described Files	91
Retrieving External AS/400 File Descriptions	91
Referencing an AS/400 Connection	92
Writing a C++ Program that Retrieves a Record Format Layout	92
Understanding Compiler-Generated Output	94
Header Description	94
Type Definition Structure	95
Level Checking	96
Using Externally Described Physical and Logical Database Files	96
Input	97
Both	97
Key	98
Null-Capable Fields	99
Level Check	100
Generating Typedefs from Logical Database Files	102
Generating Typedefs from Externally Described Device Files	104
Input	105
Output	105
Both	106
Level Check	108
Indicators	108
Using Multiple Record Formats in a Device File	110
Understanding DDS-to-C++ Data Type Mapping	113
Working in Disconnected Mode with External AS/400 File Descriptions	115
Compile Options When Working in Disconnected Mode	116
Converting _DecimalT Template Classes and Zoned Decimal Data	117
Including Externally Described Files	118
Chapter 8. Using AS/400 Database Files and DDM Files	131
Introducing AS/400 Database Files and DDM Files	131
Physical Files and Logical Files	131
Data Files and Source Files	133
Access Paths	133
Deleted Records	137
Locking Records	138
Sharing AS/400 Database Files	138
Null-Capable Fields	139
Processing AS/400 Database Files and DDM Files	140
Opening as Binary Stream Files	140
Opening as Record Files	140
Input and Output Considerations for Binary Stream Files	140
Binary Stream Functions	140
Record Functions	141

Using Commitment Control	143
Blocking Records	148
Chapter 9. Using Device Files	149
Introducing Display Files, ICF Files, and Printer Files	149
Using Indicators	149
Using Major and Minor Return Codes	153
Opening as Binary Stream Files	153
Opening as Record Files	153
Binary Stream Functions	154
Record Functions	154
Using Display Files	154
Using Subfiles	160
Using Intersystem Communication Function (ICF) Files	162
Using Printer Files	168
Introducing Tape Files, Diskette Files, and Save Files	170
Opening as Binary Stream Files	170
Opening as Record Files	170
Using Tape, Diskette and Save Files	171
Blocking Binary Stream Files	174
Using Record Files	174
Binary Stream Functions	175
Record Functions	175
Using the Device Attributes Feedback Area	175
Chapter 10. Using AS/400 Pointers	177
Introducing AS/400 Pointer Types	177
Understanding the Rules for Using Open Pointers	178
Understanding Other Pointer Rules	178
Declaring Pointer Variables	179
Using Pointer Casting	183
Chapter 11. International Locale Support	185
Elements of a Language Environment	185
Locales	186
POSIX Locale Definition and *LOCALE Support	186
Creating Locales	186
Categories Used in a Locale	186
Setting an Active Locale for Your Program	187
Using Environment Variables to Set the Active Locale	188
Locale-Sensitive Run-Time Functions	189

Part 4. Working with AS/400 C++ Features 191

Chapter 12. Calling Conventions	193
Introducing Program and Procedure Calls	193
Calling Programs	194
Calling Procedures	194

Introducing the Call Stack	194
Calling a Program Using a Linkage Specification	196
Calling an ILE Procedure Using a Linkage Specification	197
Passing Parameters	197
Passing Parameters in C++	198
Using Default Parameter Passing Styles	203
Using Operational Descriptors	204
Understanding Data-Type Compatibility	204
Changing the Names of Programs and Procedures	215
Creating C++ Classes for Use in ILE	216
Mapping a C++ Class to a C Structure	216
Using C++ Objects in a C Program	217
Qualifying Library Calls	221
Calling OPM Programs	221
Program Description	221
Program Structure	222
Program Activation	222
Program Files	223
Invoking the ILE-OPM Program	228
Calling ILE Programs	229
Program Description	229
Program Structure	229
Program Activation	230
Program Files	231
Calling an ILE C++ Program	236
Calling an EPM C Program	237
Calling ILE-Bindable APIs	239
Passing Operational Descriptors	242
Chapter 13. Using Templates in C++ Programs	245
Using Template Terms	245
How the Compiler Expands Templates	246
Generating Template Function Definitions	247
Including Defining Templates	248
Including Defining Templates Everywhere	248
Structuring for Automatic Instantiation	249
Manually Structuring for Single Instantiation	253
Chapter 14. Handling Exceptions	255
Introducing Exception Handling	255
Introducing ILE Message Handling	256
Actions Taken when a Run-time Error Occurs	257
Nesting Exceptions	260
Unhandled Exceptions	261
Handling Exceptions in Your Programs	262
Using Try-Catch-Throw	264
Using the C Language Signal Function	273
Using ILE Condition Handlers	283

Using Direct Monitor Handlers	288
Using Cancel Handlers	304
Understanding the Control Boundary in ILE	309
Percolating Exceptions	313
Handling Errors within Your Program in Other Ways	314
Checking the Return Value of a Function	315
Checking the Errno Value	315
Errno Macros	316
Checking the AS/400 System Exceptions for C Stream Files	320
Record Input and Output Error Macro to Exception Mapping	323
Checking the AS/400 System Exceptions for C Record Files	324
Chapter 15. Porting Programs to VisualAge for C++ for AS/400	327
Differences Between C and C++ on AS/400	327
Inter-Language Calls	327
Binary Coded Decimal Class Library for OS/400	328
Header Files that Work with Both C and C++	337
Type Checking	340
Name Mangling	340
File Inclusion	340
Function Prototypes, Declarations and Pointers	340
Character Array Initialization	341
String Literals	342
Integrated File System	342
Set_Terminate is Scoped to an Activation Group	342
Differences Between C++ on Windows and C++ on AS/400	343
IBM Open Class Run-Time Libraries	343
Correspondence Between Windows and OS/400 Objects	344
Inter-Language Calls	344
Record Input and Output	344
Abort	344
User Interface and GUI Applications	344
Externally Described Files	345
Operational Descriptors	345
Exception Handler	345
Pointers	345
Chapter 16. Casting with Run-Time Type Information	347
Introducing RTTI	347
Using C++ Language Defined RTTI	348
The dynamic_cast Operator	348
Dynamic Casts with Pointers	348
Dynamic Casts with References	349
The typeid Operator	350
The type_info Class	351
Using RTTI in Constructors and Destructors	351
Understanding VisualAge for C++ for AS/400 Extensions to RTTI	352
The extended_type_info Class	352

Part 5. Appendix	355
Appendix A. ANSI Notes on Implementation-Defined Behavior	357
Implementation-Defined Behavior Inherited from C	357
Identifiers	357
Characters	357
Strings	358
Integers	358
Floating-Point Values	359
Arrays and Pointers	360
Structures, Unions, Enumerations, Bit-Fields	360
Qualifiers	361
Declarators	361
Statements	361
Preprocessor Directives	361
Library Functions	362
Error Handling	362
Signals	362
Translation	363
Streams and Files	363
Memory Management	364
Environment	365
Localization	365
Time	365
C++ Specific Implementation-Defined Behavior	365
Classes, Structures, Unions, Enumerations, Bit Fields	365
Linkage Specifications	366
Member Access Control	366
Special Member Functions	366
Bibliography	367
The IBM VisualAge for C++ for AS/400 Library	367
Other IBM Publications	367
C and C++ Related Publications	367
Other Books You might Need	367
Non-IBM Publications	367
Index	369

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.

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.

Licenseses 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.

Programming Interface Information

This book is intended to help you create C++ programs using the VisualAge for C++ for AS/400 product. It primarily documents General-Use Programming Interface and Associated Guidance Information provided by the VisualAge for C++ for AS/400 product.

General-Use programming interfaces allow the customer to write C++ programs that obtain the services of the VisualAge for C++ for AS/400 compiler, debugger, browser, editor, IBM Open Class Library, and IBM Access Class Library for OS/400.

Trademarks and Service Marks

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

Application System/400	AS/400
C/400	COBOL/400
DB2/400	IBM
IBMLINK	ILE
Integrated Language Environment	OS/400
Open Class	PROFS
RPG/400	VisualAge
400	

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

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

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 Visual Edge Software, Ltd.

About This Book

This book describes coding techniques such as using templates, signal and exception handling, and application development using AS/400 database files, externally described files, and device files.

The book focuses mostly on the C++ techniques involved, rather than the lower level operating system techniques that are described in the *C++ Language Reference*.

Note: All commands are issued from the Windows workstation unless otherwise stated.

Who Should Use This Book

This book is written for application and systems programmers who want to use VisualAge for C++ for AS/400 to develop C++ applications that run on the AS/400 system from a Windows workstation. You should have a working knowledge of the C or C++ programming language, and of the Windows and OS/400 operating systems and related products, as described in *Installation Guide and Product Overview*.

How to Use This Book

See the *C++ User's Guide* for introductory information on how to use the VisualAge for C++ for AS/400 compiler and tools to edit, compile, bind, debug, and browse your program. Use this book for reference information on the more technical aspects of the compiler and advanced programming techniques.

How to Read the Syntax Diagrams

This book uses two methods to show syntax. One is for commands, preprocessor directives, and statements; the other is for compiler options.

Syntax for Commands, Preprocessor Directives, and Statements

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.

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

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

The symbol indicates the end of a command, directive, or statement.

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

In these diagrams, STATEMENT represents a C or C++ command, directive, or statement.

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

```
STATEMENT required_item
```

Optional items appear below the main path.

```
STATEMENT
        optional_item
```

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.

```
STATEMENT    required_choice1
              required_choice2
```

If the items are optional, the entire stack appears below the main path.

```
STATEMENT
        optional_choice1
        optional_choice2
```

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

```
STATEMENT    default_item
              alternate_item
```

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

```
STATEMENT ← repeatable_item
```

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 (**pragma**).

Variables appear in italicized lowercase letters (*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.

Note: The white space is not always required between tokens, but it is recommended that you include at least one blank between tokens unless specified otherwise.

This diagram shows the syntax for the **#pragma comment** directive. (See the *C++ Language Reference* for information on the **#pragma** directive.)

```
# pragma comment ( compiler
                    date
                    timestamp
                    copyright
                    user
                    , 'token_sequence' )
```

The syntax diagram is interpreted as:

The `(` is the start of the syntax diagram

The symbol `#` must appear first

The keyword **pragma** must follow the `#` symbol

The keyword `comment` must follow the keyword **pragma**

An opening parenthesis must be present

The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`

If the comment type is `copyright` or `user`, and an optional character string follows, a comma must be present after the comment type

A character string must follow the comma

A closing parenthesis is required

The `)` is the end of the syntax diagram

The following **#pragma comment** directives are syntactically correct according to the syntax:

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

Syntax for Compiler Options

Optional elements are enclosed in square brackets []

When you have a list of items from which you can choose one, the logical OR symbol (|) separates the items.

Variables appear in italicized lowercase letters (*num*).

Examples

Syntax Possible Choices

```
/L[+|-]
```

```
    /L
```

```
    /L+
```

```
    /L-
```

```
/Lt"string" /Lt"Listing File for Program Test"
```

Note that, for options that use a plus (+) or minus (-) sign, if you do not specify a sign, the plus is assumed. The /L and /L+ options are equivalent.

Highlighting Conventions

- Bold** Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
- Italics* Identifies parameters whose actual names or values are to be supplied by the programmer. *Italics* 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.

C++ Language Standards and Portability

At this time, there is no universal standard for the C++ language comparable to C standards. However, an ANSI committee is developing a C++ language standard. Its September 17, 1992 working paper, *Draft Proposed American National Standard for Information Systems - Programming Language C++ X3J16/92-0060*, was used as a base document for developing the VisualAge for C++ for AS/400 C++ compiler. The VisualAge for C++ for AS/400 C++ compiler will continue to change its design in accordance with the ANSI standard as it evolves. If you plan to develop portable code, follow this working paper. Do not use the extensions specific to the compiler as described in *C++ Language Reference*. See Appendix A, "ANSI Notes on Implementation-Defined Behavior" on page 357 for information on the C++ working paper.

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 a standard format. See "Getting Help Inside VisualAge for C++ for AS/400" on page xvii for instructions on opening standard format documents from inside VisualAge for C++ for AS/400. See "Getting Help from the Command Line" on page xvii for instructions on opening standard format documents from the command line.

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:

- **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

To get detailed information, open the **Online Information** notebook in the VisualAge for C++ for AS/400 folder. In this notebook there are 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 `\CTTASW\HELP` directory. To view the *C++ Language Reference* make `C:\CTTASW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for AS/400 for `C:`) and enter:

```
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. If you want to read the section on operator precedence in *C++ Language Reference* enter:

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


Getting Help for a Keyword or Construct

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

C++ User's Guide

C++ Programming Guide

C++ Language Reference

IBM Open Class Library User's Guide

IBM Open Class Library Reference

C Library Reference

IBM Access Class Library User's Guide

IBM Access Class Library for OS/400 Reference

IBM Access Class Library for Windows Reference

Part 1. Introducing ILE C++

This part introduces:

- The C++ language features
- The stages of developing a C++ program
- The OPM, ILE and EPM programming languages
- Compiling, binding, debugging and managing C++ programs
- The creation of an ILE program
- The effects ILE has on program calls, data, and files
- The strategies to create ILE single and mixed language programs.

Chapter 1. Introducing the C++ Programming Language

The C++ language provides additional features not found in the C language. These features include templates, keywords and strict type checking.

This section explains:

“Main Features of the C++ Language”

“Introducing the Program Development Process” on page 4

“Using C++ for Object-Oriented Programming” on page 5

Main Features of the C++ Language

The C++ programming language is based on the C language. It is still evolving, and standards for it that are comparable to the ANSI and ISO/IEC C language standards are at the proposal stage. The VisualAge for C++ for AS/400 compiler continues to change its design in accordance with the ANSI standard.

Although C++ is a descendant of the C language, the two languages are not always compatible. For information on compatibility issues between the C++ and C languages, refer to the section on C and C++ Compatibility in *C++ Language Reference*.

In C++, you can develop new data types that contain functional descriptions (member functions) as well as data representations. These new data types are called *classes*. The work of developing such classes is known as *data abstraction*. You can work with a combination of classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can contain (*inherit*) properties from one or more classes. The classes describe the data types and functions available, but they can hide (*encapsulate*) the implementation details from the client programs.

You can define a series of functions with different argument types that all use the same function name. This is called *function overloading*. A function can have the same name and argument types in base and derived classes.

Declaring a class member function in a base class allows you to override its implementation in a derived class. If you use virtual functions, class-dependent behavior may be determined at run time. This ability to select functions at run time, depending on data types, is called *polymorphism*.

You can redefine the meaning of the basic language operators so that they can perform operations on user-defined classes (new data types), in addition to operations on system-defined data types, such as `int`, `char`, and `float`. Adding properties to operators for new data types is called *operator overloading*.

The C++ language provides templates and several keywords not found in the C language. Other features include *try-catch-throw* exception handling, stricter type checking and more versatile access to data and functions compared to the C language.

See the *C++ Language Reference* for a detailed description of the C++ language and its features.

Introducing the Program Development Process

During the development process, an AS/400 program passes through five stages:

1. Preparing
2. Compiling
3. Binding
4. Running
5. Debugging

This process is not continuous. You can compile, correct compile-time errors, modify, and recompile the program several times before you bind it.

Preparing a Program

Preparing a program involves designing and writing source code. See this book and *C++ Language Reference* for more information about these tasks.

Compiling Source Files

Compiling involves translating a C++ program or preprocessed source code into machine instructions. A C++ source file is passed to the cooperative compiler on your Windows workstation. During compilation of this source file, the compiler generates an intermediate file with the extension `qwo`.

If a connection is established with an AS/400 system, this intermediate code is, by default, translated by the compiler back end into a module object (`*MODULE`) and a program object (`*PGM`) on AS/400.

Note: The intermediate file is deleted after it is translated by the compiler.

To compile a C++ program enter `iccas filename.cpp`. The `filename.cpp` is the name of the source file you want to compile. The compiler compiles the file and produces output files.

You can specify many different options. Depending on the compiler options used, several other types of output files can be created. See the *C++ User's Guide* for information on starting the compiler.

Binding

Binding binds one or more modules into a program (*PGM) or a service program (*SRVPGM). Modules written in C++ can be bound to modules written in any other ILE language. C++ programs can use routines from C++ class libraries, C libraries, and any ILE service program. The binder resolves addresses within each module, import requests and export offers between modules that are being bound together. See the *C++ User's Guide* for information on creating a program.

Running

Programs (*PGM) can be run. You can run a program from your Windows workstation. You do not need to sign on to an AS/400 session, unless you want to see the program's output on the terminal or the program requires a terminal. See the *C++ User's Guide* for information on running a program.

Using C++ for Object-Oriented Programming

The features that provide the C++ language with the facilities for object-oriented programming are the ability to:

- Define your own data types, including those that contain both data elements and a set of operations applicable to them (data abstraction)
- Inherit data elements and operations from other data types (inheritance)
- Hide the data-type implementation details so that client programs use a well-defined interface (encapsulation)
- Select functions at run time depending on data types (polymorphism)

Coding a C++ Program

This program shows you the main features and structure of the C++ language. The source code shows these C++ programming techniques:

- Defining an abstract base class
- Deriving classes from a base class
- Deriving classes from multiple base classes (multiple inheritance)
- Defining private and public members, constructors, and functions of a class
- Declaring function prototypes
- Declaring inline functions in a class and in the program
- Overloading functions
- Using virtual functions
- Using the I/O stream library for program output

Program Description

The program produces a company's monthly pay summary. It processes data on four different types of employees:

- Managers on a yearly salary

- Sales managers who get a yearly salary plus a commission for units sold

- Regular employees who are paid an hourly wage

- Sales persons who are paid a commission based on the number of units they sell

The program displays the basic monthly pay for each class of employee, the name and identification number for each employee, and the amount each employee was paid for the month. It calculates the total amount paid in salary, wages, and commission for the month.

Program Structure

The program files are:

- A user-defined header file `compclas.h` containing the class declarations

- A C++ source file `compfunc.cpp` containing class member function definitions

- A C++ source file `comprec.cpp` containing the main program logic

- A system header file `iostream.h` from the Standard Class library containing standard input and output functions.

User-Defined Header File

In the header file `compclas.h` five classes are declared:

- An abstract base class `employee` containing the employee's name and identification number

- Three derived classes `manager`, `regular_emp`, and `sales_person` contain constructor functions and functions for printing out pay information

- A class `sales_mgr` that uses the `pay` functions from both the `sales_person` and `manager` classes in its `pay` function to demonstrate multiple inheritance.

Each class contains private data on salaries, or wages and hours worked, or commission and units sold, and on public functions that use the data. The class structure demonstrates the private and public aspects of a class. Each class contains one inlined function to demonstrate inlining within a class. Figure 1 on page 7 shows the class hierarchy.

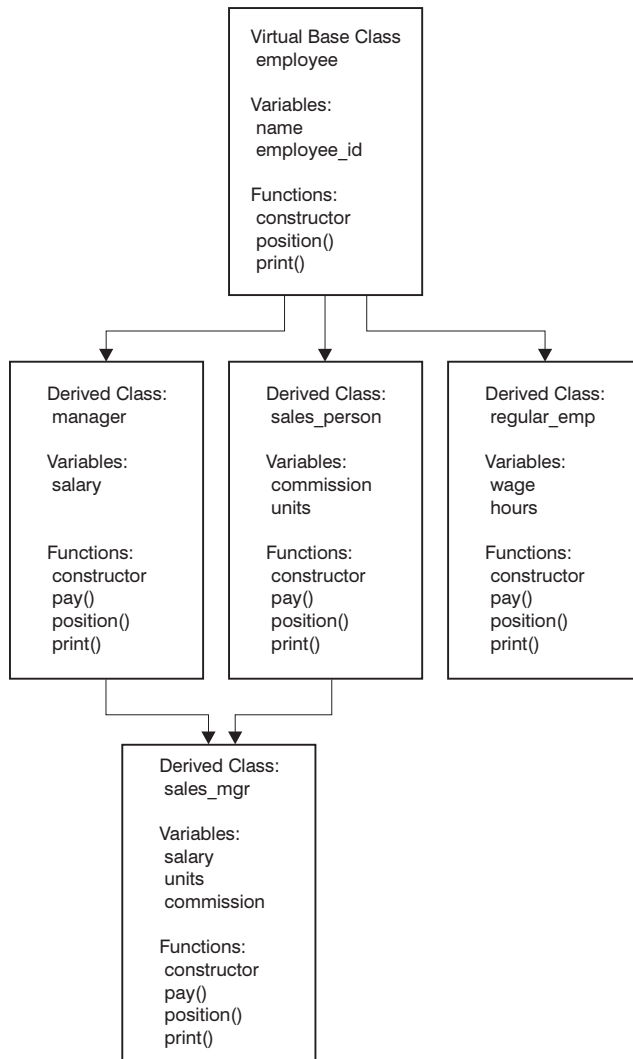


Figure 1. Class Hierarchy in the Example Program

C++ Source Files

The file `compfunc.cpp` contains the definitions for the member functions of the classes used in this program.

The file `comprec.cpp` contains the logic for this program. It defines a function `payout()` that prints out a basic salary for each category of employee.

The function `payout()` demonstrates how you prototype and overload a function. Each class of employee has a different number of parameters, but the same function name is

used for all employee classes. The functions are prototyped before the definition of the `main()` function. They are defined in the body of `comprec.cpp` after the definition of the `main()` function.

The `main()` function in `comprec.cpp` defines an instance of each of the four classes of employee: manager, sales manager, regular employee, and sales person, and uses the functions defined for each of these classes to display information about four typical employees. Figure 2 shows the program structure.

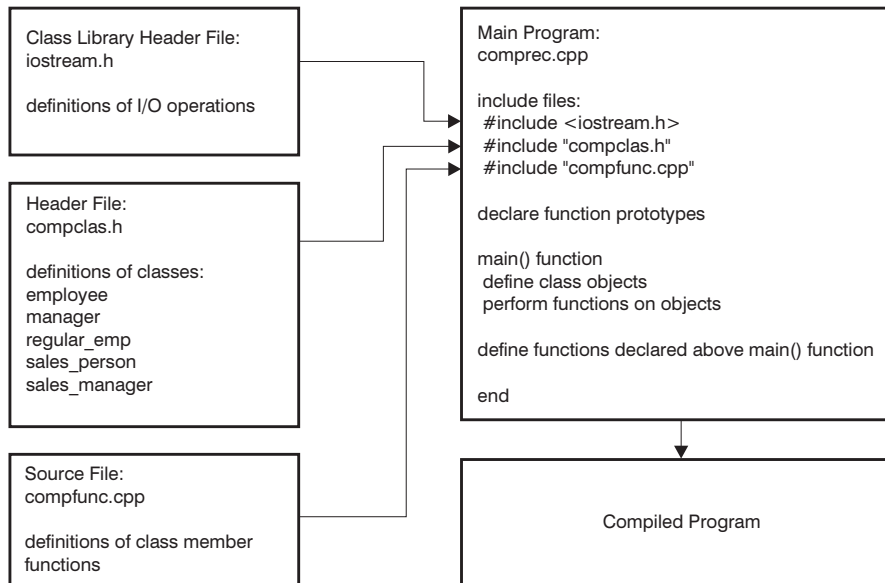


Figure 2. Structure of the Example Program

Program Files

These sections show the source code for each of the files that compose this program.

Header File

The header file `compclas.h` contains definitions of classes that are used in the main program `comprec.cpp`. See “Main Program Source File” on page 13.

```
// compclas.h -- class definitions for comprec.cpp ***

#ifndef _COMPCLAS_H
# define _COMPCLAS_H 1

// Abstract base class definition
class employee {
private:
    char * n;
    int x;
}
```

```

        employee( const employee & lhs);
        employee& operator =( const employee & lhs);

protected:
    const char * name() { return n; }
    int employee_id () { return x; }

public:
    // Constructors for class employee.           3
    employee() : n(""), x(0) {};
    employee(char * n, int id);

    virtual ~employee ();                       4
    virtual double pay() =0;                     5
    virtual void dump() =0;
};
// End of abstract base class definition

// Derived class definitions.                   6

// Derive a class manager from class employee
class manager : virtual public employee {
protected:
    double salary;
    manager(double sal);

public:
    // Constructors for class manager.
    manager(char *n, int id, double sal);

    // Member functions for class manager.
    double pay();
    friend ostream& operator << (ostream& out, manager & lhs);
    void dump() { cout << *this; }             7
};

// End of manager class definition.

// Derive a class regular_emp from class employee
class regular_emp : virtual public employee {
private:
    double wage, hours;

public:
    // Constructor
    regular_emp(char *n, int id, double wg, double hrs) ;
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, regular_emp & lhs);
    void dump() { cout << *this; }
};
// End of regular_emp class definition.

```

```

// Derive a class sales_person from class employee
class sales_person : virtual public employee {
protected:
    double commission;
    int units;
    sales_person(double com, double nts);

public:
    // Constructors
    sales_person(char *n, int id,
                 double com, double nts);
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, sales_person & lhs);
    void dump() { cout << *this; }
};
// End of sales_person class definition.

// Derive a class sales_mgr from the manager and sales_person classes
class sales_mgr : public manager, public sales_person {
public:
    // Constructors
    sales_mgr(char *n, int id, double sal, double comm, double nts);
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, sales_mgr & lhs) ;
    void dump() { cout << *this; }
};
// End of sales_mgr class definition.

#endif

```

Program References

- 1 An abstract base class must have at least one pure virtual function. You cannot declare instances of an abstract base class.
- 2 Classes can have `public`, `private`, and `protected` data members and functions. Data can be `private` (hidden) while the functions that make use of this data are `public`. Hiding data is a way of making the data available through a limited set of functions, while at the same time protecting the data from change by an unauthorized user.
- 3 Constructors have the same name as the class. For virtual base classes with multiple inheritance, you must have one constructor that takes no arguments. You can have as many constructors as you need, but each must have different numbers or types of arguments.
- 4 Destructors have the same name as the class, preceded by a tilde (~) character. The destructor is `~employee`. You cannot specify arguments or a return type for a destructor. The destructor destroys members of the class immediately before the class object is destroyed.

5 Virtual functions defined in a base class of a hierarchy are often not intended to be invoked. They are *pure virtual functions*; functions declared with the keyword *virtual*, which define a return type, and which end with = 0. Each derived class must override the definition of each pure virtual function of its base classes. Each derived class must define a function with the same name and arguments. These functions must return the same type as the corresponding function in the base class.

A class that contains at least one pure virtual function can be used only as a base class for subsequent derivations of other classes.

6 Derived classes inherit the data members and functions from the base class. The colon operator (:), followed by the name of the base class, indicates that the class being defined is a derived class.

7 Functions defined within the class definition are automatically inlined. You do not need the keyword **inline**.

8 A class with multiple inheritance inherits data structures and member functions from more than one class. Class `sales_mgr` has the properties of both a `manager` class and a `sales_person` class. Class `sales_mgr` represents someone who earns both a regular salary (such as a `manager`) and earns a commission (such as a `sales_person`).

C++ Source File

The file `compfunc.cpp` contains the function definitions for the member functions of the classes that are used in the main program `comprec.cpp`. See “Main Program Source File” on page 13.

```
//compfunc.cpp -- definitions for class member functions

#include <string.h>
#include <iostream.h>
#include "compclas.h"

// Constructor definition for class manager.
employee::employee(char * name, int id)           9
: n (new char [strlen (name) + 1 ]), x(id) {      10
    strcpy (n, name);
};
employee::~employee() { delete [] n; }

// Note different way to initialize.
manager::manager(char *n, int id, double sal)
    : employee(n, id), salary(sal){}

manager::manager(double sal)
    : salary(sal){}

// Member function definitions for class manager
double manager::pay() {
    return salary/12 ;
}
```

```

ostream& operator << (ostream& out, manager & lhs) {
    out << endl << lhs.name()
        << " is a manager with employee number " << lhs.employee_id () << endl

        << "makes " << lhs.salary << " per year." << endl

        << lhs.manager::name() << " made " << lhs.pay() << " dollars this month."
        << endl << endl;
    return out;
}

// Constructor for class regular_emp.
regular_emp::regular_emp(char *n, int id, double wg, double hrs)
    : employee(n, id), wage(wg), hours(hrs) {}

// Member function definitions for class reg_emp
double regular_emp::pay() {
    return wage*hours;
}

ostream& operator << (ostream& out, regular_emp & lhs) {
    out << endl << lhs.name() << " is a regular employee with employee number "
        << lhs.employee_id() << endl

        << "makes " << lhs.wage << " dollars per hour" << endl

        << "and worked " << lhs.hours << " hours this month. " << endl

        << lhs.name() <<" made " << lhs.pay()
        << " dollars this month." << endl << endl;
    return out;
}

// Constructor definition for class sales_person.
sales_person::sales_person(char *n, int id,
    double comm, double nts)
    :employee(n, id), commission(comm), units(nts){}

sales_person::sales_person(double comm, double nts)
    : commission(comm), units(nts){}

// Member function definitions for class sales_person.
double sales_person::pay() {
    return commission*units;
}

ostream& operator << (ostream& out, sales_person & lhs) {
    out << endl << lhs.name() << " is a sales person with employee number "
        << lhs.employee_id() << endl

        << "works for a straight commission of "
        << lhs.commission << endl

```

```

        << "dollars per unit sold and sold " << lhs.units
        << " units this month." << endl

        << lhs.name() <<" made " << lhs.pay()
        << " dollars this month." << endl << endl;
    return out;
}
// Constructor for class sales_mgr                                11
sales_mgr::sales_mgr(char *n, int id,
                    double sal, double comm, double nts)
    :employee(n, id),
    manager(sal),
    sales_person(comm, nts) { }

// Member function definitions for class sales_mgr.
double sales_mgr::pay(){                                        12
    return manager::pay() + sales_person::pay()
;}

ostream& operator << (ostream& out, sales_mgr & lhs) {
    out << endl << lhs.name() << " is a sales manager with employee number "
    << lhs.employee_id() << endl

    << "makes " << lhs.salary << " per year and earns a commission of "
    << lhs.commission << " dollars per unit sold." << endl

    << lhs.name() << " was responsible for sales of "
    << lhs.units << " units this month." << endl

    << lhs.name() <<" made " << lhs.pay()
    << " dollars this month."<< endl << endl;
    return out;
}

```

Program References

- 9 The scope resolution operator (::) in this function definition indicates that the constructor function `employee` for class `employee` is defined.
- 10 The purpose of the constructor is to initialize the data types declared in the class definition.
- 11 For classes with multiple inheritance, the constructor must fully initialize the class without creating any ambiguity.
- 12 Because `sales_mgr` is a derived class with multiple inheritance, there can be ambiguity about which values to use. To avoid ambiguity, explicitly qualify the function by using the scope resolution operator (::).

Main Program Source File

The C++ source file `comprec.cpp` contains the logic of this program.

```

//comprec.cpp
#include <iostream.h>                                13
#include <strstrea.h>
#include "compclas.h"
#include "compfunc.cpp" // for the inline functions

static void payout(double);                          14
static void payout(double, double);                  15
static void payout(double, double, double);
inline void title() {                                16
    cout << "Monthly Employee Pay Report" << endl << endl;
};
// Start of main function                            17
const char * pwcharset = "abcdefghijklmnopqrstuvwxy"
                        "ABCDEFGHIJKLMN"
                        "OPQRSTUVWXYZ"
                        "0123456789\r\n";

int main() {
    const double managers_pay = 1510.35;             18
    const double reg_emp_pay = 25.75;
    const double reg_emp_hrs = 40.00;
    const double sales_mgr_units = 200;
    const double monthly_salary = 800.00;
    const double commission = 1.00;
    const double units = 150;
    cout.setf(ios::fixed);                           19
    cout.precision(2);
    title();

    // Optionally, build in a security check          20
    char val [512] ;
    cout << "To view data on employees, "
        << "type your password (mona) and press Enter:" << endl;
    cin >> val ;
    char * p = val;
    char * where;
    int check = 32679;
    while (*p) {
        where = strchr(pwcharset,*p);
        if (where) check *= (where-pwcharset+7);
        ++p;
    }
    check &= 32767;
    if (check != 9196) {
        cout << "Wrong password, bye." << endl;
    // cout << "check is " << check << endl;
        return 999;
    }

    payout(managers_pay);
    payout(reg_emp_pay, reg_emp_hrs);
    payout(monthly_salary, commission, units);

```

```

// Define instances of classes and call their member functions
manager smith("Jack Smith", 123, 28020);           21
cout << smith << endl;

regular_emp james("Everett James", 456, 12,160) ;
cout << james << endl;

sales_person doe("Jackson Doe", 101, 31,65);
cout << doe << endl;

sales_mgr stevens("Jennifer Stevens", 789, 28000, 4, 105) ;
cout << stevens << endl;
double sal1, sal2, sal3, sal4;                    22
sal1 = smith.pay();                               23
sal2 = james.pay();
sal3 = doe.pay();
sal4 = stevens.pay();

// Use the values returned from the pay functions.
cout << "Total wages paid to these employees was: "
      << (sal1+sal2+sal3+sal4)                    24
      << " dollars" << endl << endl ;

// Alternate method of defining instances of a class      25
employee * ep[] = {
    new manager      ("Jack Smith", 123, 28020),
    new regular_emp ("Everett James", 456, 12,160),
    new sales_person ("Jackson Doe", 101, 31,65),
    new sales_mgr    ("Jennifer Stevens", 789, 28000, 4, 105),
    0
};

double sal=0;
for (int i=0; ep[i]; ++i) {
    ep[i]->dump();
    sal+=ep[i]->pay();
    delete ep[i];
}

cout << "Total wages paid to these employees was: "
      << sal
      << " dollars" << endl << endl;
return 0;
}
// End of function main

// Definition of payout functions
// An overloaded function with one, two, and 3 arguments
void payout(double managers_pay) {
    cout << "The basic salary for a manager is: "
          << managers_pay << " dollars per month." << endl << endl;
}

```



```

void payout(double reg_emp_pay, double reg_emp_hrs) {
    double reg_monthly_pay;
    reg_monthly_pay = reg_emp_pay * reg_emp_hrs;
    cout << "The basic pay for a regular employee is: "
         << reg_monthly_pay << " dollars per month." << endl << endl;
}
void payout(double monthly_salary, double commission, double units) {
    double reg_monthly_pay;
    reg_monthly_pay = (monthly_salary + (commission*units));
    cout << "The basic pay for a sales manager is: "
         << reg_monthly_pay << " dollars per month." << endl << endl;
}

```

Program References

13 The first two preprocessor directives let this program make use of system include files that declare standard library functions such as the `cout` and `cin` objects. The compiler searches for the files named in the `#include` directives.

The third and fourth preprocessor directives name the user-defined include file `compclas.h` and the source file `comfunc.cpp` that define the classes and member functions used by `comprec.cpp`.

14 All functions must be declared before they can be used. This statement declares a function `payout` that takes an argument of type `double` and does not return a value.

The function has been declared `static` to limit its scope to this file only. One reason to use the static storage class for functions is to avoid possible name conflicts.

15 An overloaded function has the same name as another function but different types or numbers of arguments.

Each function uses the same type of argument (`double`), but each has a different number of arguments.

16 The keyword **inline** specifies a function whose body is to be expanded at each point of call to the function, so that the call is replaced by the function body itself. Within a class definition, the function is automatically inlined if you include its definition in the class definition. Specifying **inline** may not inline the function. The compiler decides whether the function is inlined.

You can control inlining with the `/Oi+` compiler option.

17 Each program must have a `main` function. Within the braces are statements that make up the main body of the program. `main` takes no arguments and returns an `int`.

18 All variables must be declared before they are used.

Declaring a variable as `const` ensures that the program cannot inadvertently change the value of this variable.

In your program, the values provided here would probably be stored in a database file, since they may be revised more often than the code itself.

19 The `cout` object is part of the I/O stream class library. Its member functions are used to set decimal places for the output.

20 You can code a security feature such as a password.

21 An instance of the class provides a name for the instance and the values required by one of the constructors of the class. An instance `smith` of class `manager` is initialized with the three parameters: `name`, `employee_id`, and `salary`.

22 These variables are declared in the body of the program, because this is where they are used.

23 This is one way to use the return values from the `pay` functions from each class. To streamline the program, you can include the functions in the next step without having to define the four variables `sal1` to `sal4`.

24 Expressions and function calls can be used as input to the `cout` object.

25 As an alternative to creating each instance of a class separately, you can use an array instead.

Instead of calling member functions separately for each member of the array, you can use a `for` loop.

Functions must be defined within a class or at file scope. They must be declared (prototyped) before they are used.

Program Output

The output of the sample program is:

```
Monthly Employee Pay Report
```

```
To view data on employees, type your password (mona) and press Enter:  
The basic salary for a manager is: 1510.35 dollars per month.
```

```
The basic pay for a regular employee is: 1030.00 dollars per month.
```

```
The basic pay for a sales manager is: 950.00 dollars per month.
```

```
Jack Smith is a manager with employee number 123  
makes 28020.00 per year.  
Jack Smith made 2335.00 dollars this month.
```

```
Everett James is a regular employee with employee number 456  
makes 12.00 dollars per hour  
and worked 160.00 hours this month.  
Everett James made 1920.00 dollars this month.
```

```
Jackson Doe is a sales person with employee number 101  
works for a straight commission of 31.00
```

dollars per unit sold and sold 65 units this month.
Jackson Doe made 2015.00 dollars this month.

Jennifer Stevens is a sales manager with employee number 789
makes 28000.00 per year and earns a commission of 4.00 dollars per unit sold.
Jennifer Stevens was responsible for sales of 105 units this month.
Jennifer Stevens made 2753.33 dollars this month.

Total wages paid to these employees was: 9023.33 dollars

Chapter 2. Programming in ILE C++

C++ is one of the programming languages that are supported in the *Integrated Language Environment (ILE)*. VisualAge for C++ for AS/400 supplies the first compiler that allows you to develop C++ programs for AS/400. The compiler is cooperative; you can compile AS/400 C++ programs from your Windows workstation, targeting AS/400.

This section describes:

“Programming Languages Supported by the OS/400 Operating System”

“Creating a Program” on page 20

“Managing a Program” on page 21

“Calling a Program” on page 22

“Debugging a Program” on page 22

“Using Bindable APIs” on page 22

Programming Languages Supported by the OS/400 Operating System

ILE is an approach to programming on AS/400. You can build mixed-language programs that are composed of modules written in any ILE programming language. The ILE family of compilers includes: ILE C++, ILE C, ILE RPG, ILE COBOL, and ILE CL. Table 1 lists the programming languages supported by the OS/400 operating system.

Integrated Language Environment (ILE)	Original Program Model (OPM)	Extended Program Model (EPM)
C++	BASIC (PRPQ)	C
C	CL	FORTRAN
CL	COBOL	PASCAL (PRPQ)
COBOL	PL/I (PRPQ)	
RPG	RPG	

Compared to OPM, ILE provides you with improvements in these areas of program development:

Program creation

Program management

Program calls

Source debugging

Binding APIs

Each of the above areas is explained briefly in these sections and discussed in the *C++ User's Guide*.

Creating a Program

ILE program creation consists of:

1. Compiling source code into modules
2. Binding (combining) one or more modules into a program object.

You invoke the C++ compiler through the `iccas` command. By default this command compiles source code and binds modules into a program. You can specify a compiler option to only create a module object. A module object cannot be run as is; it must first be bound into a program object.

You invoke the binder through a compiler option that includes the Create Program (CRTPGM) command. The program results from binding the module. You can run the program.

Separate compiler options used for compiling source code and binding modules enable you to either reuse a module or update one module without recompiling other modules in the same program. You can create and maintain mixed-language programs because you can combine modules from any ILE language.

You can create a *binding directory* to contain the names of modules and service programs that your ILE C++ program or service program may need. A binding directory can reduce program size because modules or service programs listed in a binding directory are used only if they are needed.

You can bind modules into service programs (*SRVPGM). *Service programs* are a means of packaging callable routines (functions or procedures) into a separately bound program. The use of service programs provides modularity and improves maintainability. You can use off-the-shelf modules developed by third parties or package your own modules for third-party use. Service programs are created by a compiler option that includes the Create Service Program (CRTSRVPGM) command.

Figure 3 on page 21 shows the process of creating an ILE program through compiler and binder invocation.

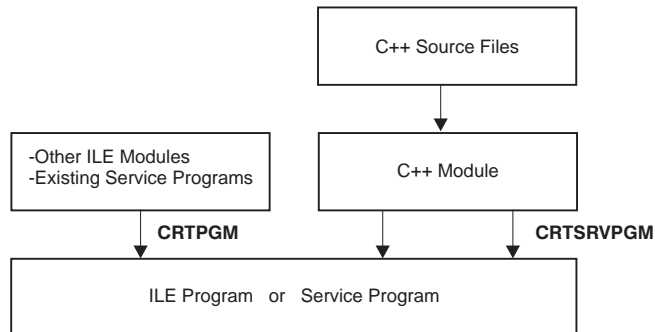


Figure 3. Program Creation in ILE

Once a program is created, you can update it using the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) commands. These commands are useful, because you only need to have the new or changed modules available when you want to update the program.

See the *C++ User's Guide* for information on the compiler.

Managing a Program

ILE provides a common basis for:

- Managing program flow
- Sharing resources
- Using application program interfaces (APIs)
- Handling exceptions during a program's run time

ILE programs and service programs are activated into *activation groups* you specify at the time you create a program. The process of getting a program or service program ready to run is known as activation. Activation allocates resources within a job so that one or more programs can run in that space. When a program is called, the system activates it into an activation group. If the specified activation group for a program does not exist when the program is called, it is created within the job to hold the program's activation.

An activation group is the key element in governing an ILE program's resources and behavior. You can scope commitment-control operations to the activation group level. You can scope file overrides and shared open data paths to the activation group of the running program. The behavior of a program upon termination is affected by the activation group in which the program runs.

See the *C++ User's Guide* for information on managing activation groups.

Calling a Program

In ILE, you can write programs in which ILE C++ programs, OPM and EPM programs interrelate through the use of *dynamic program calls*. When using such calls, the calling program specifies the name of the called program. This name is resolved to an address at run time, just before the calling program passes control to the called program.

You can write programs which interrelate through faster *static procedure calls*. A procedure is a self-contained set of code that performs a task and then returns to the caller. An ILE C++ module consists of one or more procedures. Because the procedure names are resolved at bind time (that is, when you create the program), static calls are faster than dynamic calls.

Static calls allow operational descriptors. Operational descriptors are used to call bindable APIs or procedures written in other ILE languages.

See Chapter 12, "Calling Conventions" on page 193 for information on calls between programs and procedures.

See the *C++ User's Guide* for information on running programs.

Debugging a Program

In ILE, you can perform source-level debugging on any program that is written in one or more ILE language. You can control the flow of a program by using debug commands while the program is running. You can set conditional and unconditional breakpoints prior to running the program. After you call the program, you can step through a specified number of statements and display or change variables. When a program stops because of a breakpoint, a step command, or a run-time error, the pertinent module is shown on the display at the point where the program stopped. At that point, you can enter more debug commands.

See the *C++ User's Guide* for information on the debugger.

Using Bindable APIs

ILE offers a number of bindable APIs that can be used to supplement the functions currently offered by ILE C++. Bindable APIs provide program calling and activation capability, condition and storage management, math functions, and dynamic screen management. The *System API Reference* contains information on bindable APIs.

Chapter 3. Creating Different Types of ILE Programs

In the Integrated Language Environment you can create different types of programs.

If you are new to ILE, you may want to read about basic ILE concepts and terminology in *ILE Concepts*.

This section describes:

“Creating an ILE Program”

“Understanding the Effects of the Integrated Language Environment” on page 24

“Designing ILE Programs” on page 26

Creating an ILE Program

These steps create an ILE program from source code:

1. Create a module from each source member using a command, for example, `iccas` for C++ source, `CRTRPGMOD` for RPG source, or `CRTCLMOD` for CL source.
2. Determine the ILE characteristics for the program:

Determine which module is the starting point for the program. The module you choose as the entry module is the first one that you want to get control. In a C++ program, global objects are constructed first before the module that contains the function `main` is called.

Note: If you are more familiar with OPM languages, consider that in an OPM program, this would be the command-processing program, or the program that is called when a menu item is selected.

Determine the activation group in which the program runs. This can be a named or an unnamed activation group (default `*NEW`). You can specify `ACTGRP(*CALLER)` to run the program in the callers activation group.

Determine the exports and imports to be used.

3. Determine if any of the modules are to combine to create a service program. If so, determine in which activation group the service program runs, then create it using the Create Service Program (`CRTSRVPGM`) command.
4. Bind the modules and service programs into a program using the Create Program (`CRTPGM`) command. Specify values for its parameters based on the characteristics determined in step 2.

A program can be updated easily with the Update Program (`UPDPGM`) or Update Service Program (`UPDSRVPGM`) commands. These commands enable you to add or replace one or more of the modules of a program without having to re-create the program.

Understanding the Effects of the Integrated Language Environment

ILE affects the way your program handles program calls, data, and files.

Program calls

When a new ILE program starts, the system automatically creates an activation group, but not the OPM default activation group, if one does not already exist. The program can contain both dynamic program calls and static procedure calls. Procedures within bound programs call each other through static calls. Procedures call ILE and OPM programs through dynamic calls.

Data

The lifetime of a program's static storage is related to the lifetime of the activation group. A program can run in a user-named activation group (specified with the *ACTGRP(name)* parameter or a system-named activation group (specified with the *ACTGRP(*NEW)* parameter, on the CRTPGM command) or it can be run in the callers activation group if **CALLER* is specified on the *ACTGRP* parameter of the CRTPGM and CRTSRVPGM commands. Storage remains active until the activation group is deleted. See the *ILE Concepts* for information on deleting named activation groups.

A program created with a system-named activation group (created with `iccas /B"CRTPGM grpag ACTGRP(*NEW)" grpag.cpp`), results in a return statement from the program which deletes the activation group. A program created with a user-named activation group (created with `iccas /B"CRTPGM grpag ACTGRP(ACT1)" grpag.cpp`), results in a return statement from the program which does not delete the activation group. You can delete the user-named activation group after the program returns by using the Reclaim Activation Group (RCLACTGRP) command or you can call the `exit()` function.

```

#include <stdlib.h>
#include <fstream.h>
class A {
    fstream out;
    static int count;
public:
    A () {
        system("mkdir '/tmp'");
        system("dltf qtemp/log ");
        system("crtpf qtemp/log rcdlen(1024)");
        out.open("/tmp/log", ios::out | ios::trunc);
        out << "Object constructed" << endl;
    }
    ~A () {
        out << "Object destructed, display called " <<
            count << " time(s)" << endl;
        out.close();
        system("CPYFRMSTMF FROMSTMF('/tmp/log') "
            " TOMBR('qsys.lib/qtemp.lib/log.file/log.mbr') "
            " MBROPT(*REPLACE) " );
        system("dsppfm qtemp/log");
        system("dltf qtemp/log");
    }
    void display (const char * a) {
        out << a << endl;
        ++count;
        if ( 1 < count) {
            out << "Program built with named activation group, "
                "self destruct!" << endl;
            exit(999);
        }
    }
};
int A::count = 0;

A x;
int main(void) { x.display("call display in main"); return 0; }

```

After you create the program `grpqg` with a system-named activation group, the program is activated, the `main()` function is called, but the return from `main()` deletes the activation group:

```

Object constructed
call display in main
Object destructed, display called 1 time(s)

```

After you create the program `grpqg` with a user-named activation group, the program is activated and `main()` is called. If you run the program again, the following results:

```

call display in main
call display in main
Program built with named activation group, self destruct!
Object destructed, display called 2 time(s)

```

You can delete the named-activation group by using the command `call grpag parm(exit)`. The message `Activation group deleted` appears on the display.

Program data identified as exported or imported (using the keywords `EXPORT` and `IMPORT` respectively) is external to the individual modules. This external data is available to the modules that are bound into a program.

Files

By default, file processing (including opening, sharing, overriding, and commitment control) by the system is scoped to the activation-group level. You cannot share open data paths at the data-management level with programs running in different activation groups. If you want to share an open data path across activation groups, you must open a file at the job level by specifying `SHARE(*YES)`, `OVRSCOPE(*JOB)`, and `OPNSCOPE(*JOB)` on an override command, or create the file with `SHARE(*YES)`.

Designing ILE Programs

In the Integrated Language Environment you can build different types of programs:

“Single-Language ILE Programs”

“Mixed-Language ILE Programs” on page 27

“ILE Programs that Contain Bound Service Programs” on page 28

“OPM—ILE Programs” on page 29

Single-Language ILE Programs

In ILE you can compile multiple C++ source files into modules and bind them into a program `y` which is called by an ILE C++ program `x`. Figure 4 on page 27 shows the run-time view of such a single-language program.

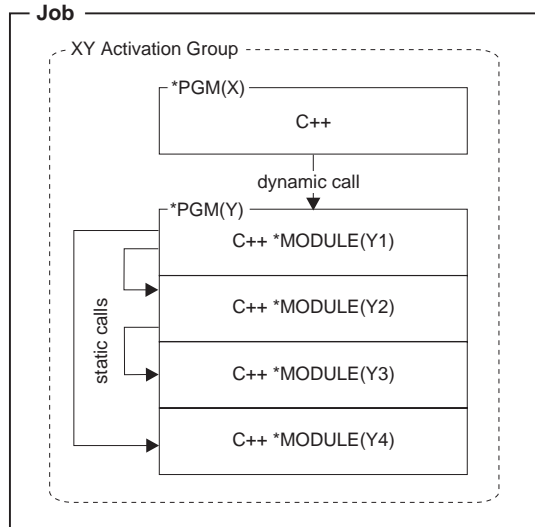


Figure 4. Single-Language ILE C++ Program

The call from program X (*PGM(X)) to program Y (*PGM(Y)) is a dynamic call. The calls among the modules in program Y (Y1, Y2, Y3, and Y4) are static calls.

Mixed-Language ILE Programs

In ILE, you can create a mixed-language program. The main module, written in one ILE language, can call procedures written in another ILE language. If you use different languages, you might not normally expect consistent behavior in the way your program handles calls, data, files, and errors. ILE was developed to ensure such consistency.

Figure 5 on page 28 shows the run-time view of an program containing a mixed-language ILE program in which one module calls a nonbindable API, QUSCRTUS (Create User Space).

The call from program Y to the OPM API (QUSCRTUS) is a dynamic call. The calls among the modules in program Y are static calls.

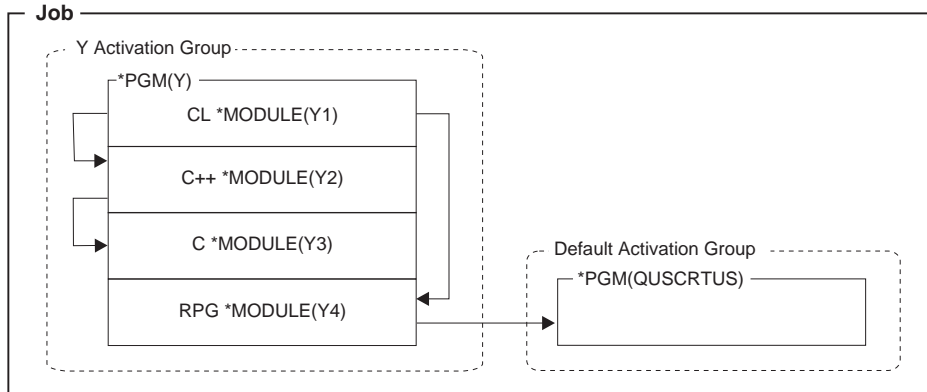


Figure 5. Mixed-Language ILE Program Using CL, C++, C, RPG, and an API

ILE Programs that Contain Bound Service Programs

Figure 6 shows the ILE program *x* calling two service programs *y* and *z*. Service program *z* contains frequently called functions within modules *z1* and *z2*. The use of static procedure calls within modules and service programs provides improved performance, especially if the service program runs in the same activation group as the calling program.

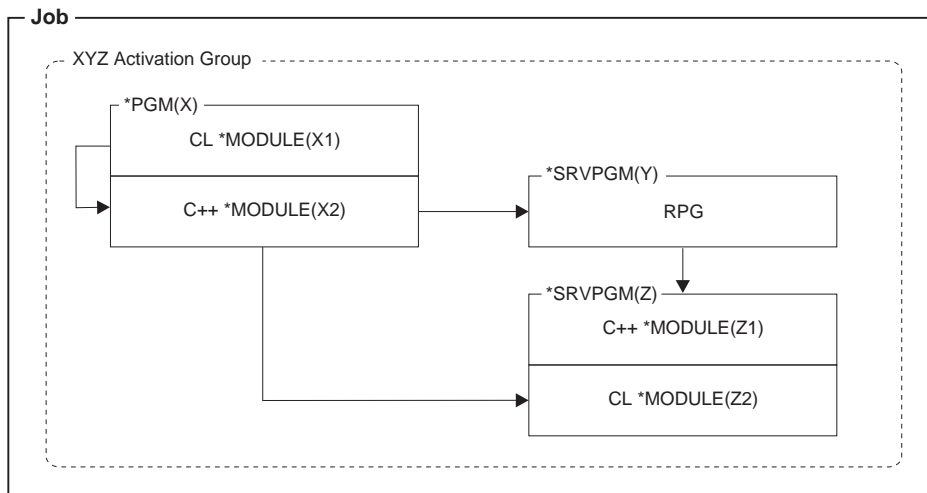


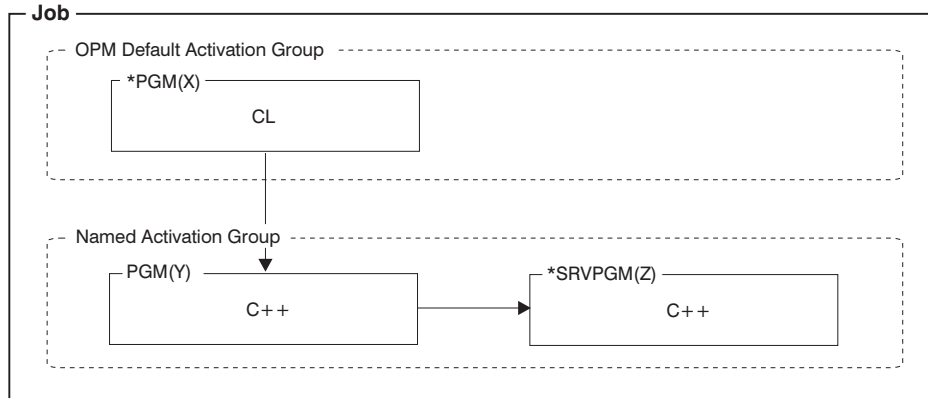
Figure 6. An ILE Program that Binds Two Service Programs

The calls from program *x* to programs *y* and *z* are static calls.

OPM—ILE Programs

ILE provides many alternatives for creating programs, but not all are equally easy to handle. You should avoid a situation in which a program consisting of OPM and ILE programs is split across the OPM default activation group and a named ILE activation group, as shown in Figure 7.

Figure 7. OPM-ILE Program Scenario. The program is split between the OPM default activation group and a named ILE activation group.



When you split programs across the OPM default activation group and any named activation group, you are mixing OPM behavior with ILE behavior. Programs in the default activation group may be expecting the ILE programs to free their resources when the program ends. These resources are not freed until the activation group ends.

The scope of overrides and shared open data paths is more difficult to manage when a program is split between the default activation group and a named group. By default, the scope for the named activation group is at the activation-group level, but for the default activation group, the scope is either call level or job level, not activation-group level.

Under certain circumstances, you may need to run your OPM program in the default OPM activation group. Run a new ILE program in a new activation group and a service program in the caller's activation group, to keep the overhead associated with frequent calls low, and avoid reinitialization of static variables. To ensure correct program termination and the cleanup of associated storage, include a routine in your C++ program that, when called by the OPM program, performs the `exit()` function and all necessary cleanup.

Part 2. Performing I/O and Optimizing Your Programs

This part explains how to:

- Store and operate on information in stream files

- Process stream files as true text or binary stream files

- Open text and binary stream files

- Perform I/O operations a record at a time

- Use open feedback and I/O feedback areas

- Optimize your program

Chapter 4. Using C or C++ Stream Input and Output

The integrated file system is optimized for input and output of stream data. The data management file system is optimized for input and output of records. The C++ compiler uses by default the integrated file system.

This section describes:

“Introducing the Integrated File System”

“Introducing Stream Files in the Integrated File System” on page 39

“Working with Text and Binary Streams in the Integrated File System” on page 42

Introducing the Integrated File System

The integrated file system provides a common interface to store and operate on information in stream files. Stream files include PC files, files in UNIX systems, LAN server files, AS/400 files and folders. The *Integrated File System Introduction* contains information about the commands, displays, and C language program programming interfaces (APIs) used to access the integrated file system.

The C stream I/O functions in VisualAge for C++ for AS/400 are implemented through the data management file system and the integrated file system. The C++ stream I/O classes in VisualAge for C++ for AS/400 are implemented through the integrated file system. You must know the integrated file system to use the integrated file system C stream I/O functions and the C++ stream I/O classes.

If you have existing programs that use AS/400 system files, you need to understand the limitations of the QSYS.LIB file system that is part of the integrated file system. If you have new programs, you can use the other file systems within the integrated file system that do not have the file-handling restrictions of the QSYS.LIB file system.

There are seven file systems in the integrated file system (see Figure 8 on page 34):

“Root” on page 34

“Open Systems” on page 35

“Library” on page 35

“Document Library Services” on page 36

“LAN Server/400” on page 37

“Optical Support” on page 38

“File Server” on page 38

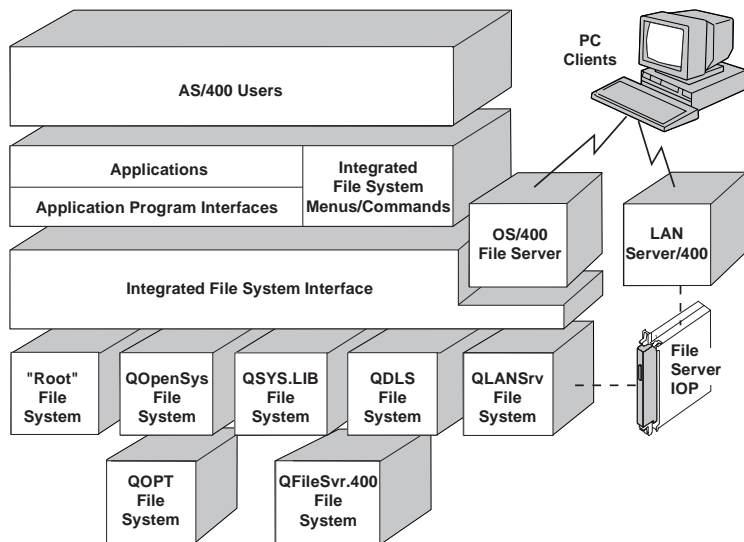


Figure 8. The Integrated File System Interface

A file system provides the support that allows programs to access specific segments of storage that are organized as logical units. These logical units are files, directories, libraries, and objects.

Users and programs can interact with any of the file systems through a common *integrated file system interface*. This interface is optimized for input/output of stream data, in contrast to the record input/output provided through the data management interfaces. The common integrated file system interface includes a set of user interfaces (commands, menus, and displays) and APIs.

Root

The "root" (/) file system can be accessed only through the integrated file system interface. You work with the "root" (/) file system using integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

The "root" (/) file system is designed to take full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.

Path Names

This file system preserves the same uppercase and lowercase form in which object names are entered, but no distinction is made between uppercase and lowercase when the system searches for names.

Path names have the form `Directory/Directory/ . . . /Object`

Each component of the path name can be up to 255 characters long; the path can be up to 16MB

There is no limit on the depth of the directory hierarchy other than program and space limits

The characters in names are converted to Universal Character Set 2 (UCS2) Level 1 form when the names are stored

Open Systems

The open systems (QOpenSys) file system is designed to be compatible with UNIX-based open system standards, such as POSIX and XPG. Like the "root" (/) file system, it takes advantage of the stream file and directory support provided by the integrated file system. It supports case-sensitive object names.

QOpenSys can be accessed only through the integrated file system interface. You work with QOpenSys using integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

Path Names

Unlike the QSYS.LIB, QDLS, QLANSrv, and "root" (/) file systems, the QOpenSys file system distinguishes between uppercase or lowercase characters when searching object names.

The path names, link support, commands, displays, ANSI stream I/O functions and system APIs are the same as defined under the "root" (/) file system.

Library

The library (QSYS.LIB) file system supports the AS/400 library structure. It provides access to database files and all of the other AS/400 object types that are managed by the library support.

The QSYS.LIB file system maps to the AS/400 file system. Those of you who use languages such as ILE C and facilities such as DDS to develop programs are already using the QSYS.LIB file system.

File-Handling Restrictions

There are some limitations in using the library file system under the integrated file system:

- Logical files are not supported

- The only types of physical files supported are program-described files containing a single field and source physical files containing a single text field

- Byte-range locking is not supported; see the *System API Reference*

- Only one job is given write access to a database file member at any time; other jobs are allowed only read access to that member

Path Names

The QSYS.LIB file system does not distinguish between uppercase and lowercase names of objects. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

If the name is enclosed in quotation marks, the case of each character in the name is preserved. The search is sensitive to the case of characters in quoted names.

Each component of the path name must contain the object name followed by the object type:

```
/QSYS.LIB/QGPL.LIB/PRT1.OUTQ  
/QSYS.LIB/PAYROLL.LIB/PAY.FILE/TAX.MBR
```

The object name and object type are separated by a period (.). Objects in a library can have the same name if they are different object types, so the object type must be specified to uniquely identify the object.

The object name in each component can be up to ten characters long, and the object type can be up to six characters long.

The directory hierarchy within QSYS.LIB can be either two or three levels deep (have two or three components in the path name), depending on the type of the object being accessed. If the object is a database file, the hierarchy can contain three levels (library, file, member); otherwise, there can be only two levels (library, object). The combination of the length of each component name and the number of directory levels determines the maximum length of the path name.

If "root" (/) and QSYS.LIB are included as the first two levels, the directory hierarchy for QSYS.LIB can be four or five levels deep.

The characters in names are converted to code page 037 when the names are stored. Quoted names are stored using the code page of the job.

Document Library Services

The document library services (QDLS) file system supports the folder objects. It provides access to documents and folders.

To work with the QDLS file system through the integrated file system interface, use the integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

All users working with objects in QDLS must be enrolled in the system distribution directory.

Path Names

QDLS does not distinguish between uppercase and lowercase in the names containing only the alphabetic characters *a* to *z*. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase. Other characters are case sensitive and are used as is.

Each component of the path name can consist of just a name `/QDLS/FLR1/DOC1` or a name plus an extension `/QDLS/FLR1/DOC1.TXT`.

The name in each component can be up to eight characters long, and the extension can be up to three characters long. The maximum length of the path name is 82 characters.

The directory hierarchy below `/QDLS/` can be 32 levels deep.

The characters in names are converted to code page 500 when the names are stored. A name may be rejected if it cannot be converted to code page 500. The *CL Reference* contains information about the naming rules for document library objects.

LAN Server/400

The LAN Server/400 (QLANSrv) file system provides access to the same directories and files that are accessed through the LAN Server/400 licensed program. It allows users of the OS/400 file server and AS/400 programs to use the same data as LAN Server/400 clients.

To work with the QLANSrv file system through the integrated file system interface, use the integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

Files and directories in the QLANSrv file system are stored and managed by a LAN server that is based on the OS/2 LAN server. This LAN server does not support the concept of a file or directory owner or owning group. File ownership cannot be changed using a command or an ANSI stream I/O function and system API. Access is controlled through *access control lists*. You can change these lists using the WRKAUT and CHGAUT commands. The *Integrated File System Introduction* contains information on these commands.

Path Names

The file system preserves the same uppercase and lowercase form in which object names are entered, but no distinction is made between uppercase and lowercase when the system searches for names. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

Path names have the form `Directory/Directory/ . . . /Object`

Each component of the path name can be up to 255 characters long

The directory hierarchy within QLANSrv can be 127 levels deep; if all components of a path are included as hierarchy levels, the directory hierarchy can be 132 levels deep

Names are stored in the code page that is defined for the file server

Optical Support

The QOPT file system can be accessed through the integrated file system interface using either the OS/400 file server or the integrated file system commands, user displays, ANSI stream I/O functions and system APIs.

Path Names

QOPT converts the lowercase English alphabetic characters *a* to *z* to uppercase when used in object names. A search for object names using only those characters is not case-sensitive.

The path name must begin with a slash (/) and contain no more than 294 characters. The path is made up of the file system name, the volume name, the directory and subdirectory names, and the file name:

```
/QOPT/VOLUMENAME/DIRECTORYNAME/SUBDIRECTORYNAME/FILENAME
```

The file system name QOPT is required.

The volume name is required and can be up to 32 characters long.

One or more directories or subdirectories can be included in the path name, but none are required. The total number of characters in all directory and subdirectory names, including the leading slash, cannot exceed 63 characters. Directory and file names allow any character except hexadecimal value 0x00 through 0x3F, 0xFF, 0x80, lowercase alphabetic characters and these characters:

- Asterisk (*)
- Hyphen (-)
- Question mark (?)
- Quotation mark (')
- Greater than (>)
- Less than (<)

The file-name is the last element in the path name. The file name length is limited by the directory-name length in the path. The directory name and file name combined cannot exceed 256 characters including the leading slash.

The characters in names are converted to code page 500 within the QOPT file system. A name may be rejected if it cannot be converted to code page 500. Names are written to the optical media in the code page specified when the volume was initialized. The *National Language Support Guide* contains information about code pages.

The *Optical Support* contains details on path name rules in the QOPT file system.

File Server

The QFileSvr.400 file system can be accessed through the integrated file system interface using either the OS/400 file server or the integrated file system commands, user displays, ANSI stream I/O functions and system APIs. The characteristics of the

QFileSvr.400 file system are determined by the characteristics of the file system being accessed on the target system.

Path Names

For a first-level directory, which actually represents the "root" (/) directory of the target system, the QFileSvr.400 file system preserves the same uppercase and lowercase form in which object names are entered. No distinction is made between uppercase and lowercase when QFileSvr.400 searches for names.

For all other directories, case sensitivity is dependent on the specific file system being accessed. QFileSvr.400 preserves the same uppercase and lowercase form in which object names are entered when file requests are sent to the OS/400 file server.

Path names have the form

```
/QFileSvr.400/RemoteLocationName/Directory/Directory . . . /Object
```

The first-level directory `RemoteLocationName` represents both:

- The name of the target system that is used to establish a communications connection. The target system name can be either:
 - A TCP/IP host name (`beowulf.newyork.corp.com`)
 - An SNA LU 6.2 name (`appn.newyork`)
- The "root" (/) directory of the target system

When a first-level directory is created using an integrated file system interface any specified attributes are ignored.

Note: First-level directories are not persistent across IPLs. That is, the first-level directories must be created again after each IPL.

Each component of the path name can be up to 255 characters long. The full path name can be up to 16MB long. The file system in which the object resides may restrict the component length and path name length to less than the maximum allowed by QFileSvr.400.

There is no limit to the depth of the directory hierarchy, other than program and system limits and any limits imposed by the file system being accessed.

The characters in names are converted to UCS2 Level 1 form when the names are stored.

Introducing Stream Files in the Integrated File System

The compiler allows your program to process stream files as true text or binary stream files (using the default integrated file system enabled stream I/O) or as simulated text and binary stream files (using the data management system stream I/O). When writing a program that uses stream files, for better performance, use the default integrated file system.

Streams and Database Files

On the integrated file system, a stream is a continuous string of characters. A database file is record-oriented; it has predefined subdivisions consisting of one or more fields that have specific characteristics, such as length and data type. Figure 9 compares a stream file and a record-oriented database file.

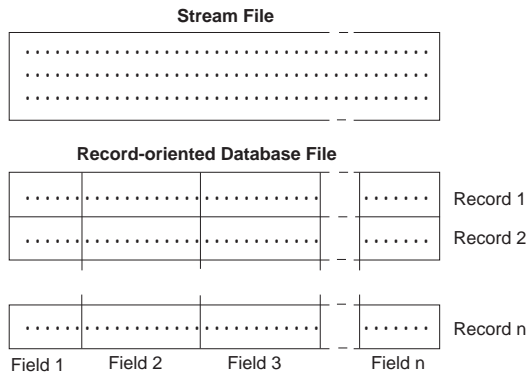


Figure 9. Comparison of a Stream File and a Record-Oriented Database File

The default stream I/O for programs compiled with ILE C is simulated on top of an AS/400 database file. This is simulated stream file processing with AS/400 records. Figure 10 shows how an AS/400 record is mapped to a C stream.

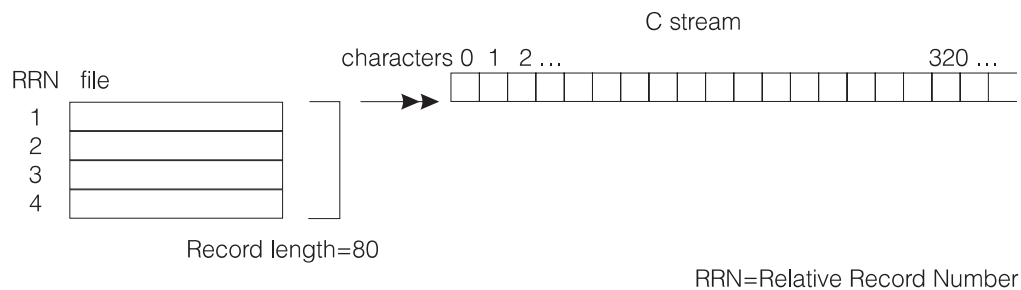


Figure 10. AS/400 Records Mapping to a C Stream File

The differences in the structure of stream files and record-oriented database files affects how a program is written to interact with them and which type of file is best for a program. A *record-oriented file* can store customer information, such as name, address, and account balance. These fields can be individually accessed and manipulated using the programming facilities of AS/400. A *stream file* can store information such as a customer's picture, which is composed of a continuous string of bytes representing variations in color. Stream files can also store strings of data such as the text of a document, images, audio, and video.

Text Streams

Text streams contain printable characters and control characters organized into lines. Each line consists of zero or more characters and ends with a new-line or escape character (`\n`). A new-line character is not automatically appended to the end of file.

The `/ASi` compiler option changes the value for the `\n` escape character value to the `0x25` line feed character. If the `/ASi-` compiler option is specified, the `\n` escape character has a value of `0x15`.

The compiler run time may add, alter, or ignore some special characters during input or output so as to conform to the conventions for representing text in the AS/400 environment. There may not be a one-to-one correspondence between characters written to a file and characters read back from the same file.

Data read from an integrated file system text stream is equal to the data that was written if the data consists only of printable characters and the horizontal tab, new-line, vertical tab, and form-feed control characters.

For most integrated file system stream files, a line consists of zero or more characters and ends with a carriage-return new-line character combination. The integrated file system can have logical links to files on different systems that may use a single line feed as a line terminator. A good example of this are the files on most UNIX systems.

On input the default in text mode is to strip all carriage returns from new-line carriage-return character combination line delimiters. On output, each line-feed character is translated to a carriage-return character followed by a line-feed character. The line-terminator character sequence can be changed with the `crln` parameter on `fopen()`.

When a file is opened in text mode, there may be code page conversions on data processed to and from that file. When the data is read from the file, it is converted from the code page of the file to the code page of the program, job, or system receiving the data.

When data is written to an AS/400 file, it is converted from the code page of the program, job, or system to the code page of the file. For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are converted from one code page to another.

When reading from QSYS.LIB files, end-of-line characters (carriage return and line feed) are appended to the end of the data returned in the buffer.

The code page conversion completed when a text file is processed can be changed by specifying the `codepage` or `ccsid` parameter on `fopen()`. The default is to convert all data read from a file to the job's CCSID or code page.

Binary Streams

A *binary stream* is a sequence of characters that has a one-to-one correspondence with the characters stored in the associated AS/400 system file. The data is not altered on

input or output. The data read from a binary stream is equal to the data that was written except for files in QSYS.LIB of which the size must be a multiple of the record length.

New-line characters have no special significance in a binary stream. The data is treated like any other data. The program must handle the data.

Working with Text and Binary Streams in the Integrated File System

Each text stream file and each binary stream file is represented by a file control structure of type `FILE`. This structure is initialized depending on the mode in which the file was opened. Unpredictable results may occur if you attempt to change the file control structure.

The format of the `fopen()` function is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The *mode* variable is a character string that consists of an open *mode* which may be followed by *keyword* parameters. The open *mode* and *keyword* parameters must be separated by a comma or one or more blank characters.

To open a text stream file use the `fopen()` function with a mode of `r`, `w`, `a`, `r+`, `w+`, or `a+` and the optional keyword parameters *ccsid*, *codepage*, and *crlf*.

To open a binary stream file use the `fopen()` function with a mode of `rb`, `wb`, `ab`, `rb+` or `r+b`, `wb+` or `w+b`, `ab+` or `a+b` and the optional keyword parameters *type*, *ccsid*, *codepage*, and *crlf*.

See Table 27 on page 318 for a list of `errno` values for integrated file system enabled C stream I/O.

Storing Data as a Text Stream or as a Binary Stream

If two streams are opened, one as a binary stream and the other as a text stream, and the same information is written to both, the contents of the streams may differ.

This program shows two streams of different types. The hexadecimal values of the resulting files (which show how the data is actually stored) are not the same.

```

#include <stdio.h>
#include <assert.h>

void main(void)
{
    FILE *fp1, *fp2;
    char lineBin[15], lineTxt[15];
    int x;
    int read1, read2;

    fp1 = fopen("/script.bin","wb");
    assert(fp1);
    fprintf(fp1,"hello world\n");

    fp2 = fopen("/script.txt","w");
    assert(fp2);
    fprintf(fp2,"hello world\n");

    fclose(fp1);
    fclose(fp2);

    fp1 = fopen("/script.bin","rb");
    assert(fp1);

    /* opening the text file as binary to suppress
    the conversion of internal data */
    fp2 = fopen("/script.txt","rb");
    assert(fp2);

    read1 = fread(lineBin, 1,15, fp1);
    read2 = fread(lineTxt, 1,15, fp2);

    printf("Hex value of binary file = ");
    for (x=0; x<read1; ++x)
        printf("%02x", lineBin[x]);

    printf("\nHex value of text file = ");
    for (x=0;x<read2;++x)
        printf("%02x", lineTxt[x]);

    printf("\n");

    fclose(fp1);
    fclose(fp2);

    // The output is:
    //
    // Hex value of binary file = 888593939640a69699938425
    // Hex value of text file = 888593939640a6969993840d25
}

```

As the hexadecimal values of the file contents show, in the binary stream (`script.bin`), the new-line character is converted to a line feed hexadecimal value (`0x25`), while in the text stream (`script.txt`), the new-line is converted to a carriage-return line-feed hexadecimal value (`x0d25`).

Open Modes for Dynamically Creating Stream Files

If you specify `w`, `a`, `w+`, `a+`, `wb`, `ab`, `wb+`, `ab+`, `w+b`, or `a+b` as the mode when opening a file, the AS/400 system automatically creates the file if the file does not already exist. If the file name is directory qualified, the file is created in that directory. If a directory is not specified, the file is created in the current directory.

Understanding Session Input and Output

When a program starts three text streams are defined:

- Standard input `stdin` for reading input from the terminal
- Standard output `stdout` for writing output to the terminal
- Standard error `stderr` for writing diagnostic output to the terminal

Streams `stdin`, `stdout`, and `stderr` are implicitly opened the first time they are used.

- Stream `stdin` is opened with `fopen("stdin", "r")`
- Stream `stdout` is opened with `fopen("stdout", "w")`
- Stream `stderr` is opened with `fopen("stderr", "w")`

These streams are not real AS/400 system files. They are simulated as files by the C library routines and by default they are directed to the terminal session.

The `stdin`, `stdout`, and `stderr` streams can be associated with other devices by using the OS/400 override commands on the files `STDIN`, `STDOUT`, and `STDERR` respectively. If `stdin`, `stdout`, and `stderr` are used, and a file override is present on any of these streams, prior to opening the stream the override takes effect, and the I/O operation may not go to the terminal. An override to a file in the integrated file system is not supported. You can redirect I/O from `STDIN`, `STDOUT`, and `STDERR` to an integrated file system file by using the `freopen()` function:

```
fp = freopen ("\stdout_file", "w+", stdout);
```

If `stdout` or `stderr` are used in a noninteractive job, and if there are no file overrides for the stream, the run-time library overrides the streams to the printer file `QPRINT`.

If `stdin` is specified (or the default accepted) for an input file that is not part of an interactive job, the `QINLINE` file is used. `QINLINE` is handled by the database reader as an unnamed file, and cannot be read again. You can issue an override to avoid this problem.

Session Manager

The session manager is the same as the session manager for the data management file system. See "Session Manager" on page 56.

Understanding Stream Buffering

There are three buffering schemes defined for ANSI standard C streams: unbuffered, fully buffered and line buffered. See “Understanding Stream Buffering” on page 56 for information on the buffering schemes. The POSIX API functions, with the integrated file system enabled, supports any unbuffered streams except `stdin`, `stdout`, and `stderr`.

The integrated file system is most efficient when reading and writing in 32K blocks. The default C and C++ stream I/O when using VisualAge for C++ for AS/400 is fully buffered to 32K. Since session I/O to `stdout`, `stdin`, and `stderr` uses the Dynamic Screen Manager (DSM) support, `setvbuf()` and `setbuf()` have no effect on these predefined session files.

Note: The `setbuf()` and `setvbuf()` functions allow you to control buffering and buffer size when using integrated file system files.

Enabling Integrated File System Stream Input and Output

The default C and C++ stream I/O used by the VisualAge for C++ for AS/400 compiler is integrated file system enabled stream I/O. If the `/ASi-` compiler option is specified with a program that uses C stream I/O, the data management stream I/O is used, and the `\n` escape character has a value of `0x15`. If the `/ASi-` compiler option is specified with a program that uses C++ stream I/O the results may be incorrect. The C++ stream I/O only uses the integrated file system I/O. For service programs shipped with the compiler that use C or C++ stream I/O, the `\n` escape character has a value of `0x25`.

The integrated file system includes a macro `__IFS_IO__`. When `__IFS_IO__` is defined, the prototypes associated with stream I/O in `<stdio.h>` are no longer defined. The header file `<ifs.h>` is included by the `<stdio.h>` header file, which declares all the structure and prototypes associated with the integrated file system enabled C or C++ stream I/O.

This program uses the `ofstream` class and `ifstream` classes to copy one character at a time from an input file to an output file. The "root" (/) file system is used.

```

// This program copies contents of file "fin.txt" onto "fout.txt"

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main()
{
    ofstream fout("fout.txt",ios::trunc | ios::out, filebuf::openprot);
    ifstream fin("fin.txt");

    if (fin && fout)
    {
        char c=0;
        while (fout && fin.get(c))
            fout.put(c);
        if (!fin.eof())
        {
            cerr << "unexpected end of file" << endl;
            exit (0);
        }
        fin.close();
        fout.close();
    }
    else
    {
        cerr << "Could not open FILE(s)" << endl;
        exit (0);
    }
}

```

Using Stream Input and Output with Other File Systems

The integrated file system enabled C stream I/O is not compatible with the C data management stream I/O. The default for VisualAge for C++ for AS/400 is that all stream I/O compiled in a module is integrated file system enabled. To use the C data management stream I/O keep the C data management stream I/O in a separate module and compile it by specifying the /ASi- compiler option.

The integrated file system enabled C stream I/O FILE and fpos_t structures are not compatible with the C data management stream I/O FILE and fpos_t structures.

The default AS/400 file-name format LIBRARY/FILE(MEMBER) is not supported by the integrated file system enabled C stream I/O. If you want to access a data-management file, you must specify a QSYS.LIB file system path name:

```
QSYS.LIB/LIBRARY_NAME.LIB/FILE_NAME.FILE/MEMBER_NAME.MBR
```

The C data management stream I/O only supports the LIBRARY/FILE(MEMBER) name format and does not support any integrated file system path names.

New-line characters or `\n` escape characters have different hexadecimal values depending on whether or not the module was compiled with the integrated file system enabled stream I/O. The default integrated file systems enabled modules use a value of `0x25` for new-line characters. Modules that use C data management stream I/O use `0x15` for new-line characters.

C record I/O can be used along with the integrated file system enabled C stream I/O. The C record I/O functions provide the best performance and functionality for accessing data in data management files.

The low level POSIX integrated file system APIs can be used along with the integrated file system enabled C stream I/O functions. The *System API Programming* contains further information. The POSIX APIs use a file descriptor rather than a file pointer to refer to a file when performing I/O operations on that file. The integrated file system enabled stream I/O uses a pointer to a `FILE` structure to refer to an open instance of a file. Since the integrated file system C stream I/O uses the POSIX APIs, the internal portion of the file structure contains the file descriptor associated with the open instance of that file. You can retrieve the descriptor value associated with a file pointer by using the `fileno()` function.

It is not recommended that integrated file system files be manipulated by both integrated file system C stream I/O and with the POSIX integrated file system APIs at the same time. Doing so may corrupt the file buffers in the C stream I/O `FILE` control block, which may cause both unpredictable results and data corruption.

Chapter 5. Using C Stream I/O or Record I/O Functions

There are two interfaces provided by the C library: stream I/O and record I/O. The data management file system offers an interface that is optimized for record I/O. The C stream I/O that uses data management files is simulated stream file processing with AS/400 records. This is in contrast to the I/O of stream data through the integrated file system interface which is true stream-file processing as discussed in Chapter 4, "Using C or C++ Stream Input and Output" on page 33.

Notes:

1. The only way to use the C data management stream I/O is to keep the C data management stream I/O in a separate module and compile it by specifying the /ASi- compiler option.
2. There are no C++ record I/O classes.

The section describes:

"Introducing the AS/400 Data Management File System"

"Introducing Record Files" on page 51

"Introducing Stream Files in the Data Management File System" on page 52

"Working with Text and Binary Streams in the Data Management File System" on page 54

"Using the Open Feedback Area" on page 65

"Using the Input and Output Feedback Area" on page 65

Introducing the AS/400 Data Management File System

A C stream file or record file is the same as an AS/400 data management system file. Data management system files are also called file objects. Data is stored and accessed on the system through file objects. Each AS/400 file object is differentiated and categorized by information stored within it. Each file has its own set of unique characteristics, which determines how the file can be used and which capabilities it provides. This information is called the file description.

The file description contains:

Details on how the data associated with the file is organized into records

Detailed field descriptions and information on how they are organized within these records

Whenever a file is processed, the operating system uses the file description. The object type for file objects is *FILE.

The AS/400 data management system files are:

Database files that store data on the AS/400 data management file system

Display files that define the format of the information to present on a display and define how the information is processed by AS/400 on its way to and from the display

Printer files that define the format of the information to print on a printer

Tape files that define tape units

Diskette files that define diskette units

Intersystem communications function (ICF) files that define the layout of the data sent and received between two programs on different systems, and link the configuration objects that are used to communicate with the remote system

Save files that save data in a format used for backup and recovery purposes

Distributed data management (DDM) files that access data on remote AS/400 systems

C Streams and File Types

Table 2 summarizes which file types are supported as streams.

Stream	Data-base	Diskette	Tape	Printer	Display	ICF	DDM	Save
TEXT	X	-	-	X	-	-	X	-
BINARY								
Character at a time	X	-	-	X	-	-	X	-
Record at a time	X	X	X	X	X	X	X	X

All file types are supported by the C record I/O library.

File-Naming Conventions

The `_Ropen()` and `fopen()` functions that refer to AS/400 data management system files require a *file-name*. This file name must be a null-terminated string.

The syntax of a C file name is:

```
library-name/      file-name
                  file-name(member-name)
```

You use the *library-name* to enter the name of the library that contains the file. If you do not specify a library, the system searches the job's library list for the file.

You use the *file-name* to enter the name of the file. This is a required parameter.

You use the *member-name* to enter the name of the file member. If you do not specify a member name, the first member (*FIRST) is used.

Note: If you specify *ALL for *member-name* when you open a file for input using the `fopen()` and `_Ropen()` functions, sequential multi-member processing occurs.

All characters specified for *library-name*, *file-name*, or *member-name* are folded to uppercase unless you surround the string by the back slash and quotation mark (\") control sequence. This allows you to specify the AS/400 quoted names:

```
"\"tstlib\"/tstfile(tstnbr)"
Library is:  "tstlib"
File is:    TSTFILE
Member is:  TSTMBR
```

If you surround the *file-name*, *library-name*, or *member-name* in double quotation marks and the name is a normal name, the double quotation marks are discarded by the compiler. A normal name is any file, library, or member name with these characters:

- Uppercase characters
- Numeric values
- \$ (hexadecimal value 0x5B)
- @ (hexadecimal value 0x7C)
- # (hexadecimal value 0x7B)
- _ (hexadecimal value 0x6D)
- .(hexadecimal value 0x4B)

These characters cannot appear anywhere in your file names, library names, or member names:

Character	Hexadecimal Representation
(0x4D
*	0x5C
)	0x5D
/	0x6E
?	0x6F
'	0x7D
"	0x7F
(blank)	0x40

Note: "() / " can be used in quoted file names.

Introducing Record Files

The C run-time library provides a set of extensions to the C standard library definition for I/O. This set of extensions, referred to as *record I/O*, allows your program to perform I/O operations one record at a time.

The C record I/O library works with all the file types supported on AS/400. A complete set of record I/O functions and the data management operations that they map to are defined in the *C Library Reference*.

Each file that is opened with the `_Ropen()` function has an associated structure of type `_RFILE`. This structure is defined in the `<recio.h>` header file. Unpredictable results may occur if you attempt to change this structure. To use stream files (`type=record`) with record I/O functions, you must cast the `FILE` pointer to an `_RFILE` pointer.

The record-format name for a device file defaults to blank unless you explicitly set it to a name with the `_Rformat()` function. You can reset the format name to blanks by passing a blank name to `_Rformat()`.

Each AS/400 data management system file type has its own open modes and keyword parameters. See “Introducing the AS/400 Data Management File System” on page 49 for an overview of each of the AS/400 system file types. You can review information on each file type and how to open a record file using the `_Ropen()` function from:

Chapter 7, “Using Externally Described Files” on page 91

Chapter 8, “Using AS/400 Database Files and DDM Files” on page 131

Chapter 9, “Using Device Files” on page 149

See also the *C Library Reference* for information on the `_Ropen()` function.

Introducing Stream Files in the Data Management File System

In the ANSI definition of the C language, a *stream file* is a sequence of data that is read and written one character at a time. All I/O operations in the ANSI C library are stream operations.

On the AS/400 data management file system, all files are made up of records. All I/O operations at the operating system level are carried out one record at a time, using data management operations. *Data Management* contains information about using data management operations.

The run time allows your program to process stream files as text stream files (character-at-a-time processing) or as binary stream files (character-at-a-time processing or record-at-a-time processing). Since the AS/400 data management file system carries out I/O operations one record at a time, the C run-time library simulates stream file processing with AS/400 records. Although the C run-time library logically handles I/O one character at a time, the actual I/O that is performed by the operating system is done a record at a time.

Note: Since the AS/400 data management file system carries out I/O operations one record at a time, using the AS/400 commands such as Open Query File (OPNQRYF) that logically work with records instead of streams, together with stream I/O operations on the same file may cause positioning problems in the file your program is processing. Similarly, if you mix the use of the AS/400 extensions for C record I/O with stream file functions on the same file, unpredictable results may occur.

Streams and Database Files

The C library presents a stream as a continuous string of characters. If the file is a data management file, the records in the file are mapped to a stream as shown in Figure 11.

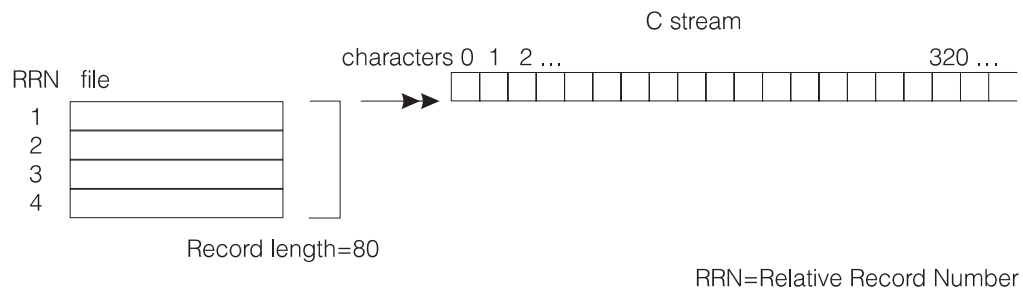


Figure 11. AS/400 Data Management Records Mapping to a C Stream File

Text Streams

A *text stream* is an ordered sequence of characters composed of lines. Each line consists of zero or more characters and ends with a new-line character. The C stream I/O may add, alter, or delete some special characters during input or output; there may not be a one-to-one correspondence between the characters written to a text stream, and characters read from the same text stream.

When a file is closed, an implicit new-line character is appended to the end of the file unless a new-line character is already specified.

Data read from a text stream compares as equal to the data that is written to the text stream if:

- The data consists of printable characters, horizontal tab, vertical tab, new-line characters, or form-feed control characters

- No new-line character is immediately preceded by a space (blank) character

- The last character in a stream is a new-line character

- The lines written to a file do not exceed the record length of the file

Binary Streams

A *binary stream* is a sequence of characters that has a one-to-one correspondence with the characters stored in the associated AS/400 data management system file. Similar to text streams, binary streams map to records in AS/400 data management system files. They can be processed one character or one record at a time. Character translation is not performed on binary streams. When data is written to a binary stream, it is the same when it is read back later.

There is one exception to the one-to-one correspondence. Since binary streams are stored in data management files, which are made up of records, the last record in a

binary stream opened for output is padded with null bytes (hexadecimal value 0x00) when the file is closed. A program that reads from such a file must account for this.

New-line characters have no special significance in a binary stream.

On the AS/400, the length of a binary stream file is a multiple of the record length.

Working with Text and Binary Streams in the Data Management File System

Each text stream file and each binary stream file is represented by a file control structure of type `FILE`. This structure is defined in the `<stdio.h>` header file. Unpredictable results may occur if you attempt to change the file control structure.

The format of the `fopen()` function is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The `mode` variable is a character string that consists of an open `mode`, which may be followed by `keyword` parameters. The `open mode` and `keyword` parameters must be separated by a comma or one or more blank characters.

If you specify a `mode` or `keyword` parameter that is not valid on the `fopen()` function, `errno` is set to `EBADMODE`, and `NULL` is returned.

The number of files that can be simultaneously opened by `fopen()` depends on the size of the system storage available.

The `fopen()` open modes and keywords also apply to the `freopen()` function.

If you specify the `type` parameter, the value must be `memory` for binary stream character-at-a-time processing.

Notes:

1. The `type` parameter identifies a file as a memory file. This is the default. The parameter is ignored. It is included for portability.
2. If the binary stream file contains deleted records, the deleted records are skipped by the binary stream I/O functions.

Open Modes for Dynamically Created Stream Files

If you specify `w`, `a`, `w+`, `a+`, `wb`, `ab`, `wb+`, `ab+`, `w+b`, or `a+b` as the mode when opening a file, the AS/400 data management file system automatically creates the file if it does not already exist. A physical database file is created if you are using binary mode. A source physical file is created if you are using text mode. If the file exists, but the member does not, the AS/400 adds the member to the file.

If you do not specify a library name when you open the file, the database file is dynamically created in library QTEMP. If you do not specify a member name, a member is created with the same name as the file.

The length specified on the *lrecl* parameter of `fopen` is used for the record length of the file created, with the following exceptions:

If you do not specify a record length when you open a text file, a source physical file with a record length of 266 is created.

If you do not specify a record length when you open a binary or record file, a physical file with a record length of 80 is created.

If you specify a record length of zero (*lrecl=0*) when you open a text file, a source physical file with a record length of 266 is created.

If you specify a record length of zero (*lrecl=0*) when you open a binary file, a physical file with a record length of 80 is created.

Note: If you edit a dynamically created stream file using Source Entry Utility (SEU), it must have a record length of 240 characters or less.

Dynamic file creation for text stream files is the same as specifying this command:

```
CTTHCMD /ASnname CRTSRCPF FILE(filename) RCDLEN(recln)
```

Dynamic file creation for binary stream files is the same as specifying this command:

```
CTTHCMD /ASnname CRTPF FILE(filename) RCDLEN(recln)
```

Understanding Session Input and Output

When a program starts, three text streams are defined:

Standard input `stdin`, for reading input from the terminal

Standard output `stdout`, for writing output to the terminal

Standard error `stderr`, for writing diagnostic output to the terminal

Streams `stdin`, `stdout`, and `stderr` are implicitly opened the first time they are used:

Stream `stdin` is opened with `fopen("stdin", "r")`

Stream `stdout` is opened with `fopen("stdout", "w")`

Stream `stderr` is opened with `fopen("stderr", "w")`

These streams are not real AS/400 data management system files, but are simulated as files by the C library routines. By default, they are directed to the terminal session.

The `stdin`, `stdout`, and `stderr` streams can be associated with other devices by using the AS/400 override commands on the files `STDIN`, `STDOUT`, and `STDERR` respectively. If `stdin`, `stdout`, and `stderr` are used, and a file override is present on one of these streams, prior to opening it, the override takes effect and, the I/O operation may not go to the terminal and the I/O operation is performed on the overriding file.

If `stdout` or `stderr` are used in a non-interactive job, and if there are no file overrides for the stream, the compiler overrides the streams to the printer file `QPRINT`.

If `stdin` is specified (or the default accepted) for an input file that is not part of an interactive job, the `QINLINE` file is used. You cannot reread a file with `QINLINE` specified, since the database reader handles it as an unnamed file, and, therefore, it cannot be read twice. You can get around this by issuing an override.

If `stdin` is specified in batch and has no overrides associated with it, `QINLINE` is used. If `stdin` has overrides associated with it, the override is used instead of `QINLINE`. If you are reading characters from `stdin`, `F4` triggers the run time to terminate any pending input and set the `EOF` indicator on, and `F3` is the same as calling `exit()` from your program. This is for an interactive session.

Note: You can use the `freopen()` function to reopen text streams. The `stdout` and `stderr` streams can be reopened for printer and database files. Unpredictable results may occur when performing I/O operations on a database file. The `stdout` and `stderr` streams must not be used on a database file. The `stdin` stream can only be overridden with database files. Use the C record I/O functions described in Chapter 8, "Using AS/400 Database Files and DDM Files" on page 131 to handle database files.

Session Manager

C stream I/O functions that output information to the display are implemented through the Dynamic Screen Manager (DSM) session manager APIs. You can obtain the session handle for the C session and then use the DSM APIs to manipulate that session. The session handle is supplied through the `_C_Get_Ssn_Handle` function in the header file `<stdio.h>`.

You can write a C program to clear the C session using the DSM `QsnClrScl` API:

```
#include <stdio.h>
#include "qsnapi.h"

int main (int argc, char *argv[])
{
    QsnClrScl(_C_Get_Ssn_Handle(), '0', NULL);
}
```

You can use the DSM APIs to perform any operation that is valid with a session handle, which includes the window services APIs and many of the low-level services also. You can display the session using a combination of the `QsnStrWin`, `QsnDspSsnBot`, and `QsnReadSsnDta` APIs, or you can write a program that contains a `getc()` function. You can use the `QsnRtvWinD` and `QsnChgWin` APIs to change the C session from the default full-screen window to a smaller window.

Understanding Stream Buffering

Three buffering schemes are defined for ANSI standard C streams. They are:

Unbuffered

Characters are intended to appear from the source or at the destination, immediately. The data management stream I/O does not support unbuffered streams.

Fully buffered

Characters are transmitted to and from a file a block at time, after the buffer is full. When using the default data management file system stream I/O, the compiler treats a block as the size of the record of the system file.

Line buffered

Characters are transmitted to and from a file, as a block, when a new-line control character (`\n`) is encountered.

Because a block and a line are equal to the record length of the opened file, the compiler supports fully-buffered and line-buffered streams the same way.

Note: The `setbuf()` and `setvbuf()` functions do not allow you to control buffering and buffer size when using the data management file system.

Opening Text Stream Files

To open an AS/400 data management system file as a text stream file, use the `fopen()` function with a mode of `r`, `w`, `a`, `r+`, `w+`, or `a+` and the optional keyword parameters `lrecl`, `ccsid`, and `recfm` (`F`, `FA`, and `FB` only).

If the text stream file contains deleted records, the deleted records are skipped by the text stream I/O functions.

Reading and Writing to Text Stream Files

If you are working with text stream files it is important to remember that a line corresponds to a record in an AS/400 database file. Operations on text streams work with, at most, one record at a time.

Figure 12 shows that during a write operation to a text stream file, a new-line character in the buffer causes the remainder of the record in the text stream file to be padded with blank characters (hexadecimal value `0x40`), and the record to be written to the text stream file. The new-line character itself is discarded.

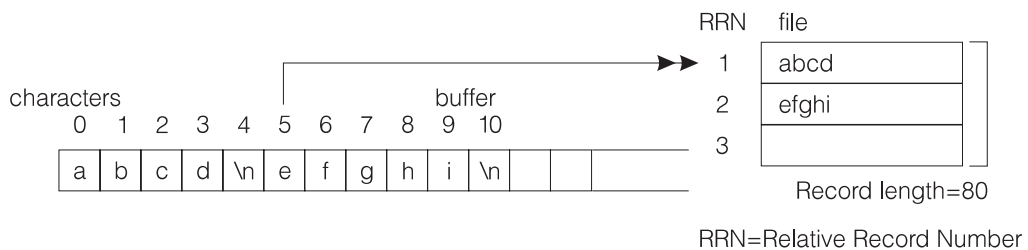


Figure 12. Writing to a Text Stream File

If the number of characters being written in the buffer exceeds the record length of the file, the data written to the file is truncated, and `errno` is set to `ETRUNC`.

Figure 13 shows that during a read operation from a text stream file, all the trailing blank characters (hexadecimal value `0x40`) in the record that is read from the file into a buffer are ignored, and a new-line character is inserted after the last non-blank.

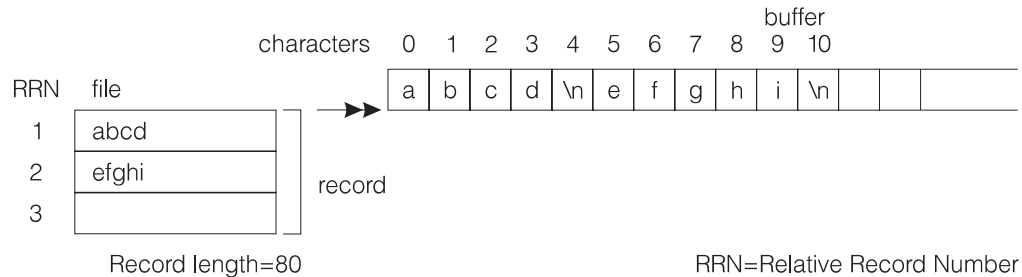


Figure 13. Reading from a Text Stream File

Overwriting Existing Data with in a Stream File

During an update operation to a text stream file, if the number of characters being written to the file exceeds the record length of the file, trailing characters in the record are truncated, and `errno` is set to `ETRUNC`.

If the data being written to the text stream file is shorter than the record length being updated, and the last character of the data being written is a new-line character, the record is updated and the remainder of the record is filled with blank characters. If the last character of the data being written is not a new-line character, the record is updated and the remainder of the record remains unchanged.

Opening, Reading, and Writing to a Text Stream File

This program shows how to open, read, and write to a text stream file. Library `MYLIB` must exist. The file `TEST1` must exist. The file `TEST2` is created for you if it does not exist. The mode `"r"` indicates that the member `MBR` must exist in the file `TEST1`. The mode `"w"` indicates that the member `MBR` in file `TEST2` is created if it does not already exist. If the member `TEST2(MBR)` does exist, its contents are destroyed unless it is a logical file.

```

#include <stdio.h>
#include <stdlib.h>

#define INFILE "MYLIB/TEST1(MBR)"
#define OUTFILE "MYLIB/TEST2(MBR)"
#define BUFSIZE 1024

int main(void)
{
    FILE *infp, *outfp;
    char buf [BUFSIZE]
    int line_num;

    /* Open an existing text file for reading.*/

    if (( infp = fopen ( INFILE, "r" ) ) == NULL )
    {
        printf ( "Cannot open %s\n", INFILE);
        exit (1);
    }

    /* Open an existing text stream file for writing.*/

    if (( outfp = fopen ( OUTFILE, "w" ) ) == NULL )
    {
        printf ( "Cannot open %s\n", OUTFILE);
        exit (2);
    }

    line_num = 0;

    /* While there are more lines to read.*/

    while (fgets (buf, sizeof(buf), infp) !=NULL)

        {
            fprintf(outfp, "%4d) %s", line_num, buf);
        }

    /* Close the text files.*/

    fclose ( outfp );
    fclose ( infp );
    exit(0);
}

```

Opening Binary Stream Files (character-at-a-time processing)

To open an AS/400 data management system file as a binary stream file for character-at-a-time processing, use the `fopen()` function with a mode of `rb`, `wb`, `ab`, `r+b`, or `rb+`,

w+b, or wb+, or a+b, or ab+ and the optional keyword parameters *blksize*, *lrecl*, *recfm*, *type*, *ccsid*, and *arrseq*.

Figure 14 shows that if you write data to a binary stream that is processed one character at a time, and the size of the data is greater than the current record length, the excess data is written to the current record up to its record size, and the remaining data is written to the next record in the file.

Reading From and Writing to Binary Stream Files

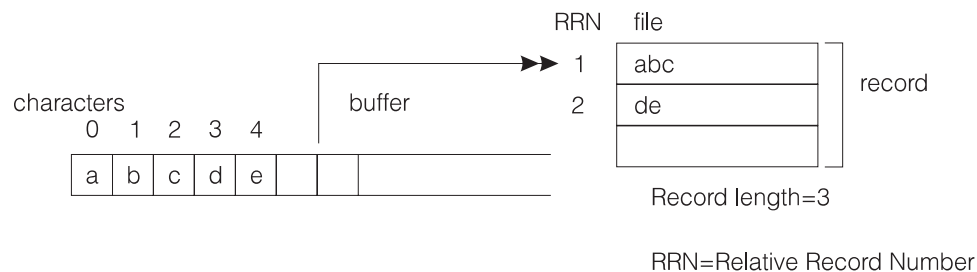


Figure 14. Writing to a Binary Stream File One Character at a Time

Figure 15 shows that during a read operation from a binary stream that is processed one character at a time, if the length of the data being read is greater than the record length of the file, data is read from the next record in the file.

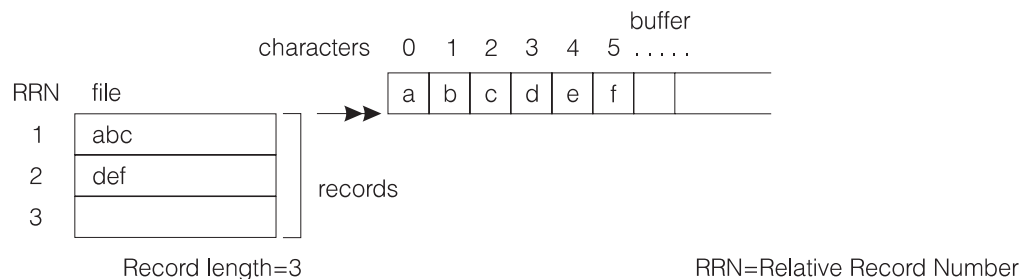


Figure 15. Reading from a Binary Stream File One Character at a Time

Opening, Reading From, Writing to a Binary Stream File

This program shows how to open, read from, and write to a binary stream file one character at a time. The file TEST1 is created for you if it does not exist. The member MBR in TEST1 is created for you. The mode "wb+" indicates that a binary file is open for reading and writing, and that, if the member MBR exists, its contents are cleared. The file TEST2 must exist. The mode "rb" indicates that a binary file is opened for reading.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp1;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};

    /* Open an existing binary file for writing.*/

    if ( ( fp1 = fopen ( "MYLIB/TEST1(MBR)", "wb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }

    printf ("Opened the file successfully\n");

    /* Perform some I/O operations.*/

    fprintf (fp1, "Hello, world");

    /* Write 5 characters from the buffer to the file.*/

    fwrite ( buf, 1, sizeof(buf), fp1 );
    fclose ( fp1 );

    FILE *fp2;
    char buf2[6];

    /* Open an existing binary file for reading.      */
    if ( ( fp2 = fopen ( "MYLIB/TEST1(MBR)", "rb" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer.    */

    fread ( buf2, 1, sizeof(buf2), fp2 );
    printf ( "%.6s\n", buf2 );

    fclose ( fp2 );
    return (0);
}

```

Updating a Binary Stream File

Figure 16 on page 62 shows that if the amount of data being updated exceeds the current record length, the next record is updated with the excess data. If the current record is the last record in the file, a new record is created.

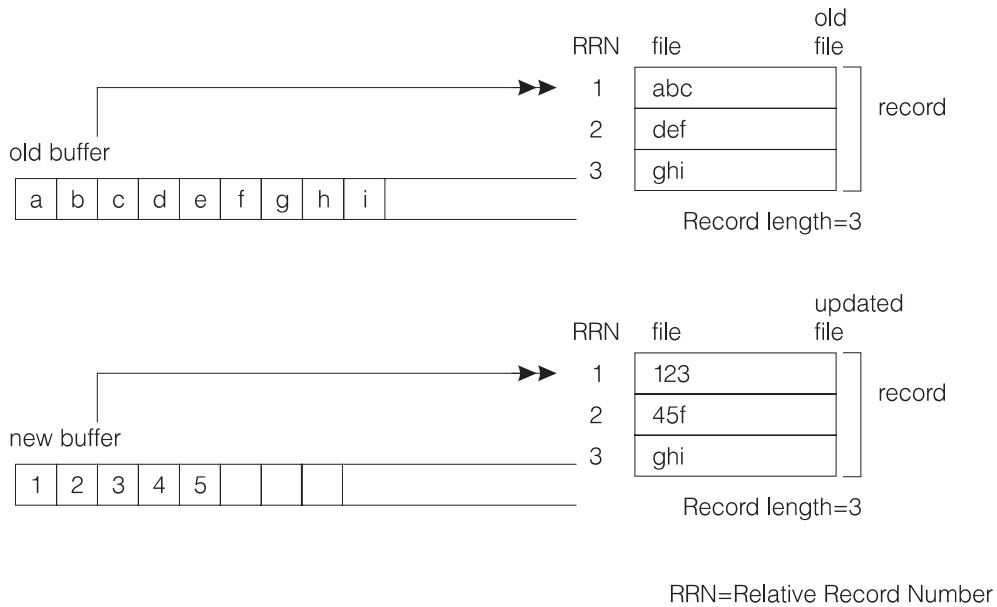


Figure 16. Updating a Binary Stream File With Data Longer Than the Record Length

Figure 17 shows that if the amount of data being updated is shorter than the current record length, the record is partially updated and the remainder is unchanged.

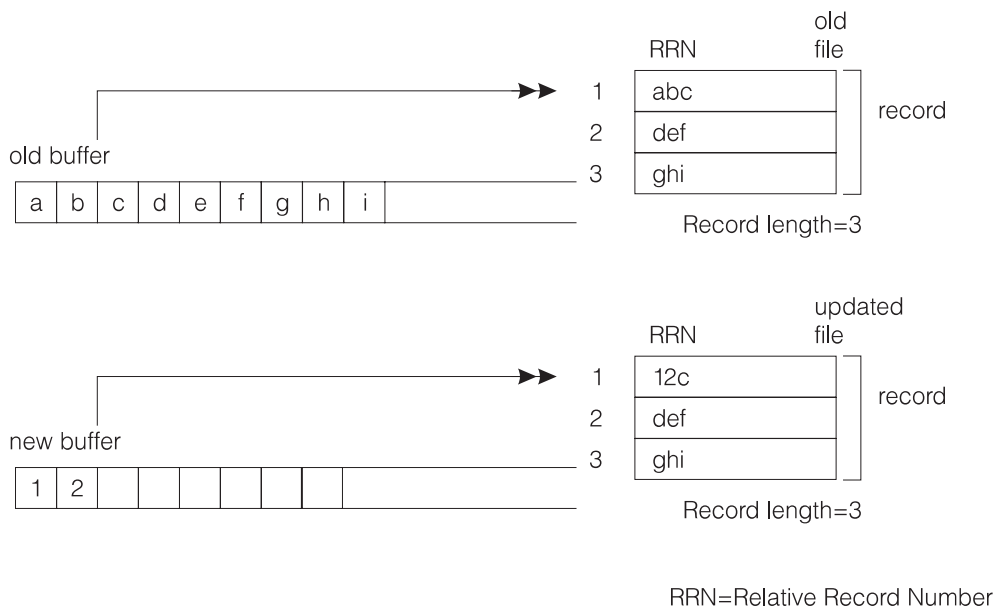


Figure 17. Updating a Binary Stream File With Data Shorter Than the Record Length

Updating a Binary Stream File

This program shows updating a binary stream file `TEST1` with data that is longer than the record length, and updating a binary stream file `TEST2` with data that is shorter than the record length. `TEST1` is created for you if it does not exist. The file `TEST2` must exist. Assume that `TEST1` has a record length of less than six and `TEST2` has a record length greater than three.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp1;
    char buf[6] = {"12345"};

    /* Open an existing binary file for updating.*/

    if ((fp1 = fopen ("QTEMP/TEST1(MBR)", "rb+")) == NULL)
    {
        printf ("Cannot open file\n");
        exit (1);
    }

    /* Write 6 characters from the buffer to the file.*/

    fwrite (buf, 1, sizeof(buf), fp1);

    fclose (fp1);

    FILE *fp2;
    char buf2[3] = {"12"};

    /* Open an existing binary file for updating.*/

    if ((fp2 = fopen ("QTEMP/TEST2(MBR)", "rb+")) == NULL)
    {
        printf ("Cannot open file\n");
        exit (1);
    }

    /* Write 3 characters from the buffer to the file.*/

    fwrite (buf2, 1, sizeof(buf2), fp2);
    fclose (fp2);
    return (0);
}
```

Opening Binary Stream Files (record-at-a-time processing)

To open an AS/400 data management system file as a binary stream file for record-at-a-time processing, use the binary stream functions described in Chapter 8, "Using

AS/400 Database Files and DDM Files” on page 131 and Chapter 9, “Using Device Files” on page 149.

Only the `fwrite()` function is valid for writing to binary stream files opened for record-at-a-time processing. All other output and positioning functions fail, and `errno` is set to `ERECIO`.

Only the `fread()` function is valid for reading binary stream files opened for record-at-a-time processing. All other input and positioning functions fail, and `errno` is set to `ERECIO`. See Chapter 8, “Using AS/400 Database Files and DDM Files” on page 131 and Chapter 9, “Using Device Files” on page 149 for examples of working with binary stream files one record at a time.

Reading From and Writing to Binary Stream Files

Figure 18 shows that if you write data to a binary stream processed one record at a time, and the product of *size* and *count* (parameters of `fwrite()`) is greater than the record length, only the data that fits in the current record is written, and `errno` is set to `ETRUNC`.

If the product of *size* and *count* is less than the actual record length, the current record is padded with blank characters, and `errno` is set to `EPAD`.

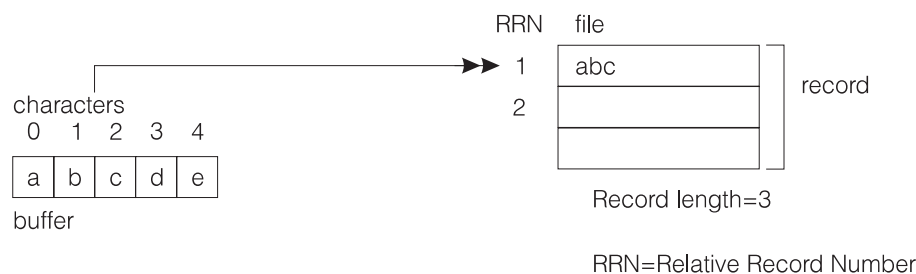


Figure 18. Writing to a Binary Stream File One Record at a Time

If you read data from a binary stream processed one record at a time, and the product of *size* and *count* (parameters of `fread()`) is greater than the record length, only the data in the current record is read into the buffer. The `fread()` function returns a value indicating that there is less data in the buffer than was specified.

If the product of *size* and *count* is less than the actual record length, `errno` is set to `ETRUNC` to indicate that there is data in the record that was not copied into the buffer.

Figure 19 on page 65 shows how only the current record is read into the buffer when the product of *size* and *count* is greater than the record length.

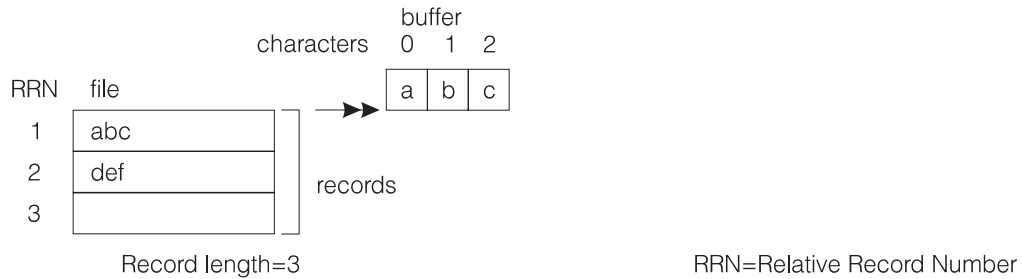


Figure 19. Reading From a Binary Stream File One Record at a Time

Using the Open Feedback Area

The open feedback area is the part of the open data path that contains information about the open file associated with that open data path. You can assign a pointer to a copy of this information by using the `_Ropnfbk()` function. See the *Data Management* for information on the fields contained in the open feedback area. The structure that maps to the open feedback area can be found in the `<xxfdbk.h>` header file.

Using the Input and Output Feedback Area

The I/O feedback area is the part of the open data path that contains information about a file that is updated after each successful non-blocked I/O operation. If record blocking is taking place, the I/O feedback is updated after each block of records is transferred between your program and the data management system.

The I/O feedback consists of two parts. The first part is common to all file types and the second part is specific to the type of the file.

To assign a pointer to the common part of the I/O feedback area, use the `_Riofbk()` function. To assign a pointer to the file-type-specific part of the I/O feedback area, add the offset contained in the `file_dep_fb_offset` field of the common part to a pointer to the common part.

Note: The offset is in bytes, so you need to cast the pointer (`char *`) to the common part when performing the pointer arithmetic.

See the *Data Management* for information on the I/O feedback areas. The structures that map to the I/O feedback areas are contained in the `<xxfdbk.h>` header file.

Obtaining Additional Information About Program Devices

This program uses the `_Riofbk()` function to obtain additional information about the program devices associated with your program.

The display file `T2123DDF` is the same DDS source as that described in “Change the Default Program Device” on page 157.

To override the file `STDOUT` with the printer file `QPRINT`, use this AS/400 command:

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

This source uses two typedefs: `_XXIOFB_T` for common I/O feedback, and `_XXIOFB_DSP_ICF_T` for display-file-specific I/O feedback. A pointer to the I/O feedback is returned by `_Riofbk (fp)`.

```
/* This program illustrates how to use the _Riofbk function to */  
/* access the I/O feedback area.*/
```

```
#include <stdio.h>  
#include <recio.h>  
#include <signal.h>  
#include <xxfdbk.h>  
#include <stdlib.h>  
#include <string.h>  
  
static void handler (int);  
  
_RFILE *fp;  
  
/* Signal handler for _Racquire exceptions*/  
  
static void handler (int sig)  
{  
    _XXIOFB_T          *io_feedbk;  
    _XXIOFB_DSP_ICF_T *dsp_io_feedbk;  
  
    signal ( SIGALL, handler );  
  
    io_feedbk = _Riofbk ( fp );  
    dsp_io_feedbk = (_XXIOFB_DSP_ICF_T *) ( (char *) (io_feedbk) +  
        io_feedbk->file_dep_fb_offset );  
    printf ( "Acquire failed\n" );  
    printf ( "Major code: %2.2s\tMinor code: %2.2s\n",  
        dsp_io_feedbk->major_ret_code,  
        dsp_io_feedbk->minor_ret_code);  
    exit ( 1 );  
}  
  
int main(void)  
{  
    char    buf[20];  
    _RIOFB_T *rfb;  
  
    if (( fp = _Ropen ( "MYLIB/T2123DDF", "ar+" ) ) == NULL )  
    {  
        printf ( "Could not open the display file\n" );  
        exit ( 2 );  
    }  
}
```

```

signal ( SIGALL, handler );

_Racquire ( fp, "DEVICE1" ); /*Acquire the device. DEVICE1 is*/
                             /* now the default program device.*/
                             /* NOTE : If the device is not */
                             /* acquired, exceptions are issued. */

_Rformat ( fp, "EXAMPLE" ); /* Select the record format. */

_Rwrite ( fp, "Hello", 5 ); /*Write to default program device.*/

                             /* Read from default program device. */
rfb = _Rreadn ( fp, buf, 21, __DFT );

printf ( "user entered: %20.20s\n", buf );

_Rclose ( fp );
}

```

The output is the same as in the program output for “Change the Default Program Device” on page 157.

The `signal()` function is called before the acquire operation to establish a signal handler. If an exception occurs during the acquire operation, the signal handler is called to write the major or minor return code to `QPRINT`. If `DEVICE1` is in use when the `_Racquire()` operation is performed, the following message is sent to `QPRINT`:

```

Acquire failed
Major code: 82 Minor code: A8

```

Chapter 6. Improving Program Performance

Often the cause of a program's poor performance is very specific. Inefficient code can affect performance when it is looped through many times during program execution. When you examine a program to improve performance, look at those aspects which have a significant impact every time a program is run.

Often run-time performance can be improved through minor changes to your source programs. The amount of improvement each tip provides depends on how your program is organized, and on the functions and language constructs your program uses. Some tips may provide substantial performance improvement to your program, while others may offer almost no improvement. Some tips may contradict each other because they may trade off one resource for another. One tip is to reduce the size of the call stack by using static and global variables. Another tip is to improve execution startup performance by reducing the use of static and global variables.

Use performance analysis tools to find out where your performance problems are, and then try and apply different appropriate tips to try and achieve the best performance for your program.

This section describes:

- “Data Types”
- “Classes” on page 70
- “Performance Measurement” on page 70
- “Exception Handling” on page 71
- “Function Call Performance” on page 72
- “Input and Output Considerations” on page 73
- “Open Pointers” on page 77
- “Shallow Copy and Deep Copy” on page 80
- “Space Considerations” on page 80
- “Compile-Time Performance Tips” on page 85
- “Run-Time Limits” on page 86

Note: Before trying different tips, first compile and benchmark your programs using full optimization.

Data Types

There are several ways to improve performance through data types. Replacing bit fields with other data types and minimizing the use of static and global variables are some of the ways.

Avoid Using the volatile Qualifier

Only use the `volatile` qualifier when necessary. `volatile` specifies that a variable can be changed at any time, possibly by an external program, and therefore it is not a candidate for optimization.

Replace Bit Fields with Other Data Types

Avoid using bit-fields, since it takes more time to access bit-fields than other data types such as `short` and `int`. Whenever possible, replace bit-fields with other data types. If a bit-field takes 16 bits and aligns on 2-byte boundary, you can replace it with the `short` data type.

Note: You can still obtain a run-time improvement if the bit-field is smaller than the integral type. The extra time required for bit-field manipulation code offsets the performance gain due to space saved in data.

Minimize the Use of Static and Global Variables

Minimize the use of static and global variables, if possible. These are initialized whether or not you explicitly initialize them. By not using static and global variables, the performance improvement is obtained at activation group startup.

Use the Register Storage Class

Use the register storage class for a variable that is frequently used. Do not overuse the register storage class, so that the optimizer can place the most frequently used variables into the available hardware registers.

If you use the register storage class, you cannot rely on the value displayed from within the debugger, since you may be referencing an older value that is still in storage.

Classes

When you use the IBM Open Class Library or IBM Access Class Library for OS/400 to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can result in improved performance with minimal code changes. See the *IBM Open Class Library User's Guide* for information on using classes.

When you define structures or data members within a class, define the largest data types first to align them on the largest natural boundary. Define pointers first to reduce the padding necessary to align them on quadword (16-byte) boundaries. Follow them, in order, with the double-word, and half-word items to avoid padding or improve load/store time.

Performance Measurement

You can use a performance-measurement compiler option `/Asp` to include performance hooks in your generated code.

Use a Compiler Option to Enable Performance Measurement

The performance-measurement compiler option `/ASP` allows you to specify whether or not the compiler should generate code (sometimes called "performance hooks") into your compiled program or module. The performance hooks enable the Performance Explorer to analyze your programs. The default for this option specifies that program entry procedure level performance-measurement code is generated for your compiled module or program.

Compiling performance collection code into the module or program allows performance data to be gathered and analyzed. The insertion of the additional collection code results in slightly larger module/program objects and may affect performance to a small degree.

Types of performance data collected include:

Pre- and post-call information

This information is gathered immediately before and after calling any given functions. It provides a record of where a call was made, and information on the performance of the operation called.

Procedure entry and exit information

This information is gathered immediately upon entry into a procedure and exit from that procedure. A snapshot is taken of the current performance statistics when entering a procedure, and a calculation is made of the differences in those statistics when exiting that procedure.

When performance collection code is generated into a leaf procedure, the procedure is changed so that it is no longer a leaf procedure. (A leaf procedure is one that does not call any other procedures.)

The extra expense of capturing data on a leaf procedure is mainly due to the fact that the leaf procedures being hooked lose their 'leaf-ness' in the process. This happens due to the fact that hooks are basically calls to collection routines and leaf procedures by definition do not make calls.

See the *C++ User's Guide* for information on these options.

Exception Handling

Reducing exceptions, turning off C2M messages during record I/O and using direct monitor handlers are some of ways to improve exception handling performance.

Reduce Exceptions

Exceptions are expensive to process. Try to minimize the number of exceptions in your programs.

If you use record I/O, the `rtncode=y` option can be used on `_Ropen()` to help reduce exceptions. Files that are opened with this option do not have exceptions generated for

the "Record not found" (CPF5006) and the "End-of-File" (CPF5001) conditions. When these conditions occur, the `num_bytes` field of the `_RIOFB_T` structure is updated and `errno` is set, but no exceptions are generated. For the "Record not found" condition, this field is set to zero. For the "End-of-File" condition, it is set to EOF.

If your program generates either of these conditions many times, then a significant performance improvement may be seen using this option.

Turn Off C2M Messages during Record Input and Output

Turn off C2M messages during record I/O. This is controlled by the variable `_C2M_MSG` declared in `<recio.h>`. Set the variable `_C2M_MSG` to zero to turn off C2M messages during record I/O. If `_C2M_MSG` is set, record I/O sends C2M messages to your program when it detects these errors: C2M3003, C2M3004, C2M3005, C2M3009, C2M3014, C2M3015, C2M3040, C2M3041, C2M3042 and C2M3044. Turning it off prevents record I/O sending such messages. Removing data truncation messages with signal handlers or message handlers is no longer necessary when the C2M messages are turned off during record I/O.

Use a Direct Monitor Handler

Use a direct monitor handler (this is done using the `#pragma exception_handler` in C++) instead of a signal handler. When an exception occurs, the compiler first attempts to use any direct monitor handler if there is one. Otherwise, C++ maps the exception to a signal, and calls the corresponding signal handler. By using a direct monitor handler, the signal mapping and search for signal handler are saved. The direct monitor handler should mark the exception as handled, if the exception is one you are expecting; otherwise the exception is percolated again. See the *C++ Language Reference* for information on this pragma.

Avoid Percolating Exceptions

Try to handle an exception in the place it occurs. There is some processing overhead incurred with exception percolation. See Chapter 14, "Handling Exceptions" on page 255 for information on handling exceptions.

Function Call Performance

You can reduce the number of function calls by using inline functions or macro expressions.

Reduce the Number of Function Calls and Function Arguments

You can improve performance by changing function calls to inline functions or macro expressions, provided such a change does not increase the size of the program object too much. A program object size increase may cause more page faults which slows the program down. Try to strike a balance between program size and inlining or macro expressions for an optimum level of performance.

When a function is only called in a few places but executed many times, changing the function to an inline function saves many function calls and results in performance improvement.

The `INLINE` compile-time option allows you to request that the compiler replace a function call with that function's code in place of the function call. If the compiler allows the inlining to take place, the function call is replaced by the machine code that represents the source code in the function definition.

Inlining is a method that allows you to improve the run time performance of a C++ program by eliminating the function call overhead. Inlining allows for an expanded view of the program for optimization. Exposing constants and flow constructs on a global scale allows the optimizer to make better choices during optimization.

The overhead of a bound function call is small compared to the overhead of an external dynamic call. This becomes significant when functions are called many times. Try to avoid external dynamic calls, if possible.

When calling a program dynamically from a C++ program using `extern "OS"` linkage, prototype the program to return `void` rather than `int`. Extra processing is involved in accessing the return value of a program call. Passing the address of storage that can hold a return value in the call's argument list is better from a performance viewpoint.

Function call performance can be improved if the system has all of the arguments passed in registers. Since there are only a limited number of registers, in order to increase the chance of having all arguments passed in registers, combine several arguments into a class and pass the address of the class to the function. Since an address is being passed, pass-by-reference semantics are used, which may not have been the case when the arguments were being passed as individual variables.

You can use static class members, a better programming practice that gives better performance.

However, an alternative to passing an argument to a function is to have the variable defined as being global and to have the function use the global variable. Also, global variables can inhibit optimization. Using more global variables increases the amount of work that has to be done at activation group start-up to allocate and initialize the global variables.

Input and Output Considerations

To improve I/O performance, you can, for example, use record I/O functions where appropriate, block records, reduce file open operations and file close operations and use stream I/O functions.

Use Record Input and Output Functions

Using record I/O functions instead of C stream I/O functions where it is appropriate can greatly improve I/O performance but can greatly reduce the portability of the source code. Instead of accessing one byte at a time, record I/O functions access one record at a time.

If you use C record I/O in your program, you must use C record I/O functions, for example, functions that begin with `_R`, and you must use the `_RFILE` data type as shown in the following code:

```
#include <iostream.h>
#include <recio.h>

#define MAX_LEN 80
int i;

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *iofb;
    char buf[MAX_LEN + 1];

    fp = _Ropen("MY_LIB/MY_FILE", "rr");

    for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )
    {
        _Rreadn(fp, buf, MAX_LEN, __DFT);
        buf[iofb->num_bytes] = '\0';
        cout <<buf <<endl;
        iofb =_Rreadn(fp, buf, MAX_LEN, __DFT);
    }

    _Rclose(fp);
}
```

Use Input and Output Feedback Information

`_RIOFB_T` is a structure returned by most record I/O functions, that contains feedback information on the last I/O operation. By default, the C record I/O functions update any of the fields (except for the `num_bytes` field) `_RIOFB_T` after a record I/O operation is performed. If your program does not use all these values, you can improve your program's performance by opening a file:

```
fp = _Ropen("MY_LIB/MY_FILE", "rr, riofb = N");
```

By specifying `riofb = N`, only the `num_bytes` field, the number of bytes read or written in the `_RIOFB_T` structure is updated. If you specify `riofb = Y`, all the fields in the `_RIOFB_T` structure are updated.

Block Records

You can improve record I/O performance by blocking records. When blocking is specified, the first read causes a whole block of records to be placed into a buffer. Subsequent read operations return a record from the buffer until the buffer is empty. At that time, the next block is fetched. Similar rules apply when blocking records for write operations.

If you wish to block records when using record I/O, specify `blkrcd = Y` on the call to the `_Ropen()` function.

Manipulate the System Buffer

You can improve I/O performance of your C++ programs by performing read and write operations directly to and from the system buffer, without the need for a program-defined buffer. This code shows how to directly manipulate the system buffer when reading a source physical file.

```
fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y, riofb = N");

for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )
{
    _Rreadn(fp, NULL, 92, __DFT);
    printf("%75.75s\n", ((char *) (*(fp->in_buf)) + 12));
}

_Rclose(fp);
```

The example code above prints up to 75 characters of each record contained in the file. The second parameter for the `_Rreadn()` function, `NULL`, tells the record I/O function to use the system buffer to access the data. The copy of the data between the system buffer and your buffer is avoided. An `_RFILE` structure contains the `in_buf` and `out_buf` fields, which point to the system input buffer and system output buffer, respectively. The example above prints each record by accessing the system's input buffer.

It may be tempting to save a local copy of the buffer pointer and use it to access the buffer to avoid the double indirection. This is not a correct operation because there are situations where the position of the buffer changes. Always use the doubly indirected pointers in the `_RFILE` structure.

Directly manipulating the system buffer provides a performance improvement when you process very long records. It provides a significant performance improvement when you use Intersystem Communications Function (ICF) files. You only need to access the last several bytes in an ICF file and not all the other data in the record. By using the system buffer directly, the data that you do not use for ICF files need not be copied.

Reduce the Number of File Opens and Closes

Open and close are very expensive operations. You can improve performance by opening and closing files only as often as necessary. You can use a class to encapsulate I/O operations such as opening the files once, and not closing the file until the end of the program.

Process Tape Files

You can improve the performance of programs that use tape files by using fixed-length record tape files instead of variable-length tape files.

Use Stream Input and Output Functions

The default for C or C++ stream I/O is integrated file system enabled. Also, the integrated file system C or C++ stream I/O is more efficient than C data management stream I/O. If you must use C data management stream I/O, this section describes what to consider.

Use the macro version of `getc()` instead of `fgetc()` to read characters from a file. See “Reduce the Number of Function Calls and Function Arguments” on page 72. The macro version of `getc` reads all the characters in the buffer until the buffer is empty. At this point, `getc()` calls `fgetc()` to get the next record.

Use `putc()` instead of `fputc()`. The macro version of `putc()` writes all the characters in the buffer until the buffer is full. At this point, `putc()` calls `fputc()` to write the record into the file.

Since the implementation of most data management stream I/O functions are very call intensive, reducing their use in your program improves performance.

Use C++ Input and Output Stream Classes

Use overloaded shift `<<` `>>` operators on the standard streams, instead of their C equivalents.

Use Physical Files Instead of Source Physical Files

To improve performance, Use physical files instead of source physical files for your data. When a source physical file is used for stream I/O, the first 12 bytes of each record are not visible to your program. They are used to store the record number and update time. These 12 bytes are an extra load that the C stream I/O functions must manipulate. When performing output, these 12 bytes must be initialized to zero. When performing input, these 12 bytes must be fetched even though they are not passed to your program. Since the C stream I/O functions dynamically create a source physical file when opening a text file that does not exist for output, create the file as a physical file before you start your program.

Specify Library Names

Specify the name of the library in which the file resides. If you do not specify a library name when opening a file, the library list is searched for the file. This search can be time-consuming depending on the number of libraries and the objects that they contain.

Pointers

Using and comparing pointers can impact performance.

Open Pointers

Avoid using open pointers. Open pointers inhibit optimization. Note that pointers to `void (void*)` are open pointers in ILE C++.

Pointer Comparisons

Since pointers take up 16 bytes of space, pointer comparisons on AS/400 are less efficient than comparisons using other data types. You may want to replace pointer comparisons with comparisons using other data types, such as `int`.

This is a program that constructs a linked list, processes all the elements in the list, and finally frees the linked list. Each element in the link list holds one record from a file:

```
#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80

struct link
{
    struct link *next;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;
    int i;

    // Construct the linked list and read in records.

    fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )
    {
        _Rreadn (fp, NULL, MAX_LEN, __DFT);
        ptr->next = (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record,(void const *) *(fp->in_buf), MAX_LEN);
        ptr->next = NULL;
    }
    ptr = start->next;
    free(start);
    start = ptr;
    _Rclose(fp);
}
```

```

// Process all records.

for ( ptr = start; ptr != NULL; ptr = ptr->next )
{
// Code to process the element pointed to by ptr.

}

// Free space allocated for the linked list.

while ( start != NULL )
{
    ptr = start->next;
    free(start);
    start = ptr;
}
}

```

In the preceding program, pointer comparisons are used when processing elements and freeing the linked list. The program can be rewritten using a `short` type member to indicate the end of the link list. As a result, you change pointer comparisons to integer comparisons:

```

#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80
int i;

struct link
{
    struct link *next;
    short last;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;

    // Construct the linked list and read in records.

    fp = _Ropen(" MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )

```

```

{
  _Rreadn(fp, NULL, MAX_LEN, __DFT);
  (struct link *) malloc(sizeof(struct link));
  memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);
  ptr->last = 0;
}
ptr->last = 1;
ptr = start->next;
free(start);
start = ptr;
_Rclose(fp);

// Process all records.

if ( start != NULL )
{
  for ( ptr = start; !ptr->last; ptr = ptr->next )
  {
    // Code to process the element pointed to by
  }
  // code to process the element
  //(last element) pointed.
  // Free space allocated for the linked list.

  while ( !start->last )
  {
    ptr = start->next;
    free(start);
    start = ptr;
  }
  free(start);
}
}

```

Reduce Indirect Access

You can improve performance by reducing indirect access through pointers. Each level of indirection adds some overhead:

```

for ( i = 0; i < n; i++ )
{
    x->y->z[i] = i;
}

```

Performance in the above example improves if it is rewritten as:

```

temp = x->y;
for ( i = 0; i < n; i++ )
{
    temp->z[i] = i;
}

```

Shallow Copy and Deep Copy

Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.

Note: Care must be taken that objects which use shallow copy do not destroy objects pointed to more than once.

Space Considerations

You can improve the performance of a program by reducing the space it requires. Reducing the space requirement helps reduce page faults, segment faults, and effective address overflows. It helps reduce the number of allocations beyond the size of the fast heap. See “Use the Fast Heap for Dynamic Storage” for information on the C fast heap.

Use the Fast Heap for Dynamic Storage

The first request for dynamic storage within an activation group results in the creation of a default heap from which the storage allocation takes place. On the first call within an activation group to the `malloc` or `calloc` functions, C creates a fast heap of a specified size, up to a maximum size of 16 711 600 bytes. Storage allocation requests from this heap are reasonably fast. Storage allocations beyond the size of this heap are much slower.

The default size of the fast heap is 64K-84bytes. It is recommended that you change this size to a size large enough to handle the dynamic storage requirements of your program. This way, your dynamic storage requests are always satisfied using the fast heap, and this can have a noticeable performance improvement.

You can do this using the `_HEAP_SIZE` macro, defined in `<stdlib.h>` header file:

```
_HEAP_SIZE = 262144;    // Set size of fast heap to 256K
```

If you are going to use this macro, you must use it prior to calling the `malloc()`, `calloc()`, or `realloc()` functions. Since some of the C library functions allocate dynamic storage, use this macro before the first call to any C library function or bindable API. Changing the default size after the first call that allocates dynamic storage has no effect.

Note: Static initialization takes place before `main()` is called. If a static class is constructed which allocates pointers using `new` then you must ensure that `_HEAP_SIZE` is set before this using a dummy initialization and the **#pragma priority**, for example:

```
static heap_size = SetHeapSize(262144);
int SetHeapSize(int size) {return _HEAP_SIZE = size; }
```

Use of the fast heap is optional. If you use the fast heap support:

Storage allocated by `malloc()`, `calloc()`, or `realloc()` functions, cannot be freed or reallocated with the ILE bindable APIs

`CEEFRST` and `CEEZST`.

Storage initially allocated with the ILE bindable API `CEEZST` can be reallocated with the `realloc()` function.

Storage allocated by the ILE bindable API `CEEZST` can be freed with the `free()` function.

If you choose not to use the fast heap support, there are no restrictions in using the storage management APIs, and `malloc()`, `calloc()`, `realloc()`, and `free()` functions.

The fast heap support can be bypassed by setting the `_HEAP_SIZE` macro to the value of `_NO_DEFAULT_HEAP`, which is defined in `<stdlib.h>`.

`new` and `delete` operators make use of the fast heap. Because C++ objects are often allocated from the heap and have a limited scope, memory usage in C++ programs affects performance more than in C programs. To improve memory usage and performance:

- Tailor your own `new` and `delete` operators

- Allocate memory and manage that memory as required

- Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an object manager. Each time you create an instance of an object, you pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.

- Avoid copying large complex objects

Choose Appropriate Data Types

Choosing the appropriate data type can help improve your program's performance. If possible use `short` instead of `int`, and `float` instead of `double`. C++ uses 2 bytes for `short`, 4 bytes for `int`, and 8 bytes for `double`. Using the right data types can reduce your program's space requirement.

Note: When choosing data types, consider all the platforms that your code must support. You may not know all the data types and sizes at the beginning of your code design. Since the data types can hold the same size data on various platforms, you can use typedefs, enums, or classes depending on the use of the data type.

Reduce Dynamic Memory Allocation Calls

You can improve performance by reducing the number of times you dynamically allocate memory. Every time you call the `new` operator a certain amount of space is allocated from the fast heap. This space is always aligned at 16 bytes, which is suitable for storage of any object type. In addition, 32 extra bytes are taken from the dynamic heap for bookkeeping. This means that even if you only want one byte, 48 bytes are allo-

cated from the dynamic heap, 32 bytes for bookkeeping and 15 bytes for padding. When the current space allocation in the fast heap is used up, storage allocation is slower:

```
ptr1 = new char[12];  
ptr2 = new char[4];
```

In the code above, 96 bytes are taken from the heap (including 64 bytes for bookkeeping and 16 bytes for padding) and `new` is used twice. This code can be rewritten as:

```
ptr1 = new char[16];  
ptr2 = ptr1 + 12;
```

Only 48 bytes are taken from the heap and the `new` operator is only used once. Since you reduce the dynamic space allocation requirement, less storage is taken from the fast heap. You may gain other benefits such as a reduction in page faults. Since there are fewer calls to the `new` operator, function call overhead is reduced as well.

Note: If allocating by incrementing pointers you must guarantee the proper alignment when allocating pointers or aggregates which can contain pointers (16 byte alignment). There is a performance degradation for types such as `float` whose natural alignment is word or doubleword if not allocated on their natural boundary.

Reduce Space Used for Padding

Reducing space wasted on padding by rearranging variables is another way to reduce your program's space requirement. In the C++ language:

A `char` type variable takes one byte

A `short` type variable takes 2 bytes

An `int` type variable takes 4 bytes

A `float` type variable takes 4 bytes

A `double` type variable takes 8 bytes

A pointer takes 16 bytes

A `_DecimalT` template class object takes 1 to 16 bytes

Each data type is aligned on its natural boundary. `pointers` are aligned on 16 byte boundaries, `integers` are aligned on 4 byte boundaries and `packed decimals` are aligned on a 1 byte boundary. The alignment of a structure or a union is determined by the largest alignment among its members. This class aligns on a 4 byte boundary because of the `float` member.

Note: To support portable code, you can place all the members of the same type together (this may affect readability) starting with the largest (on most systems) going to the smallest. An `array` has the same alignment as its element type and must be positioned appropriately. This program that follows shows this rule implicitly.

```

class OrderT
{
    float value;    // Four bytes.
    char flag1;    // One byte plus one byte.
    short num;     // Two bytes.
    char flag2;    // One byte plus three bytes.
}
orderT;

```

The size of the above structure is 12 bytes (not 8 bytes). This is because there is 1 byte of padding after member `flag1` since member `num` aligns on a 2 byte boundary, and there are 3 bytes of padding after member `flag2` since the structure is aligned on a 4 byte boundary.

By rearranging variables, the wasted space created by padding can be minimized. Consider the structure as:

```

class ItemT
{
    char *name;           // 16 bytes.
    int number;          // 4 bytes plus 12 bytes.
    char *address;       // 16 bytes.
    double value;        // 8 bytes plus 8 bytes.
    char *next;          // 16 bytes.
    short rating;        // 2 bytes plus 14 bytes.
    char *previous;      // 16 bytes.
    _DecimalT<25,5> tot_order; // 13 bytes plus 3 bytes.
    int quantity;        // 4 bytes.
    _DecimalT<12,5> unit_price; // 7 bytes plus 5 bytes.
    char *title;         // 16 bytes.
    char flag;           // 1 byte plus 15 bytes.
}
itemT;

```

The structure takes 176 bytes, of which 57 bytes are used for padding. It can be rearranged as:

```

class ItemT
{
    char *name;           // 16 bytes
    char *address;       // 16 bytes
    char *next;          // 16 bytes
    char *previous;      // 16 bytes
    char *title;         // 16 bytes
    double value;        // 8 bytes
    int quantity;        // 4 bytes
    int number;          // 4 bytes
    short rating;        // 2 bytes
    char flag;           // 1 byte
    _DecimalT<25,5> tot_order; // 13 bytes
    _DecimalT<12,5> unit_price; // 7 bytes plus 9 bytes
}
itemT;

```

The class only takes 128 bytes, with 9 bytes for padding. The saving of space is even more substantial when you have arrays of the above structure type.

As a general rule, the space used for padding can be minimized if 16 byte variables are declared first, 8 byte variables are declared second, 4 byte variables are declared third, 2 byte variables are declared fourth, and 1 byte variables are declared fifth. `_DecimalT` template class objects should be declared last, after all other variables have been declared. The same rule can be applied to structure or class definitions.

Note: The `/LB+` compiler option along with `/LS+` shows the layout, including padding, of the structures in a module, in both `_Packed` and normal alignment.

Activation Groups

Using activation groups can impact performance.

Calling Functions in Other Activation Groups

Within the same job, calling a function that runs in a different activation group degrades the performance of the call significantly (the call takes approximately two times longer).

If a service program was created to run in a named activation group (using the `ACTGRP(name)` parameter of the AS/400 `CRTSRVPGM` command) then any calls to that function from a program or service program would be calling *across an activation group* and would therefore be slower. Sometimes it makes sense to run programs or service programs in other activations groups (for storage isolation, exception handling) but it should be noted that call-performance suffers in that arrangement.

Reducing Program Start-Up Time

When a new C++ program is first called, the system needs to perform some initialization to prepare the program to run. Part of this initialization requires creating an activation group for all of the program storage, resolving all service programs bound to the program, getting program arguments and so on. Several recommendations for improving start-up time can be drawn from these initialization steps:

- Reduce the use of global variables.

- Reduce the number of service programs bound to the program. The more service programs used by an ILE program, the more time is required to start up the program. It is often better to have fewer, larger service programs than many, smaller ones. The C run-time is made up of a small number of service programs.

Program Control

Virtual functions and operators can impact performance.

Avoid Virtual Functions

There is a performance impact if you use virtual functions because virtual functions are compiled to be indirect calls, which are slower than direct calls. You may be able to

minimize this performance impact depending on your program design by using a minimum number of parameters on the virtual functions.

Use Cheaper Operators

The C++ language has many features that other languages do not provide which can be used to improve your C++ program's performance:

Instead of dividing an integer by 2^n , shift the integer right by n bits

Instead of using the remainder operator for number 2^n (`x % 32;`), use the bitwise-and operator `& 0x0000001F;`

Compile-Time Performance Tips

There are several ways to improved compile-time performance. These include both front and back end compile-time activities.

AS/400 Back-End Compile-Time Performance Tips

Ensure that the poolsize in which the back-end of the compiler is running in is not less than 16MB (preferably around 32MB for large programs).

Adjust the activity level such that the job performing the back-end compile does not have to compete with other jobs for system resources (if there are too many active jobs at once within the subsystem). Use this command to change prestart characteristics of your Client Access batch job:

```
CHGCLS CLS(QGPL/QCASERV) RUNPTY(50) TIMESLICE(5000) DFTWAIT(120)
```

The *Work Management* contains information about adjusting system resources.

If compiler options such as `/O1`, `/O2`, `/Li`, are not absolutely necessary do not use them as they cause the compiler to perform more work.

Windows Front-End Compile-Time Performance Tips

Separate the source into smaller files. Each file can be compiled into a module object and bound into a program or service program object. This arrangement does not impact overall run-time performance of the resulting program since call performance is the same regardless of whether the function called is in the same module, a different module within the program object, or in a module within a service program bound to the program.

Note: If you split calls across modules, you may be reducing the compiler's ability to inline, since inlining only works on a per module basis. Be aware that splitting a program into smaller modules may reduce optimization.

MI header files and AS/400 system header files are stored in the library QSYSINC. Copying the MI and AS/400 system header files you need to your workstation can save compilation time. Remember to copy these files again from QSYSINC whenever PTFs are applied that affect QSYSINC.

Only include the **#include** files that are necessary for that compilation unit. If there is a possibility of the same include file being included multiple times, use the **#define** and **#ifdef** directives to ensure that it is only included once.

Choosing Compiler Options

Table 3 describes different compiler options to make your program run faster, and to make your compile program smaller. Note that sometimes you have to decide which is more important to you, program size or program speed. In some cases optimizing for one aspect means the other suffers.

Use the guidelines in Table 3, except where they are contradicted. Intrinsic functions may improve performance, but they increase the size of your module. Unless noted, these options are not set by default.

Option	Optimize for Speed	Optimize for Size
/O10, /O20, /O30, /O40 Turns on optimization.	Yes	Yes
/Oi- Does not inline user functions. Not inlining may reduce module size especially if the inlined functions consist of small pieces of code. When /O1 is specified, this is the default. When /O1 or O2 are specified, /Oi+ becomes the default.	No	Yes
/Oi+ Inlines user functions.	Yes	No
/Ti- Does not generate debug information, which would increase module size.	No	Yes

Run-Time Limits

The maximum amount of storage of any single variable (such as a string or array) is 16 773 104 bytes

The maximum length of a command passed to the system function is 32 702 bytes

The maximum size of dynamic heap storage is 4 gigabytes

A very large memory allocation may cause a system crash if there is insufficient auxiliary storage on your system. A 4 gigabyte memory allocation requires more than 4 gigabytes of available DASD. The AS/400 Work System Status (WRKSYSSTS) command shows auxiliary storage usage.

The maximum size of fast heap storage is 16 711 600 bytes

The maximum size of a single heap allocation is 16 711 568 bytes

The initial default size of the fast heap is 64 944 bytes

The maximum auto storage is 16MB and there is a recursion limit of around 21743 levels deep

Part 3. Working with AS/400 Files, Devices, Pointers, and Locales

This part describes how to:

- Retrieve external file descriptions from AS/400
- Work in disconnected mode
- Include externally described physical and logical database files
- Convert _DecimalT template classes and zoned decimal data
- Use physical and logical database files and distributed data
- Use commitment control
- Use display files, printer files, ICF files, tape files, diskette files, and save files
- Use the device attributes feedback area
- Declare and use AS/400 pointers
- Use international locales

Chapter 7. Using Externally Described Files

The AS/400 database files you create with a field-level description are referred to as externally described files. The description of the fields and their arrangement is kept in a database file rather than your program.

This section discusses:

- “Retrieving External AS/400 File Descriptions”
- “Understanding Compiler-Generated Output” on page 94
- “Using Externally Described Physical and Logical Database Files” on page 96
- “Generating Typedefs from Externally Described Device Files” on page 104
- “Understanding DDS-to-C++ Data Type Mapping” on page 113
- “Working in Disconnected Mode with External AS/400 File Descriptions” on page 115
- “Including Externally Described Files” on page 118

Note: All examples in this section use DDS as the source for the external AS/400 file description.

Retrieving External AS/400 File Descriptions

Externally described files are files that have their field descriptions stored as part of the file. The file description includes information about the type of file (such as data or device), record formats, and a description of each field and its attributes.

You can create an externally described database file using the:

- DB2/400 database (The *SQL Database Programming* contains information about DB2/400)
- Interactive Data Definition Utility (IDDU) (The *IDDU Use* contains information about IDDU)
- Data Description Specifications (DDS) source (The *DDS Reference* contains information about DDS)

You must establish a connection between your Windows workstation and your AS/400 system before you can retrieve external AS/400 file descriptions. See the *C++ User's Guide* for information on connecting your workstation to AS/400.

Once a connection to AS/400 is established, you can use the **#pragma mapinc** directive to retrieve the external AS/400 file descriptions for your program.

Referencing an AS/400 Connection

You can retrieve the external AS/400 file descriptions from different AS/400 machines. Set up a separate AS/400 connection to each machine through the CTTCONN connection command. See the *C++ User's Guide* for information on using the CTTCONN command to connect to AS/400.

Writing a C++ Program that Retrieves a Record Format Layout

The compiler automatically creates C++ structure typedefs from external file descriptions when you use the **#pragma mapinc** directive.

The syntax for this pragma is:

```
#pragma mapinc ( 'include_name', 'LIBL/' file_name (
                *LIBL/
                *CURLIB/
                library_name/
                d z
*ALL          ) " , 'options', "
                p _P
                'format_name
                "
1BYTE_CHAR   , 'union_type_name'
                , 'prefix_name'
```

See the *C++ Language Reference* for information on this pragma.

This pragma only identifies the file formats and fields to the compiler; it does not include the file description in the program. To include a file description, you reference the *"include_name"* parameter on the **#include** directive.

To include a type definition of the input fields for the record format FMT from the file EXAMPLE/TEST, use statements like these (in the order shown), in your program:

```
#pragma mapinc("tempname", "EXAMPLE/TEST(FMT)", "input", "d")
.
.
.
#include "tempname"
```

The **#pragma mapinc** directive:

- Identifies the file and record formats to generate typedefs

- Along with its associated **#include** directive, causes the compiler to automatically generate typedefs from the record formats specified in the external file descriptions, and include them in your program

Include Name

You can specify a fully qualified path or a relative path for the *include_name* parameter, but it must be a valid Windows include file name. An include name for a fully qualified path is "c:/abc/aaa.h". If you want to use a backslash character (\) you must use a

double-backslash (\\) character. The double-backslash character (\\) allows the compiler to differentiate between control-escape sequences and include path names. See the *C++ Language Reference* for information on file inclusion.

Record Format Name

A *record format* is a description of all the fields including the arrangement of these fields within a record. You can include a record format from an externally described file in your program by providing its name on the **#pragma mapinc** directive. You can provide more than one format name, or you can specify the special value **ALL* to include all record formats from the file.

For each format specified on the pragma, the compiler creates a typedef of type `struct`, which describes the fields in the external file.

If the file you are working with contains more than one record format, you can set the format for subsequent I/O operations with the `_Rformat()` function.

If the record format does not contain fields that match the option specified on the pragma, (*input*, *output*, *both*, *key*, *indicators*, or *nullflds*), this comment appears after the header description:

```
// FORMAT HAS NO FIELDS OF TYPE(S) : XXXXXX
```

The values `XXXXXX` are the options specified on the pragma.

Note: If the format names contain characters that are not valid for C++ identifiers, they are translated to the underscore (`_`) character. If you use the special characters `@`, `#`, or `$` in a format name, those characters are changed to lowercase *a*, *p*, or *d* respectively.

Alignment of Fields in Structures

All fields that are defined in C++ structures are aligned on their natural boundaries. `int` fields are four bytes long and are aligned on four-byte boundaries. The compiler automatically inserts padding between structure members to enforce this alignment.

A packed structure does not enforce alignment of its members. This saves memory by reducing the padding. The AS/400 data management file system does not align record format fields on their natural boundaries. Record formats do not contain padding.

If the fields defined in the external AS/400 file descriptions are aligned on their natural boundaries by default (all are character fields, for example), you can use the typedef that is generated without packing the structure.

To avoid an alignment problem, specify the `_P` option to generate a packed structure. To include a packed type definition structure of `input` and `key` fields for the record format `custrec` from the file `EXAMPLE/CUSTMSTL`, use statements like these (in the order shown) in your program:

```
#pragma mapinc("custmf", "EXAMPLE/CUSTOMSTL(custrec)", "input key", "_P")
.
.
.
#include "custmf"
```

Record Field Names

Field names in the typedefs that are generated correspond to the name of the field in the DDS. The **ALIAS** keyword is supported, and puts the alias field name into the typedef that is generated.

Some of the special characters that are supported in DDS variable names are not supported by the compiler and C run-time library. The characters @, #, and \$ are valid DDS field names, but are not valid as C++ identifiers. These characters are changed to lowercase *a*, *p*, and *d*, respectively in the typedef that is generated.

Variable-Length Fields

For variable-length fields, field information is generated as a length field followed by the data field. A structure that is generated for variable-length fields is:

```
typedef struct {
    . . .                // fixed-length fields information
    . . .
    #pragma pack(1)
    struct { short len;    // Length of data
            char data[5]; // Field data
    } var_field_name;
    #pragma pack()
    . . .
    . . .
} LIBRARY_FILE_FORMAT_tag_t;
```

Understanding Compiler-Generated Output

All the generated information for the external AS/400 file description is placed in a single ASCII text file, which is then processed by the preprocessor as a standard include file.

Header Description

For each format retrieved, an associated header description in the form of a C++ comment is generated. It contains:

- File and library name of the external file
- File type (physical, logical, or device)
- Record format name
- Record format level ID (level-checking information)

The following **#pragma mapinc** and **#include** directives

```
#pragma mapinc("custmf", "MYLIB/CUSTOMSTL(CUSREC)", "both key", "d")
#include "custmf"
```

create the following header description:

```
// PHYSICAL FILE : MYLIB/CUSTOMSTL
// FILE LAST CHANGE DATE : 1996/10/12
// RECORD FORMAT : CUSREC
// FORMAT LEVEL IDENTIFIER : 4E9D9ACA60E00
```

The DDS source for the customer master file `custmstl` is:

```

A          R CUSREC                TEXT('Customer master record')
A          CUST                    5    TEXT('Customer number')
A          NAME                    20    TEXT('Customer name')
A          ADDR                    20    TEXT('Customer address')
A          CITY                    20    TEXT('Customer city')
A          STATE                   2     TEXT('State abbreviation')
A          ZIP                     5  0  TEXT('Zip code')
A          ARBAL                   10  2  TEXT('Accounts receivable balance')
A          K CUST
```

Type Definition Structure

For each format specified on the **#pragma mapinc** directive, the compiler creates one or more C++ type definition structures (typedef of type `struct`), which describes the fields in the record format. You indicate the type of fields to include in the typedef structures in your program by the option you select on the *options* parameter.

The type definition of type structure is:

```
typedef struct {
    .
    .
    .
} LIBRARY_FILE_FORMAT_tag_t;
```

Parameters of the **#pragma mapinc** directive are used to create the name of the created type. `LIBRARY`, `FILE`, and `FORMAT` are the *library_name*, *file_name*, and *format_name*, respectively, that are, specified on the `pragma`. These names are folded to uppercase unless quoted names are used. The library and file names can be replaced by your own *prefix_name* as specified on the `pragma`.

If the *prefix_name* parameter is specified in the `pragma`, it is used in the first part of the generated typedef structure name in place of `LIBRARY_FILE` name.

Any characters not recognized as valid by the C++ language that appear in library and file names are translated to the underscore (`_`) character. If you use the special characters `@`, `#`, or `$` in a format name, those characters are changed to lowercase *a*, *p*, or *d* respectively.

The tag on the structure name indicates the type of fields included in the structure definition. The possible values for the tag are:

Option	Tag
input	i
output	o
both	both
key	key
indicators	indic
nullflds	nmap,nkmap
lvlchk	lvlchk

If the option `lvlchk` is specified, the name of the array of the structure type created is `_LVLCHK_T`. This naming convention is not used for the other options listed above.

To include external file descriptions for more than one format, specify more than one format name (`format1 format2`) or `(*ALL)` on the **#pragma mapinc** directive. A header description and type definition structures are created for each format.

You can compile your C++ program with the compiler options `/L+` and `/Li+` to see the typedefs in your compiler listing. The type definitions are also generated in the listing if you specify the compiler options `/L+` and `/Lj+`. You can save the external AS/400 file descriptions as a Windows file by using the compiler option `/ASd+`. You can then browse the external AS/400 file descriptions and compile your source file in disconnected mode. See “Working in Disconnected Mode with External AS/400 File Descriptions” on page 115 for more information about compiling in disconnected mode.

Level Checking

If you specify the `lvlchk` option on the **#pragma mapinc** directive, the compiler creates the type `_LVLCHK_T` (array of structures), and a variable of type `_LVLCHK_T`. `_LVLCHK_T` is initialized so that each array element contains the level-check information for the corresponding formats specified on the **#pragma mapinc** directive. The last array element always contains two empty strings, one for each field of the structure. The name of the variable is `LIBRARY_FILE_INCLUDE_lvlchk`, where `LIBRARY`, `FILE`, and `INCLUDE` are the `library_name`, `file_name`, and `include_name`, respectively. The level-check information can be used to perform a level check on the file when it is opened.

If you specify the `lvlchk` option on the *varparm* parameter of the `_Ropen()` function and the makeup of the file is changed, the file pointer on the `_Ropen()` function returns `NULL`.

See “Level Check” on page 100 for an example of using the `lvlchk` option.

Using Externally Described Physical and Logical Database Files

To include external physical or logical database file descriptions, use the `input` or `both` options followed by one or more of these options: `key`, `nullflds`, or `lvlchk` on the **#pragma mapinc** directive.

Database files do not support the options `output` or `indicators`. If you specify the `output` or `indicators` options for a database file, the compiler issues an error message.

Note: You can include external file descriptions for Distributed Data Management (DDM) files using the same method described for database files.

Input

Input and output buffers for database files have the same format. When you specify the `input` option, the fields defined as `INPUT` or `BOTH` in the externally described database file are included in the typedef structure.

To include the input fields in the include file "myinc" use statements like these in your program:

```
#pragma mapinc("myinc", "XCPTMP/dds3f016(fmt)", "input")
#include "myinc"
```

The DDS source for the physical file `dds3f016` is:

```
      A          R FMT
      A          NAME          20A
      A          ADDRESS       50A
      A          PHONE         8A
```

The typedef structure generated by the `input` option is:

```
// PHYSICAL FILE : XCPTMP/DDS3F016
// FILE LAST CHANGE DATE : 1996/06/28
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 2913666EE7217
typedef struct {
    char NAME[20];
    char ADDRESS[50];
    char PHONE[8];
} XCPTMP_DDS3F016_FMT_i_t;
```

Both

When you specify the `both` option, the fields defined as `BOTH` or `INPUT` for logical files, or `BOTH` for physical files, are included in the typedef structure definition.

To include the `BOTH` fields in the include file "incfile", use statements like these in your program:

```
#pragma mapinc("incfile",
              "XCPTMP/MDDSLF1(FMT)",
              "both"
            )
#include "incfile"
```

The DDS source for the physical file `MDDSPF1` is:

```

A          R FMT
A          NAME          20A
A          ID            6S
A          ADDRESS      50A
A          PHONE        8A
A          AGE           3S
A          SALARY       9P 2
A          PASSWORD     10A

```

The DDS source for the logical file MDDSLF1 is:

```

A          R FMT                      PFILE(MDDSPF1)
A          NAME
A          ID
A          PASSWORD                  I

```

The typedef structure generated by the both option is:

```

// LOGICAL FILE : XCPTEMP/MDDSLF1
// FILE LAST CHANGE DATE : 1996/06/28
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 2CE7F4B2930A4
typedef struct {
    char NAME[20];
    char ID[6];                // ZONE SPECIFIED IN DDS
                                // REPLACED BY CHARACTER TYPE
    char PASSWORD[10];
} XCPTEMP_MDDSLF1_FMT_both_t;

```

Key

To include a separate structure type definition for the **KEY** fields in a format, specify the **key** option on the **#pragma mapinc** directive. Comments are listed beside the fields in the typedef structure definition to indicate how the **KEY** fields are defined in the externally described file.

To include the **KEY** fields in the include file "myinc", use statements like these in your program:

```

#pragma mapinc("myinc", "XCPTEMP/dds3f060a(fmt)", "key")
#include "myinc"

```

The DDS source for the physical file dds3f060a is:

```

A          R FMT
A          NAME          20A
A          ADDRESS      50A
A          PHONE        8A
A          K NAME
A          K ADDRESS
A          K PHONE                      DESCEND

```

The typedef structure generated by the key option is:

```

// PHYSICAL FILE : XCPTEMP/DDS3F060A
// FILE LAST CHANGE DATE : 1996/06/29
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 2913666EE7217
typedef struct {
    char NAME[20];
                                // DDS - ASCENDING
                                // STRING KEY FIELD
    char ADDRESS[50];
                                // DDS - ASCENDING
                                // STRING KEY FIELD
    char PHONE[8];
                                // DDS - DESCENDING
                                // STRING KEY FIELD
} XCPTEMP_DDS3F060A_FMT_key_t;

```

Null-Capable Fields

When the option `nullflds` is specified in the **#pragma mapinc** directive, the compiler creates a separate typedef structure representing the null field map for the record format. See “Null-Capable Fields” on page 139 for information on null capable fields.

To include the null field maps in the include file `"myinc"`, use statements like these in your program:

```

#pragma mapinc("myinc",
              "XCPTEMP/nullfldex(fmt)",
              "nullflds"
            )
#include "myinc"

```

The DDS source for the physical file `nullfldex` is:

```

A          R FMT
A          NAME          20A
A          ADDRESS       50A
A          PHONE         8A
A          AMEX_NO       15A          ALWNULL
A          AMEX_EXPDT    L           ALWNULL
A          VISA_NO       13A          ALWNULL
A          VISA_EXPDT    L           ALWNULL
A          K AMEX_NO
A          K VISA_NO

```

The typedef structure generated by the `nullflds` option is:

```

// PHYSICAL FILE : XCPTEMP/NULLFLDEX
// FILE LAST CHANGE DATE : 1996/06/29
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 438F151420131
typedef struct {
    unsigned char NAME;
    unsigned char ADDRESS;
    unsigned char PHONE;
    unsigned char AMEX_NO;
    unsigned char AMEX_EXPDT;
    unsigned char VISA_NO;
    unsigned char VISA_EXPDT;
} XCPTEMP_NULLFLDEX_FMT_nmap_t;
typedef struct {
    unsigned char AMEX_NO;
    unsigned char VISA_NO;
} XCPTEMP_NULLFLDEX_FMT_nkmap_t;

```

Note: The null field maps contain one byte for all fields in the record format, not just null-capable fields.

Level Check

If you specify the `lvlchk` option on the `#pragma mapinc` directive, the compiler produces level-check information.

To generate information needed to perform a level check during an open operation, use statements like these in your program:

```

#pragma mapinc("ORDERFILE",
              "MYLIB/ORDERL(ORDHDR ORDDTL)",
              "key lvlchk",
              "_P"
              )
#include "ORDERFILE"

```

The DDS source for the physical file `ORDHDR` is:

```

A      R ORDHDR
A      ITEM_NAME      20A
A      ITEM_NUM       10S
A      UNIT_PRICE     5B 2
A      ITEM_DESC      50A
A      K ITEM_NUM

```

The DDS source for the physical file `ORDDTL` is:

```

A      R ORDDTL
A      ITEM_NUM      10S
A      ORDER_NUM     8S
A      CUST_NAME     20A
A      CUST_PHONE    10A
A      CUST_ADDR     30A
A      QUANTITY      3B 0
A      TOT_PRICE     7B 2
A      ORDER_DATE    L
A      SHIP_DATE     L

```

The DDS source for the logical file is:

```

A      R ORDHDR      PFILE(ORDHDR)
A      ITEM_NAME
A      ITEM_NUM
A      UNIT_PRICE
A      K ITEM_NUM
A*
A      R ORDDTL      PFILE(ORDDTL)
A      ITEM_NUM
A      ORDER_NUM
A      QUANTITY
A      TOT_PRICE
A      K ITEM_NUM

```

The typedef structures generated by the preceding pragma are:

```

// LOGICAL FILE : MYLIB/ORDERL
// FILE LAST CHANGE DATE : 1996/06/30
// RECORD FORMAT : ORDHDR
// FORMAT LEVEL IDENTIFIER : 29F3AF670949B
typedef _Packed struct {
    char ITEM_NUM[10];

                                // DDS - ASCENDING
                                // SIGNED NUMERIC KEY FIELD
} MYLIB_ORDERL_ORDHDR_key_t;

// LOGICAL FILE : MYLIB/ORDERL
// FILE LAST CHANGE DATE : 1996/06/30
// RECORD FORMAT : ORDDTL
// FORMAT LEVEL IDENTIFIER : 2745E7603896B
typedef _Packed struct {
    char ITEM_NUM[10];

                                // DDS - ASCENDING
                                // SIGNED NUMERIC KEY FIELD
} MYLIB_ORDERL_ORDDTL_key_t;

#ifndef __LVLCHK__
#define __LVLCHK__
typedef struct _LVLCHK_ {

```

```

    unsigned char format_name[10];
    unsigned char sequence_no[13];
    _LVLCHK_(const char * pformat, const char * pseq) {
        for (short int i=0; i < 10; format_name[i]=pformat[i++]);
        for (i=0; i < 13; sequence_no[i]=pseq[i++]);};
} _LVLCHK_T[T];
#endif

_LVLCHK_T MYLIB_ORDERL_ORDERFILE_lvlchk = {
    _LVLCHK_("ORDHDR", "29F3AF670949B"),
    _LVLCHK_("ORDDTL", "2745E7603896B"),
    _LVLCHK_(" ", " ")
};

```

Generating Typedefs from Logical Database Files

To generate typedefs corresponding to a logical database file, specify more than one record format name or (*ALL) on the **#pragma mapinc** directive. If you specify multiple formats, the compiler creates a header description and a typedef structure definition for each format.

If you specify a *"union_type_name"* parameter on the pragma, the compiler creates a union typedef structure definition. The typedef union contains the typedef structures created for each format. The typedef structures created for KEY fields when the key option is specified are not included in the union definition. The name of the union definition is *union_type_name_t*. The name you provide for the *union_type_name* parameter is not folded to uppercase.

To include INPUT and KEY fields in the logical file "ORDERL", use statements like these in your program:

```

#pragma mapinc("ORDERFILE",
              "MYLIB/ORDERL(*ALL)",
              "input key",
              "_P",
              "Order_Info",
              "Stationary"
              )
#include "ORDERFILE"

```

The DDS source for the physical file ORDHDRP is:

```

A          R ORDHDR
A          ITEM_NAME      20A
A          ITEM_NUM       10S
A          UNIT_PRICE     5B 2
A          ITEM_DESC      50A
A          K ITEM_NUM

```

The DDS source for the physical file ORDDTLP is:

```

A      R ORDDTL
A      ITEM_NUM      10S
A      ORDER_NUM     8S
A      CUST_NAME     20A
A      CUST_PHONE    10A
A      CUST_ADDR     30A
A      QUANTITY      3B 0
A      TOT_PRICE     7B 2
A      ORDER_DATE    L
A      SHIP_DATE     L

```

The DDS source for the logical file is:

```

A      R ORDHDR      PFILE(ORDHDRP)
A      ITEM_NAME
A      ITEM_NUM
A      UNIT_PRICE
A      K ITEM_NUM
A*
A      R ORDDTL      PFILE(ORDDTLP)
A      ITEM_NUM
A      ORDER_NUM
A      QUANTITY
A      TOT_PRICE
A      K ITEM_NUM

```

The compiler creates typedefs for the logical file "ORDERL" from two record formats specified, using the `input` and `key` options and a typedef union with the tag `buffer_t`.


```

// LOGICAL FILE : MYLIB/ORDERL
// FILE LAST CHANGE DATE : 1996/06/30
// RECORD FORMAT : ORDHDR
// FORMAT LEVEL IDENTIFIER : 29F3AF670949B
typedef _Packed struct {
    char ITEM_NAME[20];
    char ITEM_NUM[10];           // ZONED SPECIFIED IN DDS
                                // REPLACED BY CHARACTER TYPE

    char UNIT_PRICE[4];
} Stationary_ORDHDR_i_t;

typedef _Packed struct {
    char ITEM_NUM[10];

                                // DDS - ASCENDING
                                // SIGNED NUMERIC KEY FIELD
} Stationary_ORDHDR_key_t;

// LOGICAL FILE : MYLIB/ORDERL
// FILE LAST CHANGE DATE : 1996/06/30
// RECORD FORMAT : ORDDTL
// FORMAT LEVEL IDENTIFIER : 2745E7603896B
typedef _Packed struct {
    char ITEM_NUM[10];           // ZONED SPECIFIED IN DDS
                                // REPLACED BY CHARACTER TYPE

    char ORDER_NUM[8];          // ZONED SPECIFIED IN DDS
                                // REPLACED BY CHARACTER TYPE

    short int QUANTITY;
    char TOT_PRICE[4];
} Stationary_ORDDTL_i_t;

typedef _Packed struct {
    char ITEM_NUM[10];

                                // DDS - ASCENDING
                                // SIGNED NUMERIC KEY FIELD
} Stationary_ORDDTL_key_t;

typedef union {
    Stationary_ORDHDR_i_t    Stationary_ORDHDR_i;
    Stationary_ORDDTL_i_t   Stationary_ORDDTL_i;
} Order_Info_t;

```

Note: A typedef union is not created for the KEY fields.

Generating Typedefs from Externally Described Device Files

To include external device file descriptions, use `lvlchk` and one of the following options: `input`, `input indicators`, `output`, `output indicators`, `both`, or `both indicators`.

Device files have separate input and output buffer formats. The compiler creates a separate typedef structure for each input and output format.

Device files do not support the `key` or `nullflds` options. If you specify either the `key` option or the `nullflds` option in a device file, the compiler issues an error message.

Input

You specify the `input` option when you want to include the fields defined as `INPUT` or `BOTH` in the externally described device file. Response indicators are included if the DDS keyword **INDARA** is not specified.

To include only the `INPUT` capable fields in the generated typedef, in the include file "myinc", use statements like these in your program:

```
#pragma mapinc("myinc", "XCPTTEMP/dds3ft015a(rcvfmt)", "input")
#include "myinc"
```

The DDS source for the ICF file `dds3ft015a` is:

```
A          R RCVFMT
A          NAME          10A
A          ADDRESS       20A
```

See "Using Intersystem Communication Function (ICF) Files" on page 162 for information on ICF files. A blank usage field defaults to B. For a display file the usage field defaults to zero.

The typedef structure generated by the `input` option is:

```
// DEVICE FILE : XCPTTEMP/DDS3FT015A
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : RCVFMT
// FORMAT LEVEL IDENTIFIER : 108636CF7216A
typedef struct {
    char NAME[10];
    char ADDRESS[20];
} XCPTTEMP_DDS3FT015A_RCVFMT_i_t;
```

Output

Specify the `output` option when you want to include fields defined as `OUTPUT` or `BOTH` in the externally described device file. Option indicators are included in the typedef structure if the DDS keyword **INDARA** is not specified.

To include only the `OUTPUT` capable fields in the generated typedef, in the include file "MYINC", use statements like these in your program:

```
#pragma mapinc("MYINC",
               "XCPTTEMP/DISPLAY1(fmt)",
               "output",
               "z"
            )
#include "MYINC"
```

The DDS source for the display file DISPLAY1 is:

```
A          R FMT
A          3 4'ENTER YOUR USERID: '
A          8A B +3
A          4 4'ENTER YOUR PASSWORD: '
A          10A I +3
A          5 4'NAME: '
A          20A O +3
A          6 4'ADDRESS: '
A 10      ADDRESS      30A O +3
A          7 4'PHONE NUMBER: '
A 20      PHONE        8A O +3
A          8 4'SALARY: '
A          9S 2O +3
```

The typedef structure generated by the output option is:

```
// DEVICE FILE : XCPTEMP/DISPLAY1
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 1E3A24E75D557
typedef struct {
    char IN10;
    char IN20;
    char USERID[8];
    char NAME[20];
    char ADDRESS[30];
    char PHONE[8];
    char SALARY[9];
                                // ZONED SPECIFIED IN DDS
                                // REPLACED BY CHARACTER TYPE
} XCPTEMP_DISPLAY1_FMT_o_t;
```

Both

Option and response indicators are included in the typedef structures if the DDS keyword **INDARA** is not specified in the external file description. If the **INDARA** keyword is specified, the compiler creates a separate typedef for the indicators.

If the `both` option is specified, if there are no indicators in the DDS, and if all fields are defined as `INPUT` or all fields are defined as `OUTPUT`, only one typedef structure (`both_t`) is generated.

If your record format has different type fields (`INPUT`, `OUTPUT`, `BOTH`), the compiler creates two typedef structures. If you are including the external file description for only one record format, the compiler creates a typedef `union` containing the two typedefs. The name of this typedef `union` is `LIBRARY_FILE_FMT_both_t`. If you specify a `union_type_name` parameter on the `#pragma mapinc` directive, the compiler creates the name `union_type_name_t`.

To use the `both` option, use statements like these in your program:

```

#pragma mapinc("myinc",
              "XCPTMP/DISPLAY1(fmt)",
              "both",
              "z",
              "Salary_Info"
            )
#include "myinc"

```

The DDS source for the display file DISPLAY1 is:

```

A          R FMT
A          3 4'ENTER YOUR USERID: '
A          USERID      8A B  +3
A          4 4'ENTER YOUR PASSWORD: '
A          PASSWORD    10A I  +3
A          5 4'NAME: '
A          NAME        20A O  +3
A          6 4'ADDRESS: '
A 10       ADDRESS     30A O  +3
A          7 4'PHONE NUMBER: '
A 20       PHONE       8A O  +3
A          8 4'SALARY: '
A          SALARY      9S 20  +3

```

The typedef structure generated by the both option is:

```

// DEVICE   FILE : XCPTMP/DISPLAY1
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 1E3A24E75D557
typedef struct {
    char USERID[8];
    char PASSWORD[10];
} XCPTMP_DISPLAY1_FMT_i_t;

typedef struct {
    char IN10;
    char IN20;
    char USERID[8];
    char NAME[20];
    char ADDRESS[30];
    char PHONE[8];
    char SALARY[9];
    // ZONED SPECIFIED IN DDS
    // REPLACED BY CHARACTER TYPE
} XCPTMP_DISPLAY1_FMT_o_t;

typedef union {
    XCPTMP_DISPLAY1_FMT_i_t  XCPTMP_DISPLAY1_FMT_i;
    XCPTMP_DISPLAY1_FMT_o_t  XCPTMP_DISPLAY1_FMT_o;
} Salary_Info_t;

```

Level Check

Level-check information produced on a device file when it is opened is the same information that is produced on a database file when it is opened. See "Level Check" on page 100 for information on this option.

Indicators

When you create a device file, you can define the indicators for a record format either as part of a separate indicator area independent of the record format, or as fields in the record format.

Separate Indicator Area

To use indicators in a separate area, you specify all of the following:

- The DDS keyword **INDARA** in the external description of the file

- The option `indicators` on the **#pragma mapinc** directive

- The parameter `indicators=y` on `_Ropen` when opening the file

You set the address of the separate indicator area using the `_Rindara()` function before performing I/O operations. If you specify `indicators` on the **#pragma mapinc** directive and do not use the keyword **INDARA** in your external file description, you receive a warning message at compile time.

If the `indicators` option is specified on the pragma and the **INDARA** keyword is specified in the DDS, the compiler creates a 99-byte structure definition. This structure definition contains a field declaration for each indicator defined in the DDS. The name of each field is `INXX`, where `XX` is the DDS indicator number. The sequence of bytes between indicators is defined as `INXX_INYY`, where `XX` is the first undefined byte and `YY` is the last undefined byte in a sequence.

If indicators are defined for a record format, and the **INDARA** keyword is specified in the DDS, but the `indicators` option is not specified on the pragma, a list of the indicators in the DDS is included as a comment in the header description.

The following program shows a typedef of a structure for the indicators in the format `FMT` of the file `EXINDIC/TEST`. The external file description contains three indicators: `IN50`, `IN51`, and `IN99`. The keyword **INDARA** is also specified in the DDS for the file:

```
#pragma mapinc("example", "exindic/test(fmt)", "indicators")
#include "example"
```

The DDS source for the display file `test` is:

```

A                                INDARA
A      R FMT
A                                CF01(50)
A                                CF02(51)
A                                CF03(99 'EXIT')
A                                1 35 'PHONE BOOK'
A                                DSPATR(HI)
A                                7 28 'Name:'
A      NAME          11A  I  7 34
A                                9 25 'Address:'
A      ADDRESS      20A  O  9 34
A                                11 25 'Phone #:'
A      PHONE_NUM    8A  O  11 34
A                                23 34 'F3 - EXIT'

```

The structure definition for the indicators is:

```

// DEVICE   FILE : EXINDIC/TEST
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 371E00A681EA7
typedef struct {
    char IN01_IN49[49];      // UNUSED INDICATOR(S)
    char IN50;
    char IN51;
    char IN52_IN98[47];     // UNUSED INDICATOR(S)
    char IN99;
} EXINDIC_TEST_FMT_indic_t;

```

Indicators in the File Buffer

If you do not specify the **INDARA** keyword in the DDS, the indicators are part of the file buffer being read or written.

Only the indicators that are used are declared in the type definition of the structure. They are declared as `char`, and are created when one of the `input`, `output`, or both options is specified on the **#pragma mapinc** directive.

To include only the `OUTPUT` capable fields and option indicators in the generated typedef, use statements like these in your program:

```

#pragma mapinc("myinc",
              "XCPTMP/dds3f063(fmt sflctlf)",
              "output"
            )
#include "myinc"

```

The DDS source for the display file `dds3f063` is:

```

A          R FMT                SFL
A          NAME                20A   4 10
A          ADDRESS             50A   5 13
A          PHONE               8A    6 11
A*
A          R SFLCTLF           SFLCTL(FMT)
A                               SFLPAG(5)
A                               SFLSIZ(17)
A          09                   SFLDSP
A                               SFLDSPCTL
A                               1 5'NAME (ANDREW) '
A                               2 5'ADDRESS (1 PIKES PEAK) '
A                               3 5'PHONE (222-2222) '

```

The typedef structure generated by the `output` option is:

```

// DEVICE FILE : XCPTEMP/DDS3F063
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : FMT
// FORMAT LEVEL IDENTIFIER : 1054582DDA3EF
typedef struct {
    char NAME[20];
    char ADDRESS[50];
    char PHONE[8];
} XCPTEMP_DDS3F063_FMT_o_t;

// DEVICE FILE : XCPTEMP/DDS3F063
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : SFLCTLF
// FORMAT LEVEL IDENTIFIER : 0230713C3ED9A
typedef struct {
    char IN09;
} XCPTEMP_DDS3F063_SFLCTLF_o_t;

```

Using Multiple Record Formats in a Device File

If you specify `*ALL`, or more than one record format, on the `format_name` parameter on the `#pragma mapinc` directive for a device file, the compiler creates a separate header description and a typedef structure definition for each format.

If you specify multiple formats and the `input` or `output` option, the compiler creates one structure for each format if the format contains fields corresponding to the option.

If the `union_type_name` parameter is specified, the compiler creates a union typedef, which contains the typedef structures created for all the formats.

To include automatically generated typedefs corresponding to the record formats of the device file `testfile`, use statements like these in your program:

```

#pragma mapinc("example",
              "testfile(*all)",
              "output",
              "z 1BYTE_CHAR",
              "Computer_Club_Info"
            )
#include "example"

```

The DDS source for the display file testfile is:

```

A          R HEADER                      CA10(10)
A          1 3'GROUP ID: '
A          GROUP_ID          6A B +3
A          +3'GROUP NAME: '
A          GROUP_NAME      20A O +3
A          +3'BUDGET: '
A 20      BUDGET            6S 2O +3
A          +3'PASSWORD: '
A          GROUP_PSWD      6A I +3
A*
A          R DETAIL                      SFL
A          MBR_ID           6A O 8 3
A          MBR_NAME        20A B +3
A          MBR_SEX         1A B +3
A          MBR_ADDR        30A B +3
A          MBR_PHONE       8A B +3
A          MBR_PSWD        6A I +3
A*
A          R DETAIL_CTL              SFLCTL(DETAIL)
A          SFLPAG(5)
A          SFLSIZ(10)
A          SFLDSP
A          SFLDSPCTL
A          3 5'Instructions: '
A          +2'You can change the member'
A          +1'Name, Sex, Address,'
A          +1'and Phone # fields.'
A          4 20'Can input password, but'
A          +1'it will not be displayed.'

```

The compiler creates the typedef structures from all record formats (*ALL) specified, using the output option and a typedef union with the tag unionex_t:


```

// DEVICE FILE : XCPTEMP/TESTFILE
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : DETAIL_CTL
// FORMAT LEVEL IDENTIFIER : 088A9B6C20C41
typedef struct {
    char IN66;
} XCPTEMP_TESTFILE_DETAIL_CTL_o_t;

typedef union {
    XCPTEMP_TESTFILE_HEADER_o_t      XCPTEMP_TESTFILE_HEADER_o;
    XCPTEMP_TESTFILE_DETAIL_o_t      XCPTEMP_TESTFILE_DETAIL_o;
    XCPTEMP_TESTFILE_DETAIL_CTL_o_t  XCPTEMP_TESTFILE_DETAIL_CTL_o;
} Computer_Club_Info_t;

```

When both is specified as an option, the compiler creates two typedef structures for each format. This program shows the typedef structures created when you include two formats, HEADER and DETAIL, for the device file testfile, and specify the both option:

```

#pragma mapinc("example",
               "testfile(*all)",
               "both",
               "z 1BYTE_CHAR",
               "Computer_Club_Info"
               )
#include "example"

```

The compiler creates two typedef structures for each format:

```

// DEVICE FILE : XCPTEMP/TESTFILE
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : HEADER
// FORMAT LEVEL IDENTIFIER : 12293B264F238
typedef struct {
    char IN10;
    char GROUP_ID[6];
    char GROUP_PSWD[6];
} XCPTEMP_TESTFILE_HEADER_i_t;

typedef struct {
    char IN20;
    char GROUP_ID[6];
    char GROUP_NAME[20];
    char BUDGET[6];
} XCPTEMP_TESTFILE_HEADER_o_t;
// ZONED SPECIFIED IN DDS
// REPLACED BY CHARACTER TYPE

// DEVICE FILE : XCPTEMP/TESTFILE
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : DETAIL
// FORMAT LEVEL IDENTIFIER : 217332D24D3F4
typedef struct {
    char MBR_ID[6];
    char MBR_NAME[20];
}

```

```

    char MBR_SEX;
    char MBR_ADDR[30];
    char MBR_PHONE[8];
    char MBR_PSWD[6];
} XCPTMP_TESTFILE_DETAIL_i_t;

typedef struct {
    char MBR_ID[6];
    char MBR_NAME[20];
    char MBR_SEX;
    char MBR_ADDR[30];
    char MBR_PHONE[8];
} XCPTMP_TESTFILE_DETAIL_o_t;

// DEVICE FILE : XCPTMP/TESTFILE
// FILE LAST CHANGE DATE : 1996/07/05
// RECORD FORMAT : DETAIL_CTL
// FORMAT LEVEL IDENTIFIER : 088A9B6C20C41
typedef struct {
    char IN66;
} XCPTMP_TESTFILE_DETAIL_CTL_both_t;

typedef union {
    XCPTMP_TESTFILE_HEADER_i_t          XCPTMP_TESTFILE_HEADER_i;
    XCPTMP_TESTFILE_HEADER_o_t          XCPTMP_TESTFILE_HEADER_o;
    XCPTMP_TESTFILE_DETAIL_i_t          XCPTMP_TESTFILE_DETAIL_i;
    XCPTMP_TESTFILE_DETAIL_o_t          XCPTMP_TESTFILE_DETAIL_o;
    XCPTMP_TESTFILE_DETAIL_CTL_both_t   XCPTMP_TESTFILE_DETAIL_CTL_both;
} Computer_Club_Info_t;

```

Notes:

1. The field `MBR_ID` in `XCPTMP_TESTFILE_DETAIL_i_t` is an OUTPUT field in the DDS, but is included in the input buffer. A subfile record is a special case in that its input buffer always contains all the fields in the subfile record, even if the fields are output only. The output buffer for a subfile record only contains output-capable fields.
2. In the `DETAIL_CTL` record, the only field is option indicators 66. The compiler creates one typedef structure `both_t`. When the `both` option is specified and all the fields in the record are INPUT or OUTPUT, the compiler creates only one typedef structure `((both_t))`.

Understanding DDS-to-C++ Data Type Mapping

Table 4 on page 114 shows DDS data types and the corresponding C++ declarations that the compiler uses to map fields from externally described files to your program. The compiler creates fields in typedef structures based on the DDS data types in the externally described file. See the *DDS Reference* for information about DDS.

Table 4 (Page 1 of 2). DDS-to-C++ Data Type Mappings

DDS Data Type	Length	Decimal Position	C++ Declaration
Indicator	1	0	char INxx; for used indicator xx char INxx_INyy[n]; for unused indicators xx through yy
A - alphanumeric	1-32766	none	char field[n]; (where n = 1 to 32766)
A - alphanumeric variable length VARLEN keyword	1-32740	none	#pragma pack (1) struct { short len; char data[n]; }; #pragma pack() where n is the maximum length of the field
B - binary	1-4	0	short int field;
B - binary	1-4	1-4	char field[2];
B - binary	5-9	0	int field;
B - binary	5-9	1-9	char field[4];
H - hexadecimal	1	none	char field;
H - hexadecimal	2-32766	none	char field[n]; (where n = 2 to 32766)
H - hexadecimal variable length VARLEN keyword	1-32740	none	#pragma pack (1) struct { short len; char data[n]; }; #pragma pack() where n is the maximum length of field
G - graphic variable length VARLEN keyword	4-1000	none	#pragma pack (1) struct { short len; wchar_t data[n]; }; #pragma pack() (where n = 4 to 1000)
P - binary coded decimal class	1-31	0-31	_DecimalT<n,p> where n is length and p is decimal position on parameter d
S - zoned decimal	1-31	0-31	char field[n]; (where n = 1 to 31)
F - floating point	1	1	float field;
F - floating point	1	1	double field;
J - DBCS only	4-32766	none	char field[n]; (where n = 4 to 32766 and n is an even number)
E - DBCS either	4-32766	none	char field[n]; (where n = 4 to 32766 and n is an even number)
O - DBCS open	4-32766	none	char field[n]; (where n = 4 to 32766)

Table 4 (Page 2 of 2). DDS-to-C++ Data Type Mappings

DDS Data Type	Length	Decimal Position	C++ Declaration
J - DBCS only variable length VARLEN keyword	4-32740	none	<pre>#pragma pack (1) struct { short len; char data[n]; }; #pragma pack()</pre> <p>(where n = 4 to 32740 and n is an even number)</p>
E - DBCS either variable length VARLEN keyword	4-32740	none	<pre>#pragma pack (1) struct { short len; char data[n]; }; #pragma pack()</pre> <p>(where n = 4 to 32740 and n is an even number)</p>
O - DBCS open variable length VARLEN keyword	4-32740	none	<pre>#pragma pack (1) struct { short len; char data[n]; }; #pragma pack()</pre> <p>(where n = 4 to 32740)</p>
T - time	8	none	<pre>char field[8];</pre>
L - date	6, 8, or 10	none	<pre>char field[n]; (where n = 6, 8 or 10)</pre>
Z - time stamp	26	none	<pre>char field[26];</pre>
<p>Note: 1 The C++ declaration (<code>float</code> or <code>double</code>) is based on what is specified in the FLTPCN (floating-point precision) keyword in the DDS: <code>*SINGLE</code> (default) is <code>float</code>, <code>*DOUBLE</code> is <code>double</code>.</p> <p>See the <i>DDS Reference</i> for information on using the database data types and variable length fields.</p>			

Working in Disconnected Mode with External AS/400 File Descriptions

You can specify a compiler option to save the include files generated by the **#pragma mapinc**, directive to your Windows workstation. By saving these files locally, you can recompile the source later, in disconnected mode, and still have the typedefs corresponding to the record formats of AS/400 files available.

When the compiler processes the pragma, it retrieves the requested external AS/400 file descriptions from the specified AS/400 system. It creates a header file that contains C++ typedef structure information corresponding to the requested external AS/400 file descriptions. This Windows file is placed in the directory specified by you in the `include_name` parameter in the pragma. The compiler overwrites any existing file with the same name.

To reference the external AS/400 file descriptions, you must include the specified directory in a valid `#include` search path. You can either include this directory in the `INCLUDE_ASV3R7` or `INCLUDE_ASV3R6` environment variable, (if targeting a different release), or use the `/I` compiler option.

If you compile your C++ source in disconnected mode, all the `#pragma mapinc` directives are ignored by the compiler. You must make sure that the `#include` search path is set up correctly to reference the saved header files that contain the external AS/400 file descriptions.

Compile Options When Working in Disconnected Mode

Before working in disconnected mode, save the external AS/400 file descriptions on your Windows workstation. Table 5 describes the compile options you should consider to work in disconnected mode with include files generated by the `#pragma mapinc` directive.

Option	Description
<code>/ASd-</code>	Do not save the external AS/400 file descriptions locally; delete files at the end of compiling. If you compile using this option in disconnected mode, all <code>#pragma mapinc</code> directives are flagged as errors.
<code>/ASd+</code> (Default)	Save the external AS/400 file descriptions locally to the file whose name is specified in the <code>#pragma mapinc</code> directive. If you compile using this option in the disconnected mode, all <code>#pragma mapinc</code> directives are ignored. The saved external AS/400 file descriptions are used.
<code>/ASdpath</code>	Save the external AS/400 file descriptions locally to the directory specified in the path.

Saving DDS Information Locally

Before you save the generated header file containing the record format typedefs locally, consider:

1. If a fully qualified path is specified in the `include_name` parameter in the `#pragma mapinc` directive, the external AS/400 file descriptions are saved there. A fully qualified path name is:

```
#pragma mapinc("c:/abc/aaa.h", "*LIBL/ABC(REC1)", "input", ...)
```

If the source including this directive is compiled with the `/ASd-` compiler option, no external AS/400 file descriptions are saved.

If the source including this directive is compiled with the `/ASd+` compiler option, the external AS/400 file descriptions are saved in `c:\abc\aaa.h`.

If the source including this directive is compiled with the `/ASdpath` compiler option, (for example, `/ASdc:\abc`), the external AS/400 file descriptions are saved using the path name specified for the `/ASdpath` compiler option, `c:\abc\aaa.h`.

2. If a relative path is specified in the *include_name* parameter, it is always relative to the current directory. A relative path name is:

```
#pragma mapinc("proj1/aaa.h", "*LIBL/FILE(RECl)", "input", ...)
```

If the source including this directive is compiled with the /ASd- compiler option, no external AS/400 file descriptions are saved.

If the source including this directive is compiled with the /ASd+ compiler option, the external AS/400 file descriptions are saved as `c:\source\proj1\aaa.h`, assuming the current directory is `c:\source`.

If the source including this directive is compiled with the /ASdpath compiler option, (for example, /ASdd:\appl), the external AS/400 file descriptions are saved using the path name specified for the /ASdpath compiler option, `d:\appl\proj1\aaa.h`.

3. If the compiler option /ASdpath is used, the path specified by this compiler option is used. If a relative path is specified in the *include_name* parameter, it is always relative to this path. If that path itself is also a relative path, the current directory is used. If the source including this directive is compiled with the /ASdpath compiler option, (for example, /ASd\appl), the external AS/400 file descriptions are saved as `c:\appl\proj1\aaa.h`, assuming that the current directory is `c:\appl`.

Converting `_DecimalT` Template Classes and Zoned Decimal Data

The compiler and the C++ run-time library provide limited support for the ILE C packed decimal data type, through a Binary Coded Decimal Class Library. The *d* parameter of the **#pragma mapinc** directive causes the compiler to create binary coded decimal template-class objects for any packed decimal fields defined in the external AS/400 file descriptions.

If you use the *p* parameter of the **#pragma mapinc** directive, the compiler creates character arrays to store the binary coded decimal values. These character arrays must be converted to a C numeric data type to be used in a C or C++ program. If neither the *p* or *d* parameter is specified, the *d* parameter is assumed.

See the *IBM Open Class Library User's Guide* for examples in using binary coded decimal objects.

Note: You must include the `<bcd.h>` header file if the typedefs generated by the external AS/400 file descriptions contain binary coded decimal objects.

The compiler and C++ run-time library do not support zoned decimal data.

To convert binary coded decimal data stored in character arrays to integer or floating point (`double`), and vice versa, you can use the conversion functions within the `_ConvertDecimal` class. See the *IBM Open Class Library User's Guide* for examples. You can also use the `QXXDTOP()`, `QXXITOP()`, `QXXPTOD()`, and `QXXPTOI()` functions.

To convert zoned decimal data stored in character arrays to integer or floating point (`double`) and vice versa, you can use the `QXXDTOZ()`, `QXXITDZ()`, `QXXZTOD()`, and `QXXZTOI()` functions.

These conversion functions are included with the compiler because you cannot use the Binary Coded Decimal Class Library to convert zoned decimal character arrays. The *C Library Reference* contains examples of each of these functions.

You can use the MI `cpynv()` function to convert binary coded or zoned decimal data to an ILE C++ numeric data type, and to convert an ILE C++ numeric data type to binary coded or zoned decimal data.

Note: If you are performing database I/O operations, you can use a logical file with integer or floating-point fields to redefine packed and zoned fields in your physical file. When you perform an input or output operation through the logical file, the AS/400 system converts the data for you automatically.

Including Externally Described Files

These programs, `STOREPG1` and `STOREPG2`, show how to use the `#pragma mapinc` directive to include externally described database and device files in an application. These programs use the externally described files `STOREINV` and `STOREORD`.

The source code for file `STOREINV` shows the record layout of a grocery master file:

```
A*
A* Grocery Master File -- STOREINV
A*
A      R GROCERY                TEXT('Grocery record')
A      CATEGORY                1A  TEXT('Type of grocery')
A      SERIAL_NUM              3A  TEXT('Grocery serial #')
A      NAME                    20A  TEXT('Grocery name')
A      QTY_ONHAND              3P 0  TEXT('Quantity in stock')
A                                  ALWNULL
A      UNIT                    10A  TEXT('Unit - lbs, pint, etc.')
A                                  ALWNULL
A      QTY_ORDER              3P 0  TEXT('Quantity to order')
A                                  ALWNULL
A      K CATEGORY
A      K SERIAL_NUM
```

The source code for file `STOREORD` shows a display file for order entry:

```

A*
A* Display File for Order Entry
A*
A          INDARA
A      R REQUEST
A          CF03(03 'EXIT')
A          1 30 'Inventory Order' DSPATR(HI)
A          4 5 'Grocery categories:'
A          4 26 '1: Fruits          2: Vegetables'
A          5 26 '3: Dairy Products 4: Processed Food'
A          8 5 'Enter category number:'
A      CATEGORY      1A B  +2 VALUES('1' '2' '3' '4')
A          24 30 'F3 - EXIT' DSPATR(HI)
A*
A      R SFL          SFL
A      SERIAL_NUM    3A  O 10 5
A      NAME          20A O 10 17
A      QTY_ONHAND    3S  00 10 40
A      UNIT          10A O 10 52
A      QTY_ORDER     3S  0B 10 63
A*
A      R SFLCTL      SFLCTL(SFL)
A          SFLPAG(5) SFLSIZ(5)
A          SFLDSP
A          SFLDSPCTL
A          CF03(03 'EXIT')
A          3 5 'F3 - EXIT' DSPATR(HI)
A          7 5 'Serial#'
A          7 17 'Name'
A          6 40 'Quantity'
A          7 40 'in Stock'
A          7 52 'Unit'
A          6 63 'Quantity'
A          7 63 'to Order'

```

Program STOREPG1 populates the database file STOREINV with values for the GROCERY, CATEGORY, SERIAL_NUM, NAME, QTY_ONHAND, UNIT, and QTY_ORDER fields.

The **#pragma mapinc** directive and its associated **#include** directive cause the compiler to create typedefs in your source from the record format specified in file STOREINV.

The source code for file storepg1.cpp is:

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

/* Includes for Physical File STOREINV */

#pragma mapinc("inventory", \

```



```

        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' },
        { '0', '0', '0', '0', '0', '0' };

int main(void)
{
    int                rec_cnt, sizeof_data;
    _RFILE             *fp;
    _RIOFB_T           *fb;
    STORE_STOREINV_GROCERY_nmap_t  null_map;
    STORE_STOREINV_GROCERY_both_t   data [NUM_ITEMS];

    /* Grocery Categories: */
    /* 1 -- Fruits */
    /* 2 -- Vegetables */
    /* 3 -- Dairy products */
    /* 4 -- Processed Food */

    InitData(&data [0], '1', "101", "Apples",          25, "lbs", 0 );
    InitData(&data [1], '1', "102", "Oranges",         12, "lbs", 0 );
    InitData(&data [2], '1', "103", "Kiwi",            0, "lbs", 0 );
    InitData(&data [3], '1', "104", "Bananas",         10, "lbs", 0 );
    InitData(&data [4], '1', "105", "Mangoes",         0, "lbs", 0 );
    InitData(&data [5], '1', "106", "Blueberries",     10, "pints", 0 );
    InitData(&data [6], '1', "107", "Star fruits",      3, "lbs", 0 );
    InitData(&data [7], '1', "108", "Papayas",          9, "lbs", 0 );
    InitData(&data [8], '2', "201", "Lettuce",          8, "heads", 0 );
    InitData(&data [9], '2', "202", "Tomatoes",         5, "lbs", 0 );
    InitData(&data1[0], '2', "203", "Asparagus",        0, "lbs", 0 );
    InitData(&data1[1], '2', "204", "Spinach",          4, "bunches", 0 );
    InitData(&data1[2], '2', "205", "Cauliflower",     12, "heads", 0 );
    InitData(&data1[3], '2', "206", "Broccoli",         5, "heads", 0 );
    InitData(&data1[4], '2', "207", "Leek",             3, "heads", 0 );
    InitData(&data1[5], '2', "208", "Green onions",     6, "lbs", 0 );
    InitData(&data1[6], '3', "301", "Milk",             20, "cartons", 0 );
    InitData(&data1[7], '3', "302", "Eggs",            15, "dozens", 0 );
    InitData(&data1[8], '3', "303", "Yogurt",          30, "cups", 0 );
    InitData(&data1[9], '3', "304", "Cheddar",         5, "lbs", 0 );
    InitData(&data2[0], '3', "305", "Mozzarella",       3, "lbs", 0 );
    InitData(&data2[1], '3', "306", "Chocolate ice cream", 5, "tubs", 0 );
    InitData(&data2[2], '3', "307", "Vanilla ice cream", 3, "tubs", 0 );
    InitData(&data2[3], '3', "308", "Butter",          30, "lbs", 0 );
    InitData(&data2[4], '4', "401", "Potato chips",     15, "bags", 0 );
    InitData(&data2[5], '4', "402", "Microwave popcorn", 10, "bags", 0 );
    InitData(&data2[6], '4', "403", "Chocolate cookies", 35, "bags", 0 );

```

```

InitData(&data2[7], '4', "404", "Nanaimo bars",      20, "dozens", 0 );
InitData(&data2[8], '4', "405", "Cola",            20, "bottles", 0 );
InitData(&data2[9], '4', "406", "SoftDrink",       20, "bottles", 0 );
InitData(&data3[0], '4', "407", "Half moon",       25, "dozens", 0 );
InitData(&data3[1], '4', "408", "Butter tarts",    5, "dozens", 0 );

/* Open file */

fp = _Ropen("STORE/STOREINV", "wr", nullcap = y");

if ( fp == NULL )
{
    printf("File not opened for adding records.\n");
    exit(1);
}

/* Loop to add records to file */

for (rec_cnt=0; rec_cnt < NUM_ITEMS; ++rec_cnt)
{
    /* Set output null map for new record */

    _CLEAR_NULL_MAP(fp, null_map);

if (null_values[rec_cnt].CATEGORY == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, CATEGORY);
if (null_values[rec_cnt].SERIAL_NUM == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, SERIAL_NUM);
if (null_values[rec_cnt].NAME == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, NAME);
if (null_values[rec_cnt].QTY_ONHAND == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, QTY_ONHAND);
if (null_values[rec_cnt].UNIT == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, UNIT);
if (null_values[rec_cnt].QTY_ORDER == '1')
    _SET_NULL_MAP_FIELD(fp, STORE_STOREINV_GROCERY_nmap_t, QTY_ORDER);

    /* Add new record to file */

    sizeof_data = sizeof(data[rec_cnt]);
    fb = _Rwrite(fp, &data[rec_cnt], sizeof_data);
    if ( fb->num_bytes != sizeof_data )
    {
        printf("_Rwrite failed\n ");
        printf("Record is = %s \n", data[rec_cnt].NAME);
        exit(2);
    }
} /* ends for loop that adds new records */

/* Close file */

```

```

    _Rclose(fp);

} /* end of main program */

```

The output of program STOREPG1 is stored in the database file STOREINV is:

CATEGORY	SERIAL_NUM	NAME	QTY_ONHAND	UNIT	QTY_ORDER
1	101	Apples	25	lbs	0
1	102	Oranges	12	lbs	0
1	103	Kiwi	-	-	-
1	104	Bananas	10	lbs	0
1	105	Mangoes	-	-	-
1	106	Blueberries	10	pints	0
1	107	Star fruits	3	lbs	0
1	108	Papayas	9	lbs	0
2	201	Lettuce	8	heads	0
2	202	Tomatoes	5	lbs	0
2	203	Asparagus	-	-	-
2	204	Spinach	4	bunches	0
2	205	Cauliflower	12	heads	0
2	206	Broccoli	5	heads	0
2	207	Leek	3	heads	0
2	208	Green onions	6	lbs	0
3	301	Milk	20	cartons	0
3	302	Eggs	15	dozens	0
3	303	Yogurt	30	cups	0
3	304	Cheddar	5	lbs	0
3	305	Mozzarella	3	lbs	0
3	306	Chocolate ice cream	5	tubs	0
3	307	Vanilla ice cream	3	tubs	0
3	308	Butter	30	lbs	0
4	401	Potato chips	15	bags	0
4	402	Microwave popcorn	10	bags	0
4	403	Chocolate cookies	35	bags	0
4	404	Nanaimo bars	20	dozens	0
4	405	Cola	20	bottles	0
4	406	Softdrink	20	bottles	0
4	407	Half moon	25	dozens	0
4	408	Butter tarts	5	dozens	0

Program STOREPG2 performs these tasks:

1. Requests users to input a grocery category.
2. Initializes a subfile with records from the physical file STOREINV based on the user input.
3. Displays the subfile
4. Lets the user modify the Quantity to Order field.
5. Updates the file STOREINV based on the changes to the Quantity to Order field.

The **#pragma mapinc** directives and their associated **#include** directives cause the compiler to automatically generate typedefs in your source listing from the record formats specified in file STOREINV and STOREORD.

The source code for file storepg2.cpp is:

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

/* Includes for Physical File STOREINV */

#pragma mapinc("inventory", \
              "STORE/STOREINV(GROCERY)", \
              "both key nullflds lvlchk", \
              "d _P 1BYTE_CHAR", "Stock" \
              )

#include "inventory"

/* Includes for Display File STOREORD */

#pragma mapinc("orderEntry", \
              "STORE/STOREORD(*ALL)", \
              "both indicators lvlchk", \
              "_P 1BYTE_CHAR", "Order" \
              )

#include "orderEntry"

/* #defines */

#define PFNAME "STORE/STOREINV"
#define DSPFNAME "STORE/STOREORD"
#define IND_ON '1'
#define NUM_SUBF_RECS 5
#define FALSE 0
#define TRUE 1

/* Local function prototypes */

static void init_subfile(_RFILE *, /* Physical file pointer */
                       _RFILE *, /* Display file pointer */
                       char); /* Grocery category */

static void IntToStr(char *, /* Array to store converted int */
                    int); /* Int to convert to char array */

/*=====*/
/* Function: main */
/* Description: - Requests and processes user input */
/*=====*/

```

```

int main(void)
{
    _RFILE          *pf;
    _RFILE          *dspf;
    _RIOFB_T       *pf_fb;
    _RIOFB_T       *dspf_fb;
    Stock_GROCERY_both_t  pf_buf;
    Stock_GROCERY_key_t   key;
    Order_SFL_i_t        subf_i_buf;
    Order_REQUEST_both_t  userReq;
    Order_REQUEST_indic_t userReqInd;
    Order_SFLCTL_indic_t  sflCtlInd;

    /* Open the physical file and perform level check          */
    if ((pf = _Ropen(PFNAME, "rr+ nullcap = y varparm = (lvlchk)",
                    &Stock_inventory_lvlchk)) == NULL)
    {
        printf("Cannot open file %s\n", PFNAME);
        exit(1);
    }

    /* Open the display file and perform level check          */
    if ((dspf = _Ropen(DSPFNAME,
                    "ar+ indicators=y varparm=(lvlchk)",
                    &Order_orderEntry_lvlchk)) == NULL)
    {
        printf("Cannot open file %s\n", DSPFNAME);
        exit(1);
    }

    /* Register the buffers for the indicator areas          */
    _Rindara(dspf, (char *) &userReqInd);
    _Rindara(dspf, (char *) &sflCtlInd);

    /* Ask user to input a grocery category                  */
    _Rformat(dspf, "REQUEST");
    dspf_fb = _Rwriterd(dspf, &userReq, sizeof(userReq));

    /* If user hits PF3, then exit program                  */
    if (userReqInd.IN03 == IND_ON)
    {
        exit(0);
    }

    /* Initialize the subfile with records from the physical */

```

```

/* file using category provided by user */

init_subfile(pf, dspf, userReq.CATEGORY);

/* Display the subfile so that user can modify the */
/* 'Quantity to Order' field */
_Rformat(dspf, "SFLCTL");
dspf_fb = _Rwriterd(dspf, "", 0);

/* If user hits PF3, then exit program */

if (sflCtlInd.IN03 == IND_ON)
{
    exit(0);
}

/* Based on changes to the subfile, modify the physical file */

key.CATEGORY = userReq.CATEGORY;

_Rformat(dspf, "SFL");
dspf_fb = _Rreadnc(dspf, &subf_i_buf, sizeof(subf_i_buf));

memcpy(key.SERIAL_NUM, subf_i_buf.SERIAL_NUM,
sizeof(key.SERIAL_NUM));

while (dspf_fb->num_bytes != EOF)
{
    pf_fb = _Rreadk(pf, &pf_buf, sizeof(pf_buf), __KEY_EQ,
&key, sizeof(key));

    if (pf_fb->num_bytes == sizeof(pf_buf))
    {
        pf_buf.QTY_ORDER = atoi(subf_i_buf.QTY_ORDER);
        pf_fb = _Rupdate(pf, &pf_buf, sizeof(pf_buf));
        dspf_fb = _Rreadnc(dspf, &subf_i_buf, sizeof(subf_i_buf));
        memcpy(key.SERIAL_NUM, subf_i_buf.SERIAL_NUM,
sizeof(key.SERIAL_NUM));
    }
}

/* Close the files */

_Rclose(pf);
_Rclose(dspf);
}
/* end of main */

/*=====*/

```

```

/*                                                     */
/* Function: init_subfile                               */
/*                                                     */
/* Description: Initialize subfile using records read from a      */
/*               physical file. The records from the physical file */
/*               are selected based on 'category' input by user.   */
/*                                                     */
/* Input: pf      - points to physical file                  */
/*        dspf    - points to display file                  */
/*        category - grocery category, supplied by user in    */
/*                  the main function                       */
/*                                                     */
/* Output: subfile initialized with records                 */
/*                                                     */
/*=====*/

```

```

void init_subfile(_RFILE *pf, _RFILE *dspf, char category)
{
    _RIOFB_T      *fb;
    int            rec_cnt;
    int            haveFirstRec;
    Stock_GROCERY_both_t pf_buf;
    Order_SFL_o_t  subf_o_buf;

    _Rformat(dspf, "SFL");
    haveFirstRec = FALSE;

    while (!haveFirstRec)
    {
        fb = _Rreadk(pf, &pf_buf, sizeof(pf_buf), __KEY_EQ,
                    &category, sizeof(category));

        if (fb->num_bytes != sizeof(pf_buf))
        {
            printf("1st _Rreadk failed \n");
            exit (2);
        }
        else
        {
            /* Want to make sure the record in the physical file does */
            /* not contain null values before using the record          */

            if (!(_QRY_NULL_MAP_FIELD(pf,
                                     Stock_GROCERY_nmap_t,
                                     QTY_ONHAND)))
            {
                memcpy(subf_o_buf.SERIAL_NUM, pf_buf.SERIAL_NUM,
                       sizeof(pf_buf.SERIAL_NUM));
                memcpy(subf_o_buf.NAME, pf_buf.NAME, sizeof(pf_buf.NAME));
                memcpy(subf_o_buf.UNIT, pf_buf.UNIT, sizeof(pf_buf.UNIT));
                IntToStr(subf_o_buf.QTY_ONHAND, pf_buf.QTY_ONHAND);
            }
        }
    }
}

```



```

        IntToStr(subf_o_buf.QTY_ORDER, pf_buf.QTY_ORDER);

        fb = _Rwrited(dspf, &subf_o_buf, sizeof(subf_o_buf), 1);

        if (fb->num_bytes != sizeof(subf_o_buf))
        {
            printf("1st _Rwrited failed\n");
            exit(3);
        }

        haveFirstRec = TRUE;
    }
}
rec_cnt = 2;
while (rec_cnt <= NUM_SUBF_RECS)
{
    fb = _Rreadk(pf, &pf_buf, sizeof(pf_buf), __KEY_NEXTEQ,
                &category, sizeof(category));

    if (fb->num_bytes != sizeof(pf_buf))
    {
        printf("Subsequent _Rreadk failed \n");
        exit (2);
    }
    else
    {
        if (!(_QRY_NULL_MAP_FIELD(pf,
                                Stock_GROCERY_nmap_t,
                                QTY_ONHAND)))
        {
            memcpy(subf_o_buf.SERIAL_NUM, pf_buf.SERIAL_NUM,
                  sizeof(pf_buf.SERIAL_NUM));
            memcpy(subf_o_buf.NAME, pf_buf.NAME, sizeof(pf_buf.NAME));
            memcpy(subf_o_buf.UNIT, pf_buf.UNIT, sizeof(pf_buf.UNIT));
            IntToStr(subf_o_buf.QTY_ONHAND, pf_buf.QTY_ONHAND);
            IntToStr(subf_o_buf.QTY_ORDER, pf_buf.QTY_ORDER);
            fb = _Rwrited(dspf, &subf_o_buf, sizeof(subf_o_buf), rec_cnt);
            if (fb->num_bytes != sizeof(subf_o_buf))
            {
                printf("Subsequent _Rwrited failed\n");
                exit(3);
            }
            rec_cnt++;
        }
    }
}
/* end while loop */
}
/* end of init_subfile */

```

```

/*=====*/
/*                                     */
/* Function: IntToStr                  */
/*                                     */
/* Description: Converts an integer to a string value          */
/*                                     */
/* Input: NumStr - points to char array which stores converted int */
/*        Num    - integer to be converted                    */
/*                                     */
/* Output: NumStr - points to char array which stores converted int */
/*                                     */
/*=====*/

void IntToStr(char * NumStr, int Num)
{
    char  ReverseOut[12];
    int   NumTemp, i, j;

    NumTemp = Num;
    i = 0;
    j = 0;

    ReverseOut[0] = '\0';

    do
    {
        ReverseOut[++i] = '0' + NumTemp % 10;
        NumTemp /= 10;
    } while (NumTemp != 0);

    while (NumStr[j++] = ReverseOut[i--])
    {
        continue;
    }
}
/* end of IntToStr */

```

Chapter 8. Using AS/400 Database Files and DDM Files

Database files can contain actual data (as physical files do) or views of the data (as logical files do). DDM files are remote database files.

This section discusses:

“Introducing AS/400 Database Files and DDM Files”

“Processing AS/400 Database Files and DDM Files” on page 140

“Using Commitment Control” on page 143

“Blocking Records” on page 148

Note: There are no C++ record I/O classes.

See the *C Library Reference* for a complete description of the C library functions described in this section.

Introducing AS/400 Database Files and DDM Files

An AS/400 database file contains data that is stored permanently on the AS/400 system. The object type is *FILE. The *SQL Database Programming* contains information on AS/400 database files.

You can create and use AS/400 database files as either physical files or logical files. AS/400 database files can contain either data or source statements.

Files are accessed on remote AS/400 systems through Distributed Data Management (DDM). DDM allows programs on one AS/400 system to use files stored on a remote AS/400 system as database files. No special statements are required in your programs to support DDM files.

A DDM file is created by a user or a program on a local (or source) AS/400 system. This file (with object type *FILE) identifies a file that is kept on a remote (or target) AS/400 system. The DDM file provides the information needed for a local AS/400 to locate the remote AS/400 and to access the data in the target file. The *Distributed Data Management* contains information about DDM files.

Physical Files and Logical Files

Physical files contain the actual data that is stored on AS/400, and a description of how data is to be presented to or received from a program. They contain only one record format and one or more members.

Records in AS/400 database files can be described using either a field-level description or a record-level description.

The *field-level description* of the record includes a description of all fields and their arrangement in this record. Since the description of the fields and their arrangement is

in a database file rather than in your program, the AS/400 database files created with a field-level description are referred to as *externally described files*. See Chapter 7, “Using Externally Described Files” on page 91.

To define externally described files, you can use:

- DB2/400 database (The *SQL Database Programming* contains information about DB2/400)

- Interactive Data Definition Utility (IDDU) (The *IDDU Use* contains information about IDDU)

- Data Description Specifications (DDS) source (The *DDS Reference* contains information about DDS)

A *data description specification* is a description of a database file that is entered into AS/400 in a fixed form, and is used to create files. This description is made up of one or more record formats, which define the fields that make up the record. It may also include access path information, which determines the order in which records are retrieved from the file.

Your program can use either externally described or program-described files.

If your program uses an externally described file, the compiler can extract information from the externally described file, and automatically include field information in your program. Your program does not need to define the field information. See the description of the **#pragma mapinc** directive described in Chapter 7, “Using Externally Described Files” on page 91.

Benefits of using externally described files are:

- You can have the system perform a level check on your files at open time to ensure that the layout of the file matches the program's view of the file.

- If the record format of the file changes, a recompile of the programs that use the file may be all that is required for your program to work properly with that file.

Program-described files are AS/400 database files that are created with record-level descriptions. A *record-level description* describes only the length of the record, not the descriptions of the fields within the record. Your program must describe the fields in the record. A program-described file can also be a file that has an external description if the program does not use that description.

Logical files do not contain data. They contain a description of records found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as *multi-format* logical files.

If your program processes a logical file that contains more than one record format, you can use the `_Rformat()` function to set the format.

Some operations cannot be performed on logical files, for instance:

If you open a logical file for stream file processing with open modes `w`, `w+`, `wb`, or `wb+`, the file is opened but not cleared.

If you open a logical file for record file processing with open modes `wr` or `wr+`, the file is opened but not cleared.

Data Files and Source Files

A *data file* or physical file contains the actual data.

Records in data files are grouped into members. All the records in a file can be in one member or they can be grouped into different members. Most AS/400 database commands and operations by default assume that AS/400 database files that contain data have only one member. When your program works with AS/400 database files containing data, you do not need to specify the member name for the file unless your file contains more than one member.

Source physical files can contain one or more source members. These source members can contain source for programs or for externally described files.

Access Paths

Access paths describe the logical order of records in a file. There are two types of access paths: arrival sequence and keyed sequence.

Records retrieved using an *arrival sequence access path* are retrieved in the same order in which they are added to the file. This is similar to processing sequential files on other systems. New records are physically stored at the end of the file. An arrival sequence access path is valid for both physical and logical files.

This program copies data from the input file `T2123ASI` to the output file `T2123ASO`, using the same order in which they are added to the file `T2123ASI`. The `_Rreadn()` and `_Rwrite()` functions are used.

The sample data in the input file `T2123ASI` is:

```
joe 5
fred 6
wilma 7
```

This source uses the `_Ropen()` function to open the input file `T2123ASI` to access the records in the same order that they are added. The `_Ropen()` function also opens the output file `T2123ASO`. The `_Rread()` function reads the records in the file `T2123ASI`. The `_Rwrite()` function writes them to the file `T2123ASO`.

```

/* This program illustrates how to copy records from one file to */
/* another file, using the _Rreadn, and _Rwrite functions.*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define _RCDLEN 300

int main(void)
{
    _RFILE    *in;
    _RFILE    *out;
    char record[_RCDLEN];

    /* Open the input file for processing in arrival sequence.*/

    if ( (in = _Ropen("MYLIB/T2123ASI", "rr, arrseq=Y")) == NULL )
    {
        printf("Open failed for input file\n");
        exit(1);
    };

    /* Open the output file.*/

    if ( (out = _Ropen("MYLIB/T2123ASO", "wr")) == NULL )
    {
        printf("Open failed for output file\n");
        exit(2);
    };

    /* Copy the file until the end-of-file condition occurs.*/

    for (int i=(_Ropnfbk(in))->num_records; i>0; --i)
    {
        _Rreadn(in, record, _RCDLEN, __DFT);
        _Rwrite(out, record, _RCDLEN);
    };

    _Rclose(in);
    _Rclose(out);
}

```

The physical file T2123ASO contains the output:

```

joe 5
fred 6
wilma 7

```

Records retrieved using a *keyed sequence access path* are retrieved based on the contents of one or more key fields in the record. This is similar to processing indexed or keyed files on other systems. A keyed sequence access path is updated whenever

records are added, deleted, or updated, or when the contents of the key field are changed. A keyed sequence access path is valid for both physical and logical files.

If a logical file defines more than one record format, each record format may have different key fields. The default key for the file (if no format is specified) consists of the key fields that all record formats have in common. If there is no default key (for example, no common key field(s)), the first record in the file is always returned on an input operation that does not specify the format.

Note: When your program opens a file, the default is to process the file using the access path to create the file. If you specify `arrseq=N` (the default) on `_Ropen()`, the file is processed the way it was created. If the file was created using a keyed sequence access path, your program processes the file using a keyed sequence access path. If you specify `arrseq=Y` on `_Ropen()`, the file is processed using arrival sequence. Even though the file was created using a keyed sequence access path, your program processes the file using an arrival sequence access path.

Arranging Key Fields

Keyed sequence access paths can be ordered in ascending or descending sequence. When you describe a key field, the default sequence is ascending. If you are using Data Description Specifications (DDS) to create a keyed sequence file, the **DESCEND** DDS keyword can be used to specify that the key fields are to be arranged in descending sequence.

This program updates data in the record file T2123DD3 using the KEY field SERIALNUM. The `_Rupdate()` function is used.

The DDS source for the AS/400 database records is:

```
A          R PURCHASE
A          ITEMNAME    10
A          SERIALNUM   10
A          K SERIALNUM
```

The sample data entered into the file T2123DD3 is:

```
orange  1000222200
grape    1000222010
apple    1000222030
cherry   1000222020
```

Although you enter the data as above, the file T2123DD3 is accessed by the program T2123KSP in keyed sequence. Therefore, the program T2123KSP reads the file T2123DD3 in the following sequence:

```
grape    1000222010
cherry   1000222020
apple    1000222030
orange   1000222200
```


This program T2123KSP uses the `_Ropen()` function to open the record file T2123DD3. The default access path, which is the keyed sequence access path, is used to access the records in the file T2123DD3. The `_Rlocate()` function locks the first record. The `_Rupdate()` function updates the record locked by `_Rlocate()` to PEAR 1002022244. (The first record now becomes the second record of the original keyed sequence access path because the key has changed.) The `_Rfeod()` function forces an end-of-data condition for the member that is associated with the file specified by `in`. Any outstanding updates, deletes, or writes that the AS/400 system is buffering, is forced to nonvolatile storage.

The `_Rreadk()` function reads the record with key 1000222200.

```

/* This program illustrates how to update a record in a file using*/
/* the _Rupdate function.*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE    *in;
    _RIOFB    *fb;
    char new_purchase[21] = "PEAR    1002022244";
    char buf[512]

/* Open the file for processing in keyed sequence. File is created*/
/* with the default access path.*/

    if ( (in = _Ropen("MYLIB/T2123DD3", "rr+")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

/* Update the first record in the keyed sequence. The function*/
/* _Rlocate locks the record. */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

/* Force the end of data.*/

    _Rfeod(in);

    _Rclose(in);

/* Open the file for read */

    if ( (in = _Ropen("MYLIB/T2123DD3", "rr+")) == NULL )
    {
        printf("open for read fails\n");
    }

```

```

    exit(2);
}

/* Read the record with key 1000222200 */

fb = _Rreadk(in, buf, 21, __KEY_EQ, "1000222200", 10);
printf("record %d with value %21.21s\n", fb->rrn, buf);

/* Close the file */

_Rclose(in);
}

```

Since `grape` is the first record in the keyed sequence, it is updated, and the data file `T2123DD3` is as follows:

```

cherry    1000222020
apple     1000222030
orange    1000222200
PEAR      1002022244

```

Duplicate Key Values

When a record has key fields whose contents are the same as another record's key fields in the same file, the file has records with duplicate key values. If the record has two key fields, `num` and `date`, duplicate key values occur when the contents of both `num` and `date` are the same in two or more records.

If you want to know whether your program has processed a record that contains a duplicate key value, specify `dupkey=y` on the call to `_Ropen` that opens the file. The `dup_key` flag in the `_RIOFB_T` structure indicates whether the last record read contains a duplicate key value. (The `_Rread()` function does not update this flag.)

Notes:

1. Using the `dupkey=y` option on the call to the `_Ropen()` function may cause your I/O operations to be slower.
2. You can avoid duplicate key values by specifying the keyword `UNIQUE` in the DDS file.

Deleted Records

When an AS/400 database record is deleted, the physical record is marked as deleted but remains in the file. Deleted records may be overwritten using the `_Rwrited()` function. Deleted records may be removed from a file by using the `RGZPFM` (Reorganize Physical File Member) command, or they can be reused on write operations by specifying the parameter `REUSEDLT(*YES)` on the `CRTPF` command.

Deleted records can occur in a file if the file has been initialized with deleted records using the AS/400 Initialize Physical File Member (`INZPFM`) command. Once a record is deleted, it cannot be read.

Locking Records

The AS/400 database has built-in record integrity. AS/400 determines the lock conditions based on how your program opens the file. Table 6 shows the valid open modes and the lock states associated with them.

Open Mode	Lock State
r, rb	shared for read (*SHRRD)
a, w, ab, wb, a+, r+, w+, ab+, rb+, wb+	shared for update (*SHRUPD)

The *SQL Database Programming* contains information on locking.

You can change the lock state for a file by using the AS/400 Override Database File (OVRDBF) command or the AS/400 Allocate Object (ALCOBJ) command before you open the file. Your program can use the `system()` function to call the ALCOBJ command from an AS/400 session:

```
system( "ALCOBJ OBJ(FILEA *FILE *EXCLRD) " );
```

If a file is opened for update, the database locks any record read or positioned to , unless the `__NO_LOCK` option is specified. The locked record cannot be locked to any other open data path, whether that open data path is opened by another program or even by the same program through another file pointer.

Successfully reading another record releases a lock on a previously locked record. You can also release a lock on a record by using the `_Rr1slck()` function.

Sharing AS/400 Database Files

If your program consists only of C++ and C modules, the preferred way to share a file is by opening the file in one program and passing the file pointer to the other programs. This eliminates the need to open the file more than once.

Another way to share a file is by sharing the open data path (ODP). If you want to share a file with a program written in a language other than C or C++ this is the only way to do it. Sharing an ODP allows programs in the job to share the file's status, record position, and buffer. The `SHARE(*YES)` parameter on the AS/400 commands to create a file, change a file, and override a file, allows an Open Data Path (ODP) to be shared by two or more programs running in the same job. An *open data path* is the path through which all I/O operations for a file are performed.

You can share open data paths for streams processed one record at a time. You can also share open data paths for record files. You should not share the open data path for streams processed one character at a time, or unpredictable results may occur when you perform I/O operations.

Note: The first open of a shared file determines the open mode for the file (whether it is open for INPUT, OUTPUT, UPDATE, or DELETE). If a subsequent open specifies an open mode that was not specified by the first open, the file is

opened the second time but the open mode is ignored. If the first open specifies an open mode of `IO` and the second open specifies `IOUD`, the file is opened the second time with a mode of `IO`. `errno` is not set in this case.

Null-Capable Fields

The compiler allows you to process files with records that may contain fields that are considered to be `NULL`. You must specify `nullcap=Y` on the `_Ropen()` function. If a null-capable field is set to `NULL`, any data written in that field is considered to be meaningless. The *SQL Database Programming* contains information on `NULL` fields in AS/400 database files and DDM files.

If a file is opened with `nullcap=Y`, the AS/400 database provides input and output null maps, as well as a key null map, if the file is keyed. The input and output null maps consist of one byte for each field in the current record format of the file. The AS/400 database and your program use these null field maps to indicate which specific fields should be considered `NULL`.

The `_RFILE` structure defined in the header file `<recio.h>` contains pointers to the input, output, and key null field maps, and the lengths of these maps (`null_map_len` and `null_key_map_len`).

When you write to an AS/400 database file, you specify which fields are `NULL` with a character `'1'`. If a field is not `NULL`, you specify the character `'0'`. This is specified in the null field map pointed to by the `out_null_map` pointer. If the file does not contain any null-capable fields, but has been opened with `nullcap=Y`, your program must set each field in the null field map to the character `'0'`. This must be done before writing any data to the file and before writing each record, if the file is opened with `blkrcd=y`.

When you read from an AS/400 database file, the corresponding byte in the null field map is indicated with a character `'1'` if the field is considered `NULL`. The null field map contains `'0'` for fields that are not to be considered `NULL`. This is specified in the null field map pointed to by the `in_null_map` pointer.

The null key field map consists of one byte for each field in the key for the current record format. If you are reading a keyed database file by key that has `NULL` fields, and the key contains a field(s) that may contain a `NULL` value, you must first indicate in the null key map pointed to by `null_key_map` which fields contain `NULL`. Specify character `'1'` for any field to be considered `NULL`, and character `'0'` for the other fields.

When the `_Rupdate()` function is called to update a file that has been opened to allow null field processing, the system input buffer is used. As a result, the AS/400 database requires that an input null field map be provided through the `in_null_map` pointer. Before calling the `_Rupdate()` function, the user must clear and then set the input null field map (using the `in_null_map` pointer) according to the data that is used to update the record.

You can use the `#pragma mapinc` directive to generate typedefs that correspond to the null field maps. You can cast the null field map pointers in the `_RFILE` structures to these types to manipulate these maps. See the *C++ Language Reference* for informa-

tion on this pragma. Null field macros have also been provided in the header file `<recio.h>` to assist you in clearing and setting the null field maps in your programs.

Processing AS/400 Database Files and DDM Files

AS/400 database files and DDM files can be processed as stream files or as record files. If you open the file using the `fopen()` function, it is processed as a stream file. If you open it using the `_Ropen()` function, it is processed as a record file.

Opening as Binary Stream Files

To open an AS/400 database file as a binary stream file for record-at-a-time processing, use the `fopen()` function with a mode of `rb`, `wb`, or `ab` and the optional keyword parameters `blksize`, `lrecl`, `type`, `commit`, `arrseq`, and `ccsid`.

If you specify an AS/400 database file or a DDM file, the parameter `type` must be `record`.

Creating the physical AS/400 database files when the database file does not exist (where the open mode is `wb` or `ab`) is equivalent to specifying this AS/400 command:

```
CRTPF FILE(filename) RCDLEN(reclen)
```

Records in this file are created with a record length based on the parameter `reclen`.

Note: The only way to create a DDM file is to use the AS/400 Create DDM File (CRTDDMF) command. If you use the `fopen()` function with a mode of `wb` or `ab` and the DDM file exists on the source system, but the AS/400 database file does not exist on the remote system, a physical AS/400 database file is created on the remote system. If the DDM file does not exist on the source system, a physical AS/400 database file is created on the source system.

Opening as Record Files

To open an AS/400 database file as a record file, use the `_Ropen()` function with a mode of `rr`, `wr`, `ar`, `rr+`, `wr+`, or `ar+` and optional keyword parameters `arrseq`, `blkrcd`, `commit`, `ccsid`, `dupkey`, `nullcap`, `riofb`, `secure`, `varparm`, `vlr`, and `rtncode`.

Input and Output Considerations for Binary Stream Files

If the AS/400 database file contains deleted records, the deleted records are skipped by all binary stream I/O functions.

Binary stream record-at-a-time files cannot be processed by key. Binary stream record-at-a-time files can only be opened with the `rb`, `wb`, and `ab` modes.

Binary Stream Functions

You can use `fclose()`, `fopen()`, `fread()`, `freopen()`, and `fwrite()` to process AS/400 database files and DDM files one record at a time.

Record Functions

You can use `_Rclose()`, `_Rcommit()`, `_Rdelete()`, `_Rfeod()`, `_Rformat()` (multi-format logical files), `_Riofbk()`, `_Rlocate()`, `_Ropen()`, `_Ropnfbk()`, `_Rreadd()`, `_Rreadf()`, `_Rreadk()`, `_Rreadl()`, `_Rreadn()`, `_Rreadp()`, `_Rrlslck()`, `_Rrollbck()`, `_Rupfb()`, `_Rupdate()`, `_Rwrite()`, and `_Rwrited()` record functions to process AS/400 database files and DDM files.

This program uses various record I/O functions to show how to read and print records from a data file. The program T2123REC reads and prints records from the data file T2123DD4.

The DDS source for the database records and the sample data for the program T2123REC are the same as those used in "Access Paths" on page 133.

This program T2123REC uses the `_Ropen()` function to open the file T2123DD4. The `_Ropnfbk()` function is used to retrieve a pointer to the copy of the open feedback, which contains the library name MYLIB and the file name T2123DD4. The `_Rreadl()` function reads the fourth record, cherry 1000222020. The `_Rreadp()` function reads the third record, apple 1000222030. The `_Rrlslck()` function releases the lock on this record so that `_Rreads()` can read it again. The `_Rreadd()` function reads the second record, grape 1000222010 without locking it. The `_Rreadf()` function reads the first record, orange 1000222200. The `_Rdelete()` function deletes the second record.

```
/* This program illustrates how to use the _Rread, _Rreadp, */
/* _Rreads, _Rreadd, _Rreadf, _Rreadn, _printf, */
/* _Ropnfbk, _Rdelete, and _Rrlslck functions.*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char        buf[21];
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

/* Open the file for processing in arrival sequence.*/

    if ((fp = _Ropen ("MYLIB/T2123DD4", "rr+", arrseq=Y)) == NULL)
    {
        printf ("Open failed\n");
        exit (1);
    };

/* Get the library and file names of the file opened.*/

    opfb = _Ropnfbk (fp);
    printf ("Library: %10.10s\nFile: %10.10s\n",
           opfb->library_name,
           opfb->file_name);
}
```

```

/* Get the last record.*/

    _Rreadl (fp, NULL, 20, __DFT);
    printf ("Fourth record: %10.10s\n",*(fp->in_buf));

/* Get the third record.*/    _Rreadp (fp, NULL, 20, __DFT);
    printf ("Third record: %10.10s\n",*(fp->in_buf));

/* Release lock on the record so another function can access it.*/

    _Rrslck (fp);

/* Read the same record.*/

    _Rreads (fp, NULL, 20, __DFT);
    printf ("Same record: %10.10s\n",*(fp->in_buf));

/* Get the second record without locking it.*/

    _Rreadd (fp, NULL, 20, __NO_LOCK, 2 );
    printf ("Second record: %10.10s\n",*(fp->in_buf));

/* Get the first record.*/

    _Rreadf (fp, NULL, 20, __DFT );
    printf ("First record: %10.10s\n",*(fp->in_buf));

/* Delete the second record.*/

    _Rreadn (fp, NULL, 20, __DFT);
    _Rdelete (fp);

/* Read all records after deletion.*/

    _Rreadf (fp, NULL, 20, __DFT);
    printf ("First record after deletion: %10.10s\n",*(fp->in_buf));

    _Rreadn (fp, NULL, 20, __DFT );
    printf ("Second record after deletion: %10.10s\n",*(fp->in_buf));

    _Rreadn (fp, NULL, 20, __DFT);
    printf ("Third record after deletion: %10.10s\n",*(fp->in_buf));

    _Rclose (fp);
}

```

The output from program T2123REC is:

```
Library: MYLIB
File: T2123DD4
Fourth record: cherry
Third record: apple
Same record: apple
Second record: grape
First record: orange
First record after deletion: orange
Second record after deletion: apple
Third record after deletion: cherry
Press ENTER to end terminal session.
```

The physical file T2123DD4 contains the data shown in the output:

```
orange    1000222200
apple     1000222030
cherry    1000222020
```

Using Commitment Control

Commitment control is a means of grouping file operations as a single unit so that you can synchronize changes to database files within a job. The AS/400 database allows you to make a number of changes to any database file that is opened under commitment control. These changes are buffered by the system until your program tells the database to commit them to the files all at the same time. In C++ this is done with the `_Rcommit()` function.

If your program determines that there is something wrong with the transaction during processing, any changes already made can be rolled back, leaving the files unchanged since the last commitment or rollback. In C++ you use the `_Rrollback()` function to roll back unwanted changes.

Each time a commitment or a rollback is performed, a commitment boundary is established. The first open of a file under commitment control also establishes a commitment boundary. Any I/O operation performed since the last commitment boundary on files opened under commitment control is affected by a commitment or a rollback operation. See the *Backup and Recovery – Basic* for information on the AS/400 rollback operation.

Record locking is also affected by commitment control. The record locking rules are established by the lock level specified on the STRCMTCTL command. See the *Backup and Recovery – Advanced* for information on record locking and commitment control.

In ILE the STRCMTCTL command establishes a commitment definition. All commitment control operations are scoped to a commitment definition. A commitment definition may be scoped to an activation group or to the job. There may be many commitment definitions in use by a job at one time, but only one is used by a given activation group and there is only one job-scoped commitment definition. The *ILE Concepts* contains information on commitment control scoping.

Note: All files opened under commitment control for a given commitment definition are affected by any commitment control operation performed under that commitment definition. This includes files opened by programs that are written in languages other than C or C++.

Before you can start commitment control, you must ensure that all the AS/400 database files you want processed as one unit are in one commitment-control environment. All the files within this environment must be journaled to the same journal. Use the AS/400 commands Create Journal Receiver (CRTJRNRCV), Create Journal (CRTJRN), and Start Journal Physical File (STRJRNPf) to establish the journaling environment.

Once the journaling environment is established, enter these AS/400 commands:

```
Start Commitment Control (STRCMTCTL)
CALL program-name
End Commitment Control (ENDCMTCTL)
```

You can use commitment control to define and process several changes to AS/400 database files as a single transaction.

This program uses commitment control. Purchase orders are entered and logged in two files, T2123DD5 for daily transactions, and T2123DD6 for monthly transactions. Journal entries that reflect changes made to T2123DD5 and T2123DD6 are kept in the journal JRN.

The DDS source for daily and monthly transactions is:

```
A          R PURCHASE
A          ITEMNAME      30
A          SERIALNUM     10
```

You create a physical file NFTOBJ for notification text. Notification text is sent to the file NFTOBJ when the program that uses commitment control T2123COM is run. The DDS source for a purchase order display is:

```

A          DSPSIZ(24 80 *DS3)
A          REF(MYLIB/T2123DD5)
A          INDARA
A          CF03(03 'EXIT ORDER ENTRY')
A          R PURCHASE
A          3 32'PURCHASE ORDER FORM'
A          DSPATR(UL)
A          DSPATR(HI)
A          10 20'ITEM NAME      :'
A          DSPATR(HI)
A          12 20'SERIAL NUMBER  :'
A          DSPATR(HI)
A          ITEMNAME R      I 10 37
A          SERIALNUM R    I 12 37
A          23 34'F3 - Exit'
A          DSPATR(HI)
A          R ERROR
A          6 28'ERROR: Write failed'
A          DSPATR(BL)
A          DSPATR(UL)
A          DSPATR(HI)
A          10 26'Purchase order entry ended'

```

You create a journal receiver `JRNRCV` by using the AS/400 Create Journal Receiver (`CRTJRNRCV`) command:

```
CRTJRNRCV JRNRCV(MYLIB/JRNRCV)
```

Journal entries are placed in `JRNRCV` when the program is run.

You create a journal `JRN` and attach the journal receiver `JRNRCV` to it by using the AS/400 Create Journal (`CRTJRN`) command:

```
CRTJRN JRN(MYLIB/JRN) JRNRCV(MYLIB/JRNRCV)
```

You start journaling the changes made to `T2123DD5` and `T2123DD6` in the journal `JRN` by using the AS/400 Start Journal Physical File (`STRJRNPF`) command from an AS/400 session:

```
STRJRNPF FILE(MYLIB/T2123DD5 MYLIB/T2123DD6)JRN(MYLIB/JRN) IMAGES(*BOTH)
```

This program `T2123COM` uses the `_Ropen()` function to open the purchase display file, the daily transaction file, and the monthly transaction file. The `_Rindara()` function identifies a separate indicator area for the purchase file. The `_Rformat()` function selects the purchase record format defined as `T2123DD7`. The `_Rwrite()` function writes the purchase order display. Data that is entered updates the daily and monthly transaction files `T2123DD5` and `T2123DD6`. The transactions are committed to these AS/400 database files using the `_Rcommit()` function.

```

/*This program illustrates how to use commitment control using the*/
/* _Rcommit function and to rollback a transaction using the*/
/* _Rollbck function.*/

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

#define PF03 2
#define IND_OFF '0'
#define IND_ON '1'

int main(void)
{
    char    buf[40];
    int     rc = 1;
    _SYSindara ind_area;
    _RFILE  *purf;
    _RFILE  *dailyf;
    _RFILE  *monthlyf;

/* Open purchase display file, daily transaction file and monthly */
/* transaction file.*/

    if ((purf = _Ropen ("MYLIB/T2123DD7", "ar+,indicators=y")) == NULL)
    {
        printf ( "Display file did not open.\n" );
        exit (1);
    }

    if ((dailyf = _Ropen ("MYLIB/T2123DD5", "wr,commit=y")) ==NULL)
    {
        printf ( "Daily transaction file did not open.\n" );
        exit (2);
    }

    if ((monthlyf = _Ropen ("MYLIB/T2123DD6", "ar,commit=y")) == NULL)
    {
        printf ( "Monthly transaction file did not open.\n" );
        exit ( 3 );
    }

/* The associate separate indicator area with the purchase file.*/

    _Rindara ( purf, ind_area );

/* Select the purchase record format.*/

    _Rformat ( purf, "PURCHASE" );

```

```

/* Invite the user to enter a purchase transaction.*/
/* The _Rwrite function writes the purchase display.*/

    _Rwrite ( purf, "", 0 );
    _Readn ( purf, buf, sizeof(buf), __DFT );

/* While the user is entering transactions, update daily and*/
/* monthly transaction files.*/

while ( rc && ind_area[PF03] == IND_OFF )
{
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf)))->num_bytes);
    rc = rc && (_Rwrite (monthlyf, buf, sizeof(buf)))->num_bytes;

/* If the databases were updated, then commit transaction.*/
/* Otherwise, rollback the transaction and indicate to the*/
/* user that an error has occurred and end the program.*/

    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }
    _Rwrite ( purf, "", 0 );
    _Readn ( purf, buf, sizeof(buf), __DFT );
}
}

```

To run program T2123COM under commitment control, you use the AS/400 Start Commitment Control (STRCMTCTL) command:

```
STRCMTCTL LCKLVL(*CHG) NFYOBJ(MYLIB/NFTOBJ (*FILE )) CMTSCOPE(*JOB)
```

followed by the AS/400 CALL program command.

Note: If you want the commitment definition to be scoped to an activation group you must write a CL program that runs in that activation group and calls the STRCMTCTL command.

The program output is:

```

                PURCHASE ORDER FORM
ITEM NAME      :
SERIAL NUMBER  :
                F3 - Exit

```

Enter some sample data into the Purchase Order Form display. After an item and serial number are entered, the T2123DD5 and T2123DD6 files are updated.

The contents of the daily transaction file T2123DD5 file after three purchase-order items are entered is:

TABLE	12345
BENCH	43623
CHAIR	62513

To end commitment control, you use the AS/400 End Commitment Control (ENDCMTCTL) command from an AS/400 session:

```
ENDCMTCTL
```

The journal JRN contains entries that correspond to changes made to T2123DD5 and T2123DD6.

Blocking Records

You can use record blocking to improve the performance of I/O operations on files that are opened for input or output only. Specify the *blksize=value* parameter on a call to the `fopen()` function, or specify the *blkrcd=y* on a call to the `_Ropen()` function, to turn on record blocking. Under some circumstances, the operating system returns only one record in the block when processing a file. In these cases there is no performance gain.

You can turn off record blocking without changing your program by specifying *SEQONLY(*NO)* on the OVRDBF command.

Note: When record blocking is taking place, the I/O feedback structure is only updated when a block of records is transferred between your program and the system.

Chapter 9. Using Device Files

Device files are display files, ICF files, printer files, tape files, diskette files and save files. You can write programs that access these AS/400 device files using C record I/O functions.

This section describes:

“Introducing Display Files, ICF Files, and Printer Files”

“Introducing Tape Files, Diskette Files, and Save Files” on page 170

“Using the Device Attributes Feedback Area” on page 175

Note: There are no C++ record I/O classes.

See the *C Library Reference* for information on each of the C library functions.

Introducing Display Files, ICF Files, and Printer Files

A display file is used to define the format of the information that you wish to present on a display, and to define how that information is processed by AS/400 on its way to and from the display. A subfile is a display file that contains a group of records with the same record format that can be accessed by relative record number. The records of a subfile can be displayed at the same time at a display station. AS/400 sends the entire group of records to the display in a single operation, and receives the group from the display in another operation. The *Application Display Programming* contains information on display files and subfiles.

An Intersystem Communications Function (ICF) file defines the layout of the data sent and received between two programs on different systems, and links your program to the configuration objects that are used to communicate with a remote system. The *ICF Programming* contains information about ICF files.

A printer device file can be accessed with a program-described file or with an externally-described file. The *Printer Device Programming* contains information on printer files.

The object type for display, ICF and printer files is *FILE.

Using Indicators

Indicators allow information to be passed from a program to AS/400 or from AS/400 to a program. Display, ICF, and printer files can use indicators. Indicators are Boolean data items that may contain a value of either '1' or '0' (character). There are two types of indicators: option and response.

Option Indicators pass information from a program to AS/400; they can control which fields in a record can be displayed.

Response Indicators pass information from AS/400 to a program when an input request finishes; they can be used to inform the program which function keys the user pressed.

To use indicators, the display files, ICF files, and printer files must be defined as externally described files. The data description specification (DDS) for the externally described display file must contain a one-character `INDICATOR` field for each indicator. Indicators are either in a separate indicator area, or in the records that are read or written by the program in which case the indicators are in the file buffer.

Specifying Separate Indicator Areas

If you specify the **INDARA** keyword in the DDS, the indicators for the display files, ICF files, and printer files are specified in a separate indicator area. An *indicator area* is a 99-element character array with indices from 0-98.

If the external description of display files, ICF files, and printer files includes the **INDARA** keyword, the open of the file must specify `indicators=y`. Use the `_Rindara()` function to identify the separate indicator buffer associated with the file. To use stream files (`type=record`) with record I/O functions, cast the `FILE` pointer to an `_RFILE` pointer.

This program shows how indicators are returned in a separate indicator area.

The DDS source for the phone-book display is the same as for the indicators in a separate indicator area program. The **INDARA** keyword is specified in the DDS source.

This source uses response indicators to inform the program `T2123ID2` that a user pressed F3. The `_Rindara()` function establishes the separate indicator buffer `indicator_area` associated with the externally described file `T2123DD0`. The display file `T2123DD0` is opened with the parameter `indicators=yes` to return the indicator to a separate area.

```
/*This program uses response indicators to inform the program that*/
/*F3 was pressed by a user. The response indicator is returned to*/
/*a separate indicator area to indicate that an input request is*/
/*finished.*/

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char name[11];
    char address[20];
    char phone_num[8];
    }info;

#define IND_ON '1'
#define F3     2

int main(void)
{
    _RFILE    *fp;
```

```

_RIOFB_T   *rfb;
info       phone_list;
_SYSindara indicator_area;

if ((fp = _Ropen ("MYLIB/T2123DD0", "ar+ indicators=y")) == NULL)
{
    printf ( "display file open failed\n" );
    exit ( 1 );
}

_Rindara ( fp, indicator_area );

_Rformat ( fp, "PHONE" );
rfb = _Rwrite ( fp, "", 0 );
rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
if ( indicator_area[F3]== IND_ON )
{
    printf ( "user pressed F3\n" );
}
_Rclose ( fp );
}

```

The output is the same as in the program described in "Specifying Indicators in the File Buffer."

Specifying Indicators in the File Buffer

If you do not specify the **INDARA** keyword in the DDS, the indicators for the display file, ICF file, or printer file are specified in the record buffer. The number and order of the indicators defined in the DDS determines the number and order of the indicators in the record buffer. Indicators are always positioned first in the record buffer, followed immediately by the data. The input and output record buffers for a file are pointed to by the `in_buf` and `out_buf` pointers in the `_RFILE` structure.

This program shows how to use indicators in the record buffer. The DDS for the externally described file contains the one-character indicator field.

The DDS source for the phone-book display file T2123DD9 is:

```

A          R PHONE
A
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A I 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)

```


This source uses a response indicator to inform the program T2123ID1 that a user pressed F3.

```
/*This program uses a response indicator to inform the program */
/*that F3 was pressed by a user. The response indicator is*/
/*returned in part of the file buffer.*/

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char in03;
    char name[11];
    char address[20];
    char phone_num[8];
}info;

#define IND_ON '1'

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    info phone_list;

    if ( ( fp = _Ropen ( "MYLIB/T2123DD9", "ar+" ) ) == NULL )
    {
        printf ( "display file open failed\n" );
        exit ( 1 );
    }

    _Rformat ( fp, "PHONE" );
    rfb = _Rwrite ( fp, "", 0);
    rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
    if ( phone_list.in03 == IND_ON )
    {
        printf ( "user pressed F3\n" );
    }
    _Rclose ( fp );
}
```

The output is:

```
PHONE BOOK
Name:
Address:
Phone #:
F3 - EXIT
```

Using Major and Minor Return Codes

Major and minor return codes are used to report certain status information for display files, ICF files, and printer files. After a read operation using `_Rreadindv()`, or `_Rreadn()`, or write operation using `_Rwrite()`, the `sysparm` field in the `_RIOFB_T` structure points to the major and minor return codes for the display files, ICF files, or printer files. The header file `<recio.h>` declares the `_RIOFB_T` structure.

The *Application Display Programming* contains a list of the major and minor return codes and their meanings for display files. The *ICF Programming* contains a list of the major and minor return codes and their meanings for ICF files. The *Printer Device Programming* contains a list of the major and minor return codes and their meanings for printer files.

Your program should test the return code after each I/O operation, and define any error-handling operations based on the major and minor return codes. If the major return code is 00, the operation completed successfully. If an error occurs with a display file, ICF files or printer file, your program should handle it as it occurs.

Opening as Binary Stream Files

To open an AS/400 display file, ICF file, or printer file as a binary stream file for record-at-a-time processing use the `fopen()` function with a mode of `rb`, `wb`, and `ab`, and optional keyword parameters `type`, and `indicators`.

You can also use the `fopen()` function with a mode of `rb`, or `ab+` to open an AS/400 display file or ICF file. The optional keyword parameter `lrecl` can be used to open a display file or printer file.

Notes:

1. To create a display file, use the AS/400 CRTDSPF command. If you use the `fopen()` function and the display file does not exist, a physical AS/400 database file is created.
2. To create an ICF file, use the AS/400 CRTICFF command. If you use the `fopen()` function and the ICF file does not exist, a physical AS/400 database file is created.

Opening as Record Files

To open an AS/400 display file, ICF file, as a record file, use the `_Ropen()` function with a mode of `rr`, `wr` and `ar`, `rr+` and `wr+`, or `ar+` and optional keyword parameters `lrecl`, `indicators`, `secure`, and `riofb`. The keyword parameter `lrecl` cannot be used when opening a printer file.

To open an AS/400 printer file, as a record file, use the `_Ropen()` function with a mode of `wr` and `ar` and any of the optional keyword parameters described above.

Binary Stream Functions

To process AS/400 display files, subfiles, ICF, and printer files you can use `fclose()`, `fopen()`, `fread()` (not for printer files), `freopen()`, and `fwrite()`.

Record Functions

To process AS/400 display files, subfiles, and ICF files, you can use `_Racquire()`, `_Rclose()`, `_Rdevatr()`, `_Rfeod()`, `_Rformat()`, `_Rindara()`, `_Riofbk()`, `_Ropen()`, `_Ropnfbk()`, `_Rpgmdev()`, `_Rreadindv()`, `_Rreadn()`, `_Rrelease()`, `_Rupfb()`, `_Rwrite()`, `_Rwriterd()`, and `_Rwrread()`. In addition, you can use `_Rreadd` and `_Rupdate()` to process AS/400 subfiles.

To process AS/400 printer files you can use `_Rclose()`, `_Rfeod()`, `_Rformat()`, `_Rindara()`, `_Riofbk()`, `_Ropen()`, `_Ropnfbk()`, `_Rupfb()`, and `_Rwrite()`.

Using Display Files

Your program can process display files as program-described files or as externally described files:

For program-described display files, you must specify all formatting and control information in the program that uses the file. To create a program-described display file, specify `SRCFILE(*NONE)` on the AS/400 CRTDSPF command.

For externally described display files, you must specify all formatting and control information using DDS to describe the layout of the display. To create an externally described display file, specify the name of the member that contains the DDS source on the `SRCFILE` parameter of the AS/400 CRTDSPF command.

If you are using a user-defined data stream (UDDS), the hexadecimal 3F (X'3F') blanks the display until the next display attribute. If any CCSID conversion takes place, and a character cannot be mapped to the corresponding character in another code page, the character is mapped to hexadecimal 3F, which blanks the display until the next display attribute.

The concept of clearing a file or opening a file using append mode does not apply to display files. If you open a display file using append mode, the file is opened for input and output.

Binary Stream Files

The program device that is associated with display files is an AS/400 workstation. You establish the default device by implicitly acquiring it using the `fopen()` function.

Record Files

The program device that is associated with display files is an AS/400 workstation. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The implicitly acquired program device is determined by the `DEV` parameter on the AS/400 CRTDSPF, CHGDSPF, or OVRDSPF commands. If `*REQUESTER` is specified on the `DEV` parameter, then the program device from which the program was called is implic-

itly acquired, and becomes the default program device for I/O operations to the display file.

If `*NONE` is specified on the `DEV` parameter of the AS/400 `CRTDSPF`, `CHGDSPF`, or `OVRDSPF` commands, you must explicitly acquire the program device with the `_Racquire()` function. The explicitly acquired program device now becomes the default device for subsequent I/O operations to the device file.

To change the default program device, use the:

`_Racquire()` function to explicitly acquire another program device. The device just acquired becomes the current program device.

`_Rpgmdev()` function to change the current program device associated with a file to a previously acquired device. This program device can be used for subsequent input and output operations to the file.

`_Rreadindv()` function to read from a program device. The actual program device read becomes the default device.

To release a program device, use the `_Rrelease()` function. (The program device must have been previously acquired.) This detaches the device from an open file; I/O operations can no longer be performed for this device. To use the device after releasing it, you must acquire it again.

All program devices are implicitly released when you close the file. If the device file has a shared open data path, the last close operation releases the program device.

Establish a Default Program Device

This program shows how to explicitly establish a default program device for a display file using the `_Racquire()` function.

Note: To run this program, you must use a display device defined on your AS/400 in place of `DEVICE2`.

The DDS source for a display file is:

```
A                                     DSPSIZ(24 80 *DS3)
A      R EXAMPLE
A      OUTPUT          5A  O  5 20
A      INPUT           20A  I  7 20
A                                     5 10 'OUTPUT: '
A                                     7 10 'INPUT: '
```

To override the file `STDOUT` with the printer file `QPRINT` so that any output from the `printf()` function does not interfere with the output from the display file, use the AS/400 command:

```
OVERPRTF FILE(STDOUT) TOFILE(QPRINT)
```

This source uses the `_Racquire()` function to explicitly acquire the program device `DEVICE2`, which is the current program device. The `_Rformat()` function selects the

record format EXAMPLE. The `_Rwrite()` function writes data to the default device. The `_Rreadn()` function reads the string from the current program device DEVICE2.

```
/*This program establishes a default device using the _Racquire*/  
/*function.*/
```

```
#include <stdio.h>  
#include <recio.h>  
#include <signal.h>  
#include <stdlib.h>
```

```
void handler ( int );
```

```
int main(void)  
{
```

```
    _RFILE *fp;  
    _RIOFB_T *rfb;  
    char buf[21];
```

```
    signal (SIGALL, handler );
```

```
    if (( fp = _Ropen ( "MYLIB/T2123DDD", "ar+" )) == NULL )  
    {  
        printf ( "Could not open the display file\n" );  
        exit ( 1 );  
    }
```

```
    _Racquire ( fp, "DEVICE2" );          /* Acquire the device.*/  
                                          /* DEVICE2 is now the */  
                                          /* default program device.*/
```

```
    _Rformat ( fp, "EXAMPLE" );          /* Select the record */  
                                          /* format.*/
```

```
    _Rwrite ( fp, "Hello", 5 );          /* Write to the default*/  
                                          /* program device.*/
```

```
    rfb = _Rreadn (fp, buf, 20, __DFT); /* Read from the default*/  
                                          /* program device.*/
```

```
    buf[rfb -> num_bytes]= '\0';
```

```
    printf ( "Response from device : %s\n", buf );
```

```
    _Rrelease ( fp, "DEVICE2" );  
    _Rclose ( fp );
```

```
}
```

```
void handler ( int sig )  
{
```

```

printf ( "message = %7.7s\n", _EXCP_MSGID );
printf ( "program continues \n" );
signal ( SIGALL, handler );
}

```

The output is:

```

OUTPUT: Hello
INPUT: _____

```

If you type GOOD MORNING on the input line and press Enter, the file QPRINT contains:

```
Response from device : GOOD MORNING
```

Change the Default Program Device

This program shows how to change the default program device using the `_Rpgmdev()` function.

Note: To run this program, you must use two display devices defined on your AS/400 in place of `DEVICE1` and `DEVICE2`.

The display file T2123DDE DDS source is:

```

A                                DSPSIZ(24 80 *DS3)
A                                INVITE
A      R FORMAT1
A                                9 13'Name:'
A      NAME                      20A I 9 20
A                                11 10'Address:'
A      ADDRESS                    25A I 11 20
A      R FORMAT2
A                                9 13'Name:'
A      NAME                      8A I 9 20
A                                11 10'Password:'
A      PASSWORD                  10A I 11 20

```

To override the file `STDOUT` with the printer file `QPRINT`, use this AS/400 command:

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

This source uses the `_Racquire()` function to explicitly acquire another device named `DEVICE1`, which becomes the current program device. The `_Rpgmdev()` function changes the current device named `DEVICE1` to `DEVICE2`. The `_Rreadindv()` function reads records from `DEVICE1`. The `_Release()` function releases `DEVICE1` and `DEVICE2`.

```

/*This program illustrates how to change a default program device.*/
/*using the _Racquire, _Rpgmdev, _Rrelease and _Rreadindv*/
/*functions.*/

```

```

#include <stdio.h>
#include <recio.h>

```

```

#include <string.h>
#include <stdlib.h>

typedef struct{
    char name[20];
    char address[25];
}format1 ;

typedef struct{
    char name[8];
    char password[10];
}format2 ;

typedef union{
    format1 fmt1;
    format2 fmt2;
}formats ;

void io_error_check( _RIOFB_T *rfb )
{
if ( memcmp(rfb->sysparm->_Maj_Min.major_rc,"00",2 ) ||
    memcmp ( rfb->sysparm->_Maj_Min.minor_rc,"00",2 ))
    {
        printf ( "I/O error occurred, program ends.\n" );
        exit ( 1 );
    }
}

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    _XXIOFB_T *iofb;
    int size;
    formats buf;

/* Open the device file.*/

if (( fp = _Ropen ( "MYLIB/T2123DDE", "ar+" )) == NULL )
{
    printf ( "Could not open file\n" );
    exit ( 1 );
}

_Racquire ( fp,"DEVICE1" ); /* Acquire another device.*/
/* Replace with the actual*/
/* device name.*/

_Rformat ( fp,"FORMAT1" ); /* Set the record format for the*/
/* display file.*/

rfb = _Rwrite ( fp, "", 0 );/* Set up the display.*/

```

```

io_error_check(rfb);

_Rpgmdev ( fp,"DEVICE2" ); /* Change the default program */
/* device. Replace it with the */
/* actual device name.*/
/* Device2 implicitly acquired at */
/* open.*/

_Rformat ( fp,"FORMAT2" ); /* Set the record format for */
/* the display file. */

rfb = _Rwrite ( fp, "", 0 ); /* Set up the display.*/

io_error_check ( rfb );

_Rreadindv ( fp, &buf, sizeof(buf), __DFT );
/* Read from the first device that */
/* enters data. The device becomes */
/* the default program device.*/

io_error_check ( rfb );

/* Determine which terminal responded first.*/

iofb = _Riofbk ( fp );
if ( !strcmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
{
    _Rrelease ( fp, "DEVICE1" );
}
else
{
    _Rrelease(fp, "DEVICE2" );
}
}

```

The output is:

<pre> Name: Password: </pre>

When the program is run, a different display appears on each device. Data may be entered on both displays, but the data that is first entered is returned to the program. The output from the program is in QPRINT. If the name SMITH and the address 10 MAIN ST is entered on DEVICE1 before any data is entered on DEVICE2, then the file QPRINT contains:

```
Data displayed on DEVICE1 is SMITH 10 MAIN ST
```


Using Subfiles

To use a subfile, you must initialize it before displaying it. One way to initialize the subfile is to read records from an AS/400 database file and write them to a subfile. You must write them using the `_Rwrited()` function.

Only message subfiles are supported for binary stream subfiles.

Record Subfiles

I/O operations performed on the subfile record format do not cause data to appear on the display. You must read or write the subfile-control record format to transfer data to or from the display. Use the `_Rformat()` function to distinguish between subfile record formats and subfile-control record formats. If the format you specify with the `_Rformat()` function refers to a subfile record format, no data is transferred to or from the display.

To read the next changed subfile record, use the `_Rreadnc()` function. This function searches for the next changed record from the current position in the file. If this is the first read operation, the first changed record in the subfile is read. If the end-of-file (EOF) is reached before finding a changed record, EOF is returned in the `num_bytes` field of the `_RIOFB_T` structure.

This program uses DDS from T2123DDG and T2123DDH to display a list of names and phone numbers.

The DDS source for a subfile T2123DDG is:

```
A                                DSPSIZ(24 80 *DS3)
A      R SFL                      SFL
A      NAME                      10A B 10 25
A      PHONE                     10A B  +5
A      R SFLCTL                   SFLCTL(SFL)
A                                SFLPAG(5)
A                                SFLSIZ(26)
A                                SFLDSP
A                                SFLDSPCTL
A                                22 25 '<PAGE DOWN> FOR NEXT PAGE '
A                                23 25 '<PAGE UP> FOR PREVIOUS PAGE '
```

The DDS source for the file T2123DDH containing the names and phone numbers is:

```
R ENTRY
  NAME          10A
  PHONE        10A
```

Some sample data is:

```
David      555-5634
Florence   555-4537
Irene     555-5235
Carrie    555-5347
Michele   555-4557
```

This source T2123SUB, uses the `_Ropen()` function to open the subfile T2123DDG and the physical file T2123DDH. The subfile is then initialized with records from the physical file. The subfile records are written to the display using the `_Rwrited()` function.

```

/* This program illustrates how to use subfiles.*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define LEN      10
#define NUM_RECS 20
#define SUBFILENAME "MYLIB/T2123DDG"
#define PFILENAME  "MYLIB/T2123DDH"

typedef struct{
    char name[LEN];
    char phone[LEN];
}pf_t;

#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE      *pf;
    _RFILE      *subf;

    /* Open the subfile and the physical file.*/

    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)
    {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }

    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)
    {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }

    /* Initialize the subfile with records from the physical file. */

    init_subfile(pf, subf);

    /* Write the subfile to the display by writing a record to the */
    /* subfile control format.*/

    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadn(subf, "", 0, __DFT);

```

```

/* Close the physical file and the subfile.*/
_Rclose(pf);

    _Rclose(subf);
}

void init_subfile(_RFILE *pf, _RFILE *subf)
{
    int          i;
    pf_t         record;

/* Select the subfile record format.*/

    _Rformat(subf, "SFL");

    for (int i=_Ropnfbk(pf)->num_records; i>0; --i)
        {
            _Rreadn(pf, &record, RECLEN, __DFT);
            _Rwrited(subf, &record, RECLEN, i);
            printf("error occurred during write\n");
            exit(3);
        }
}

```

The output is:

```

David      555-5634
Florence   555-4537

Irene      555-5235

Carrie     555-5347

Michele    555-4557

<PAGE DOWN> FOR NEXT PAGE
<PAGE UP> FOR PREVIOUS PAGE

```

Using Intersystem Communication Function (ICF) Files

A program can process ICF files as program-described files or as externally described files. The AS/400 file `QSYS/QICDMF` contains a system-supplied record format.

The concept of clearing or opening a file using append mode does not apply to ICF files. If you open an ICF file using append mode (`ar+` or `ab+`), the file is opened for input and output.

If you want to use records with variable lengths, you must use the keyword **VARLEN** in the DDS.

This program retrieves a user ID and password from a source program, and sends it to a target program. The target program checks the user ID and password for errors, and sends a response to the source program. To run this program you must have all of the following:

Notes:

1. A target program T2123TGT on a remote system
2. An active communications line between the source and target systems using the program T2123ICF
3. An active program T2123TGT on the target system
4. A communications protocol called Advanced Program-to-Program Communications (APPC)

The DDS source for a password and user ID database file is:

```
A                               UNIQUE
A      R PASSWRDF
A      USERID      8A
A      PASSWRD     10A
A      K USERID
```

To create the ICF file T2123DDB using this DDS source, use this AS/400 command:

```
CRTICFF FILE(MYLIB/T2123DDB) SRCFILE(QCLE/QADSSRC)
ACQPGMDEV(CAPPC2)
```

```
A      R SNDPASS
A      FLD1      18A
A      R CHKPASS
A      FLD1      1A
A      R EVOKPGM
A                               EVOKE(MYLIB/T2123TGT)
A                               SECURITY(2 'PASSWRD' +
A                               3 'USRID')
```

To create the ICF file T2123DDC using this DDS source, use this AS/400 command:

```
CRTICFF FILE(MYLIB/T2123DDC) SRCFILE(QCLE/QADSSRC)
ACQPGMDEV(CAPPC1)
```

```
A      R RCVPASS
A      UID      8A
A      PWD      10A
A      R VRYPASS
A      CHKPASS  1A
```

To add a program device entry for ICF file T2123DDB, use this AS/400 command:

```
ADDICFDEVE FILE(MYLIB/T2123DDB) PGMDEV(CAPPC2)
RMTLOCNAME(CAPPC1) MODE(CAPPCMOD)
```

To add a program device entry for ICF file T2123DDC, use this AS/400 command:

```
ADDICFDEVE FILE(MYLIB/T2123DDC) PGMDEV(CAPPC1)
RMTLOCNAME(*REQUESTER) MODE(CAPPCMOD)
```

Source Program

This source uses the `_Ropen()` function to open the record file `T2123DDB`. `_Rformat()` specifies the `EVOKPGM` as the current record format. The `EVOKE` statement in `T2123DD8` calls the target program `T2123TGT`. `_Rformat()` accesses the record format `SNDPASS` in the file `T2123DDB`. The user ID and password are sent to the target program `T2123TGT`. `_Rformat()` accesses the record format `CHKPASS` in the file `T2123DDB`. The received password and user ID are then verified.

```
/* This program sends a user ID and password to a target program */
/* on another system. The target program returns the user ID and*/
/* password. This program verifies the returned values. */
```

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define ID_SIZE      8
#define PASSWD_SIZE 10
#define RCD_SIZE    ID_SIZE + PASSWD_SIZE
#define ERROR       '2'
#define VALID       '1'
```

```
_RFILE *fp;
```

```
void ioCheck(char *majorRc)
{
    if ( memcmp(majorRc, "00", 2) != 0 )
    {
        printf("Fatal I/O error occurred, program ends\n");
        _Rclose(fp);
        exit(1);
    }
}
```

```
int main(void)
{
    _RIOFB_T *fb;
    char    idPass[RCD_SIZE];
    char    buf[RCD_SIZE + 1];
    char    passwordCheck=ERROR;
```

```
/* Open the source file T2123DDB. */
```

```
if ( (fp = _Ropen("MYLIB/T2123DDB", "ar+")) == NULL )
{
    printf("Could not open SOURCE ICF file\n");
    exit(2);
}
```

```

/* Start the target program T2123TGT. */

    _Rformat(fp, "EVOKPGM");
    fb = _Rwrite(fp, "", 0);
    ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Get the user ID and password. */

    memset(idPass, ' ', RCD_SIZE);
    printf("Enter user ID (maximum 8 characters):\n");
    scanf("%.8s", buf);
    memcpy(idPass, buf, strlen(buf));
    printf("Enter password (maximum 10 characters):\n");
    scanf("%.10s", buf);
    memcpy(idPass + ID_SIZE, buf, strlen(buf));

/* Send data to the TARGET program T2123TGT.*/

    _Rformat(fp, "SNDPASS");
    fb = _Rwrite(fp, idPass, RCD_SIZE);
    ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Receive data from TARGET program T2123TGT.*/
    _Rformat(fp, "CHKPASS");
    fb = _Rreadn(fp, &passwordCheck, 1, __DFT);
    ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* If a problem, such as a communications line is down, occurs */
/* in the TARGET program, then end the program. */
/* Otherwise, print the password verification.*/

    if ( passwordCheck == ERROR )
    {
        _Rclose(fp);
        exit(3);
    }
    else if ( passwordCheck == VALID )
    {
        printf("Password valid\n");
    }
    else
    {
        printf("Password invalid\n");
    }
    _Rclose(fp);
}

```

Target Program

This program T2123TGT uses the `_Ropen()` to open the file T2123DDC. `_Ropen()` opens the password file T2123DDA. `_Rformat()` accesses the record format RCVPASS in the file

```

T2123DDC. _Rreadn() reads the password and user ID from the source program
T2123ICF. Errors are checked, and a response is sent to the source program T2123ICF.

/* This program checks the user ID and password.*/

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

#define ID_SIZE      8
#define PASSWD_SIZE 10
#define RCD_SIZE     ID_SIZE + PASSWD_SIZE
#define ERROR        '2'
#define VALID        '1'
#define INVALID      '0'

int main(void)
{
    _RFILE    *icff;
    _RFILE    *pswd;
    _RIOFB_T  *fb;
    char      rcv[RCD_SIZE];
    char      pwr[RCD_SIZE];
    char      vry;

/* Open the TARGET file T1529DDC.*/

    if ( (icff = _Ropen("MYLIB/T2123DDC", "ar+")) == NULL )
    {
        printf("Could not open TARGET icf file T2123DDC\n");
        exit(1);
    }

/* Open the PASSWORD file T2123DDA.*/

    if ( (pswd = _Ropen("MYLIB/T2123DDA", "rr")) == NULL )
    {
        printf("Could not open PASSWORD file T2123DDA\n");
        exit(2);
    }

/* Read the information from the SOURCE program T2123ICF. */

    _Rformat(icff, "RCVPASS");
    fb = _Rreadn(icff, &rcv, RCD_SIZE, __DFT);

/* Check for errors and send response to SOURCE program.*/
    if ( memcmp(fb->sysparm->_Maj_Min.major_rc, "00", 2) != 0 )
    {
        vry = ERROR;
    }
}

```

```

else
{
fb = _Rreadk(pswd, &pwr, RCD_SIZE, __DFT, &rcv, ID_SIZE);

if ( fb->num_bytes == RCD_SIZE &&
    memcmp(pwr + ID_SIZE, rcv + ID_SIZE, PASSWD_SIZE) == 0 )
{
    vry = VALID;
}
else
{
    vry = INVALID;
}
}
_Rformat(icff, "VRYPASS");
_Rwrite(icff, &vry, 1);

_Rclose(icff);
_Rclose(pswd);
}

```

The output is:

```

Password valid
Press ENTER to end terminal session.

```

After calling the program, you may enter a user ID and password. If the password is correct, "Password valid" appears on the display; if it is incorrect, "Password invalid" appears.

Binary Stream Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `fopen()` function.

The `fwrite()` function returns the number of elements successfully written. When you use `PDATA`, the value returned by the `fwrite()` function does not take `PDATA` into consideration. When you use `PDATA`, `errno` is set to `ETRUNC`, even though all the data is successfully written.

Record Files

The `_Rwrite()` function returns the number of characters successfully transferred across a communications line. When you use `PDATA`, the value returned by the `_Rwrite()` function (`num_bytes`) includes `PDATA`, unlike the `_fwrite()` function.

Program Devices for Record ICF Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The

implicitly acquired program device is determined by the *ACQPGMDEV* parameter on the AS/400 CRTICFF, OVRICFF, or CHGICFF commands. If the program device name is specified on the *ACQPGMDEV* parameter, the program device must be defined to the device file before it is opened. It is defined by specifying the name on the *PGMDEV* parameter of the AS/400 ADDICFDEVE or OVRICFDEVE commands.

If **NONE* is specified for the *ACQPGMDEV* parameter of the AS/400 CRTICFF, OVRICFF, or CHGICFF commands, you must explicitly acquire the program device using the `_Racquire()` function.

To change the default program device see “Record Files” on page 154

Using Printer Files

Your program can process printer files as program-described files or as externally described files.

The concept of clearing a file or opening a file using append mode does not apply to printer files.

Program-described files allow first character forms control (FCFC). To use this, include the FCFC code in the first position of each data record in the printer file. The *Printer Device Programming* contains information about FCFC codes. You must use a printer stream file and the `fwrite()` function to use FCFC.

This program uses first character forms control (FCFC) in a program-described printer file. Employees' names and serial numbers are read from a physical file and written to the printer file.

To create the printer file T2123FCP, use this AS/400 command:

```
CRTPRTF FILE(MYLIB/T2123FCP) CTLCHAR(*FCFC)
CHLVAL((1 (13)))
```

Create a physical file T2123FCI for the program. Sample data for the program is:

```
Jim Roberts          1234567890
Karen Smith          2314563567
John Doe              5646357324
```

This source uses the `fopen()` function to open the printer stream file T2123FCP using record-at-a-time processing. The `fopen()` function also opens the physical file T2123FCI for record-at-a-time processing. The `strncpy()` function copies the records into the print buffer. The `fwrite()` function prints out the employee records.

```
/* This program illustrates how to use a printer stream file, */
/* the fwrite function and first character forms control. */
```

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

```
#define BUF_SIZE 53
#define BUF_OFFSET 20
```

```

int main(void)
{
    FILE      *dbf;
    FILE      *prtf;
    char buf   [BUF_SIZE];
    char tmpbuf [BUF_SIZE];

    /* Open the printer file using the first character forms control. */
    /* recfm and lrecl are required.*/

    prtf = fopen ("MYLIB/T2123FCP", "wb type=record recfm=fa lrecl=53" );

    dbf = fopen ("MYLIB/T2123FCI", "rb type=record blksize=0" );

    /* Print out the header information. */

    memset ( buf, ' ', BUF_SIZE );

    /* Use channel value 1.*/

    strncpy ( buf, "1                EMPLOYEE INFORMATION",47);

    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use single spacing.*/

    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use triple spacing. */

    strncpy ( buf,"-                NAME                SERIAL NUMBER"
                ,BUF_SIZE );
    fwrite ( buf, 1, BUF_SIZE, prtf );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Print out the employee information.*/

    while ( fread ( tmpbuf, 1, BUF_SIZE, dbf ))
    {
        memset ( buf, ' ', BUF_SIZE );

    /* Use double spacing. */

        buf[0]= '0';
        strncpy ( buf + BUF_OFFSET, tmpbuf, strlen(tmpbuf) );
        fwrite ( buf, 1, BUF_SIZE, prtf );
    }

    fclose ( prtf );

```

```

    fclose ( dbf );
}

```

The output file and the printed output file contain:

EMPLOYEE INFORMATION	
NAME	SERIAL NUMBER
Jim Roberts	1234567890
Karen Smith	2314563567
John Doe	5646357324

Introducing Tape Files, Diskette Files, and Save Files

You can write records to and read records from a tape file, a diskette file, or save file. A tape file is a device file used for tape units. A save file is a file allocated in auxiliary storage to store saved data on disk (without requiring diskettes or tapes), or to receive objects sent through the network. A diskette file is a device file used for a diskette unit. The object type for tape files, diskette files, and save files is `*FILE`. The *Tape and Diskette Device Programming* contains information on tape files, and diskette files. The *Backup and Recovery – Basic* contains information on save files.

Opening as Binary Stream Files

To open an AS/400 tape file, diskette file, or save file as a binary stream file for record-at-a-time processing use the `fopen()` function with a mode of `rb`, or `wb`, and optional keyword parameters `type`, and `lrecl`, `recfm`.

To open an AS/400 tape file, or save file as a binary stream file for record-at-a-time processing you can also use the `fopen()` function with a mode of `ab`. You can use the optional keyword parameter `blksize` when you open an AS/400 tape file or diskette file. You can use the optional keyword parameter `recfm` when you open an AS/400 tape file.

Notes:

1. To create a tape file, use the AS/400 CRTTAPF command. If you use the `fopen()` function and the tape file does not exist, a physical AS/400 database file is created.
2. To create a diskette, file use the AS/400 CRTDKTF command. If you use the `fopen()` function and the diskette file does not exist, a physical AS/400 database file is created.
3. To create a save file, use the AS/400 CRTSAVF command. If you use the `fopen()` function with a mode of `wb` or `ab` and the save file does not exist, a physical AS/400 database file is created.

Opening as Record Files

To open an AS/400 tape file, diskette file or save file as a record file, use the `_Ropen()` function with a mode of `rr`, `wr`, and optional keyword parameters `lrecl`, `secure`, and `riofb`. In addition, the optional keyword parameter `blksize` can be used to open an AS/400 tape file or diskette file. As well, the mode `ab` can be used when you open an AS/400 tape file or save file.

Using Tape, Diskette and Save Files

A program can only process tape, diskette, or save files sequentially.

A diskette, or tape device, or save file, can only be accessed with a program-described file.

All records read or written to save files must be 528 characters in length. Any records written to another save file cannot be changed by the program.

The concept of clearing a file or opening a file using append mode does not apply to diskette files. If you open a tape file, diskette file or save file using append mode, the file is opened for input and output.

The diskette file-label name is required when the file is opened. You specify this label name using the AS/400 Override Diskette File (OVRDKTF) command.

These considerations apply to diskette files opened for input:

If the *lrecl* parameter is not specified or is specified as zero, the record length in the data file label on the diskette name is used to determine the length of the records to read.

If the *lrecl* parameter is greater than the length of the records on the diskette file, the records that are read are padded with blanks.

If the *lrecl* parameter is less than the length of the records on the diskette file, the records that are read are truncated.

If the file type in the diskette file is a source file, a date and sequence number (of 12 bytes) is appended at the beginning of each record. You must consider these extra bytes when doing I/O operations on diskette source files, and add bytes to the *lrecl* parameter on the open statement.

Note: Output may not always result in an I/O operation to a diskette file. The I/O buffer must contain enough data to fill an entire track on a diskette.

Writing to a Tape File

This program shows how to write to a tape file. To create the tape file T2123TPF, use this AS/400 command:

```
CRTTAPF FILE(MYLIB/T2123TPF) DEV(TAP01) SEQNBR(*END)
LABEL(CSOURCE) FILETYPE(*SRC)
```

Assume the source file CSOURCE contains some source statements.

This program opens the source physical file T2123TPF. The file QCSRC contains the member CSOURCE with the source statements. `_Ropen()` opens the tape file T2123TPF to receive the source statements. `_Rreadn()` reads the source statements, finds their sizes, and adds them to the tape file T2123TPF.

```

/* This program illustrates how to write to a tape file.*/

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char buf [92];
    int i;

/* Open the source physical file containing the C source. */

    if ((fp = _Ropen ("MYLIB/QCSRC(CSOURCE)", "rr blkrcd=y")) == NULL)
    {
        printf ( "could not open C source file\n" );
        exit ( 1 );
    }

/* Open the tape file to receive the C source statements*/

    if ((tape = _Ropen ("MYLIB/T2123TPF", "wr lrecl=92 blkrcd=y")) == NULL)
    {
        printf ( "could not open tape file\n" );
        exit ( 2 );
    }

/* Read the C source statements, find their sizes */
/* and add them to the tape file. */

    for (int i=_Ropnfbk(fp)->num_records; i>0; --i)
    {
        _Rreadn(fp, buf, sizeof(buf), __DFT);
        for ( i = sizeof(buf) - 1 ;buf[i]== ' ' && i > 12; --i);
        i = ( i == 12 ) ? 80 : ( i - 12 );
        memmove ( buf, buf+12, i );
        _Rwrite ( tape, buf, i );
    }
    _Rclose ( fp );
    _Rclose ( tape );
}

```

After you run the program, the tape file contains the source statements from CSOURCE.

Writing to a Diskette File

This program shows how to write to a diskette file. To create the diskette file T2123DKF, use this AS/400 command:

```

CRTDKTF FILE(MYLIB/T2123DKF) DEV(DKT02) LABEL(FILE1)
EXCHTYPE(*I) SPOOL(*NO)

```

The DDS source is:

```
A          R CUST
A          NAME          20A
A          AGE           3B
A          DENTAL        6B
```

The sample data is:

```
Dave Bolt      35 350
Mary Smith     54 444
Mike Tomas     25 545
Alex Michaels  32 512
```

To select only records that have a value greater than 400 in the DENTAL field, use this AS/400 command:

```
OPNQRYF FILE(MYLIB/T2123DDI) QRYSLT('DENTAL *GT 400')
```

This program uses `_Ropen()` to open the diskette file T2123DKF and the database file T2123DDI. `_Rreadn()` reads all the AS/400 database records from T2123DDI. `_Rwrite()` copies all the AS/400 database records that have a value > 400 in the DENTAL field to the diskette file T2123DKF.

```
/* This program illustrates how to write to a diskette file.*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define BUF_SIZE 30

int main(void)
{
    _RFILE *dktf;
    _RFILE *dbf;
    char buf [BUF_SIZE];

    /* Open the diskette file. */

    if ((dktf = _Ropen ("MYLIB/T2123DKF", "wr blkrcd=y lrecl=100")) == NULL)
    {
        printf ( "DISKETTE file did not open \n" );
        exit ( 1 );
    }

    /* Open the database file. */

    if ( ( dbf = _Ropen ( "MYLIB/T2123DDI", "rr" ) ) == NULL)
    {
        printf ( "DATABASE file did not open\n" );
        exit ( 2 );
    }
}
```

```

/* Copy all the database records meeting the OPNQRYP selection */
/* criteria to the diskette file. */

for (int i=_Ropnfbk(dbf)->num_records; i>0; --i)
{
    _Rreadn ( dbf, buf, BUF_SIZE, __DFT);
    _Rwrite ( dktf, buf, BUF_SIZE );
}

_Rclose ( dktf );
_Rclose ( dbf );
}

```

The output to the diskette file is:

```

Mary Smith      444
Mike Tomas      454
Alex Michaels   512

```

After you run the program, the diskette file contains only the records that satisfied the selection criteria.

Blocking Binary Stream Files

If your program processes tape files, performance can be improved if records are blocked.

Note: The value you specify on the *blksize* parameter for the `fopen()` function overrides the one you specified on the AS/400 CRTTAPF or CHGTAPF commands. You can still override the *BLKLEN* parameter with the AS/400 OVRTAPF command.

If you specify 0 on either *BLKLEN* or *blksize*, AS/400 calculates a block size for you. You can specify a value on either parameter of between 0 and 32 767 characters.

Using Record Files

You can use the `_Rfeod()` function to open tape record files, diskette record files and save record files for input operations. As well you can use the `_Rfeov()` function to open tape record files for input and output operations.

Using `_Rfeod`

If a tape, diskette, or save file is opened for input, the `_Rfeod()` function returns an `end-of-file` to your program. For tape files, the `_Rfeod()` function also positions the tape at the last volume in the file.

If a tape file is opened for output, the `_Rfeod()` function forces all unbuffered data to be written to the tape.

If a diskette file is opened for output, the `_Rfeod()` function does not write any data.

If a save file is opened for output, the `_Rfeod()` function ensures that any data written to the file is forced to auxiliary storage. If you want to continue reading from or writing to the save file after calling this function, you must close the file and open it again.

Using `_Rfeov`

The `_Rfeov()` function is valid for tape record files opened for input and output operations. For input operations, it signals the `end-of-file` and positions the tape at the next volume. For output operations, any unwritten data is forced to the tape. An `end-of-volume` trailer is written to the tape, which means that no data can be written after this trailer. Any write operations that take place after `_Rfeov()` occur on a new volume.

Blocking Record Files

If your program processes tape or diskette files, performance can be improved if records are blocked. If you specify `blkrcd=Y` on the `_Ropen()` function, AS/400 calculates the number of records to be transferred as a block to your program.

Reading From and Writing to Diskette Files

If you read from a diskette file, the next sequential record in the diskette file is processed. Use the `_Rreadn()` function for reading from diskette files, and the `_Rwrite()` function for writing to diskette files.

Binary Stream Functions

To process the AS/400 tape files, diskette files, or save files you can use `fclose()`, `fopen()`, `fread()`, `freopen()`, and `fwrite()`.

Record Functions

To process the AS/400 tape files, diskette files, or save files you can use `_Rclose()`, `_Rfeod()`, `_Riofbk()`, `_Ropen()`, `_Rreadn()`, `_Rupfbk()`, and `_Rwrite()`. You can also use the `_Ropfbk()` function to process AS/400 tape files and diskette files.

To process the AS/400 tape files you can also use `_Rfeov()`.

To process the AS/400 save files you can also use `_Ropnfbk()`.

Using the Device Attributes Feedback Area

The device attributes feedback area is part of the open data path that contains information about the device attributes associated with that open-data path. You can assign a pointer to a copy of the information by using the `_Rdevatr()` function. The *Data Management* contains information on the fields contained in the device attributes feedback area. The structure that maps to the device attributes feedback area can be found in the `<xxfdbk.h>` header file.

Chapter 10. Using AS/400 Pointers

In ILE C++ you can declare an ANSI C++ pointer such as a data pointer and you can declare an AS/400 pointer such as a system pointer. AS/400 pointers are used extensively in the ILE C Machine Interface Library and in the ILE C exception-handling structures and functions.

This section describes:

“Introducing AS/400 Pointer Types”

“Understanding the Rules for Using Open Pointers” on page 178

“Understanding Other Pointer Rules” on page 178

“Declaring Pointer Variables” on page 179

“Using Pointer Casting” on page 183

Introducing AS/400 Pointer Types

In ANSI C++ a `pointer` type is a type that is derived from a `function` type, a `data object` type, or an `incomplete` type. AS/400 pointer types can also be derived from other AS/400 entities such as system objects (programs), code labels, and process objects. These pointer types are referred to as AS/400 pointers:

<code>Open</code>	Can hold any of the other pointer types
<code>Space</code>	Generic pointers to data objects
<code>Function</code>	System pointers to program objects or procedure pointers to bound ILE procedures
<code>System</code>	Pointers to system objects such as queues, indexes, libraries, and program objects
<code>Label</code>	Pointers to fixed locations within the executable code of a procedure or function
<code>Invocation</code>	Pointers to process objects for procedure (function) calls under ILE, or program calls under EPM or OPM
<code>Suspend</code>	Pointers to the location in a procedure where control is suspended

These pointer types, as well as pointers to data objects and incomplete types, are not data-type-compatible (assignment or comparison) with each other. A variable declared as a pointer to a data object cannot be assigned the value of a `function` pointer or `system` pointer, and a `system` pointer cannot be compared for equality with an `invocation` pointer or pointer to a data object. The above is not true for `open` pointers.

Notes:

1. Label pointers are only used by the `setjmp` macro and `longjmp()` function.
2. An `open` pointer is a “pseudo” pointer type. It may contain any other pointer type but it is not a pointer type in itself.

Understanding the Rules for Using Open Pointers

Before you use `open` pointers in your program, consider these rules:

An `open` pointer and a `void` pointer are the same on AS/400.

An `open (void)` pointer can hold any type of pointer. It is data-type-compatible with all pointer types on AS/400 with an explicit cast. No compile-time error occurs when you assign an `open` pointer to other pointer types with a cast nor when you assign other pointer types to an `open` pointer. You may receive a run-time exception if the type of pointer held in the `open` pointer is not data-type-compatible with the target of the assignment. You receive a run-time exception if you perform arithmetic operations on `system` pointers.

Note: C++ allows `void` pointers to be assigned only to other `void` pointers. A `void` pointer may be assigned to a `non-void` pointer using an explicit cast.

An `open` pointer can be compared for equality (`==`, `!=`) to any pointer type.

An `open` pointer can be compared in a relational operation (`<`, `>`, `<=`, `>=`) to another `open` pointer or to a data-object pointer expression other than the `NULL` pointer. You may receive a run-time exception if the type of pointer held in the `open` pointer is not a pointer to a data object.

Note: `Open` pointers inhibit optimization and type checking. Use them only when absolutely necessary.

Understanding Other Pointer Rules

Before you use pointers in your program consider these rules:

A `NULL` pointer can be assigned to and compared for equality (`==`, `!=`) with a pointer of any type.

A `NULL` pointer cannot be used in a relational operation (`<`, `>`, `<=`, `>=`) with any pointer type. Since `void` pointers map to `open` pointers, relational operations between the `NULL` pointer constant and pointers to `void` are not allowed.

A pointer can be assigned to and compared for equality (`==`, `!=`) only with a pointer of the same type or with an `open` pointer; otherwise a compile-time error occurs.

Only pointers to data objects or `open` pointers that contain pointers to data objects can be used in relational operations (`<`, `>`, `<=`, `>=`); otherwise a compile-time error or run-time exception may occur.

Only pointers to data objects or `open` pointers that contain pointers to data objects can be used in arithmetic operations (`+`, `-`, `++`, `--`); otherwise a compile-time error or run-time exception may occur.

Note: The conditional expression `if (!ptr)` is equivalent to the expression `if (ptr == 0)`.

Declaring Pointer Variables

You can declare pointers to objects, functions (pointers to bound ILE procedures), and member functions using the C++ programming language. Pointers to program objects can be declared in one of two ways:

1. Declare a pointer to a typedef of a function that has been specified to have OS linkage with the `extern "OS"` linkage specification
2. Declare a `system` pointer (`_SYSPTR`)

You can declare variables of the other AS/400 pointer types by using the typedefs provided by the C++ `<pointer.h>` header file. These types are defined as pointers to void (`void *`). The **#pragma pointer** directives in the header file cause the compiler to associate these types with AS/400 pointer types.

Note: Pointers to OS linkage functions (programs) and `system` pointers (`_SYSPTR`) are data-type-compatible. You can use a `system` pointer to hold the address of a program and to call that program through the `system` pointer. A call through a `system` pointer containing the address of a system object that is not a program results in a run-time exception.

AS/400 pointer declarations are:

```
#include <pointer.h> // The pointer header file.

_SYSPTR sysp;      // A system pointer.
_SPCPTR spcp;      // A space pointer.
_INVPTR invp;      // An invocation pointer.
_OPENPTR opnp;     // An open pointer.
_LBLPTR lblp;      // A label pointer.

void (*fp) (int);  // A function pointer.

extern "OS" {typedef void (OS_FN_T) (int);} // Typedef of an OS linkage function.
OS_FN_T * os_fn_p; // An OS linkage function pointer.

int * ip;          // A pointer to a data object.
```

A function pointer is a pointer that points to either a bound procedure (function) within an ILE program object or an OS linkage program object (`system` pointer) in AS/400.

You can declare a pointer to a bound procedure (a function defined within the same ILE program object):

```

int fct1( void ) {...}
int fct2( void ) {...}
int (*fct_ptr)(void) = fct1;

int main()
{
    fct_ptr();          // Call fct1 using fct_ptr.
    fct_ptr = fct2;    // Dynamically set fct_ptr to fct2.
    fct_ptr();          // Call fct2 using fct_ptr.
}

```

You can declare a pointer to an AS/400 program as a function pointer with OS linkage. (If the `extern "OS"` linkage specification is omitted from the code, the compiler assumes that the function named `PGMCALL()` is a bound ILE procedure defined in this program.)

```

#include <miptrnam.h>
#include <iostream.h>

extern "OS" {typedef void (OS_fct_t) ( void );}

int main ( void )
{
    OS_fct_t *os_fct_ptr;
    char    pgm_name[10];
    cout <<"Enter the program name :"<<endl;
    cin >> pgm_name;

    // Dynamic assignment of a system pointer to program "MYPGM"
    // in *LIBL. The rslvsp MI library function will resolve to
    // this program at run time and return a system pointer to
    // the program object.

    os_fct_ptr = (OS_fct_t *)rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);

    os_fct_ptr();          // extern OS linkage *PGM call using a
                          // pointer.
}

```

This program demonstrates how to pass AS/400 pointers as arguments on a dynamic program call (`extern "OS"` linkage specification) to another C++ program.

There are two C++ programs. Program T2123DL8 passes several types of AS/400 pointers as arguments to Program T2123DL9.

```

// Program T2123DL8 passes several types of AS/400 pointers as arguments
// to another C++ program T2123DL9.

#include <iostream.h>
#include <pointer.h>

class PtrClass{
    public:

```

```

        PtrClass(_SPCPTR spc, _SYSPTR sys, void(*fp) ()):spcptr(spc), sysptr(sys),
        fnptr(fp) {}
        int CheckSpcptr(_SPCPTR s) {return s !=spcptr;}
        int CheckSysptr(_SYSPTR s) {return s !=sysptr;}
        int CheckFnptr(void (*f) ()) {return f !=fnptr;}

private:
        _SPCPTR spcptr;    // A space pointer.
        _SYSPTR sysptr;    // A system pointer.
        void (*fnptr)();    // A function pointer.
};

typedef void(*fntype)(); // A typedef is needed to override the
                        // extern "OS" for the fourth parameter
                        // of T2123DL9. extern "C++" cannot be used
                        // directly for arguments.

extern "OS" void T2123DL9 (PtrClass *, _SPCPTR, _SYSPTR, fntype);

void function1(void)    // A function definition.
{
    cout <<"Hello!"<<endl;
}

int main(void)
{
    int          i = 4;
    PtrClass PtrClass ((_SPCPTR)&i, (_SYSPTR)T2123DL9, &function1);

    // Call T2123DL9, passing the address of PtrClass and other
    // valid AS/400 pointer arguments.

    T2123DL9(PtrClass, (_SPCPTR)&i, (_SYSPTR)T2123DL9, &function1);
}

```

Program T2123DL9 receives the arguments and checks them to make sure that they were passed correctly.

```

// This program receives the arguments from T2123DL8 and checks them
// to make sure they were passed correctly.

```

```

#include <iostream.h>
#include <pointer.h>

```

```

class PtrClass {
public:
    PtrClass(_SPCPTR spc, _SYSPTR sys, void(*fp) ()):spcptr(spc), sysptr(sys),
    fnptr(fp) {}
    int CheckSpcptr(_SPCPTR s) {return s !=spcptr;}
    int CheckSysptr(_SYSPTR s) {return s !=sysptr;}
    int CheckFnptr(void (*f) ()) {return f !=fnptr;}
}

```

```

private:
    _SPCPTR spcptr;    // A space pointer.
    _SYSPTR sysptr;   // A system pointer.
    void (*fnptr)();  // A function pointer.
};

int main( int argc, char **argv)
{
    _OPENPTR    openptr;    // An open pointer.
    _SPCPTR     spcptr;     // A space pointer.
    _SYSPTR     sysptr;     // A system pointer.
    void        (*fnptr)(); // A function pointer.
    PtrClass    *ptr_struct_ptr;
    int         error_count = 0;

    // Receive the class pointer passed into a local variable.

    ptr_struct_ptr = (PtrClass *)argv[1];

    // Receive the AS/400 pointers passed into an open pointer,
    // then assign them to pointers of their own type.

    openptr = (_OPENPTR)argv[2];
    spcptr = openptr;           // A space pointer.
    openptr = (_OPENPTR)argv[3];
    sysptr = openptr;         // A system pointer.
    openptr = (_OPENPTR)argv[4];
    fnptr = (void (*)())openptr; // A function pointer.

    // Test the pointers passed with the pointers in ptr_struct_ptr.

    if (ptr_struct_ptr->CheckSpcptr(spcptr))
        ++error_count;
    if (ptr_struct_ptr->CheckSysptr(sysptr))
        ++error_count;
    if (ptr_struct_ptr->CheckFnptr(fnptr))
        ++error_count;

    if (error_count > 0)
        cout <<"Pointers not passed correctly."<<endl;
    else
        cout<<"Pointers passed correctly."<<endl;
    return 0;
}

```

The output is:

```

Pointers passed correctly.
Press ENTER to end terminal session.

```

Using Pointer Casting

In the C++ language, *casting* is a construct used to convert a data object from one type to another. There are several rules to consider when using pointer casting, especially when casting pointers to and from integers, when pointers do not point to data objects (such as pointers to functions), and in the case of pointer to pointer casts, (when a pointer to one type is converted to a pointer to another type):

A pointer can only be assigned to a compatible pointer type with the same type qualifiers or with more type qualifiers than the assigned pointer type. For relational or equality expressions (<, >, <=, >=, ==, !=), pointer conversion brings the operands to the same type qualifiers when possible.

A pointer can only be cast to another pointer of the same AS/400 pointer type. If the compiler detects a type mismatch in an expression a compile-time error occurs.

An `open (void)` pointer can hold a pointer of any type. Casting an `open` pointer to other pointer types and casting other pointer types to an `open` pointer does not result in a compile-time error. You may receive a run-time exception if the pointer contains a value unsuitable for the context. A `system` pointer in a pointer addition expression results in a run-time exception.

When a pointer to a data object is converted to a `signed` or `unsigned integer` type, the offset of the pointer is returned. If the pointer is `NULL`, a value of zero (0) is returned. It is not possible to determine whether the conversion originated from a `NULL` pointer or a valid pointer with an offset 0.

When a `function (procedure) pointer`, `system pointer`, `invocation pointer`, `label pointer`, or `suspend pointer` is converted to a `signed` or `unsigned integer` type, the result is always zero.

When an `open` pointer is converted to a `signed` or `unsigned integer` type, if the `open` pointer contains a `space` pointer at the time of the conversion, the offset of the `space` pointer is returned; otherwise a run-time exception occurs.

When a `signed` or `unsigned integer`, regardless of the content of the data object, is converted to a pointer type, the result of the conversion is a `NULL` pointer.

You must cast a `void` pointer before using it in a pointer comparison or assignment unless it is a `NULL` pointer.

These are some results of casting AS/400 pointers:

```
#include <pointer.h>

int    i;
int    *ip;
void (*fp)(void);
_SYSPTR sysp;
_OPENPTR opnp;

int main( void )
{
    i = ( int ) ip;    // Pointer to integer (ip) cast to integer.
```



```

        // i contains the offset portion of the
        // pointer ip.

ip = ( int * ) i;    // Integer cast to pointer to integer.
                    // A NULL pointer is assigned to ip.

i = ( int ) fp;     // Function pointer cast to integer.
                    // i is zero.

fp = (void (*)( )) i; // Integer cast to function pointer.
                    // A NULL pointer is assigned to fp.

i = ( int ) sysp;   // System pointer cast to integer.
                    // i is zero.

sysp = ( _SYSPTR ) i; // Integer cast to system pointer.
                    // A NULL pointer is assigned to sysp.

opnp = &i;         // Address of an integer is assigned to
i = ( int ) opnp; // open pointer opnp. Open pointer
                    // containing the address of an integer
                    // cast to an integer.
                    // i is assigned the offset portion of the
                    // space pointer contained in the open
                    // pointer opnp.

opnp = ( _SYSPTR ) i; // Integer cast to _SYSPTR.
                    // A NULL pointer is assigned to opnp.

opnp = sysp;      // _SYSPTR assigned to open pointer. Open
i = ( int ) opnp; // pointer containing a _SYSPTR cast to
                    // integer. Run-time exception occurs on
                    // the assignment.
}

```

Chapter 11. International Locale Support

International locale support allows programs to modify their behavior according to the user's language environment. This support has three key components:

- Programming tools that create language-specific data
- Programming interfaces (functions) that allow access to this data
- Methods of creating programs that are automatically sensitive to the language environment in which they run

See the *International Application Development*, for information on the definition and creation of international locale objects.

This section describes:

- "Elements of a Language Environment"
- "Locales" on page 186
- "POSIX Locale Definition and *LOCALE Support" on page 186
- "Creating Locales" on page 186
- "Categories Used in a Locale" on page 186
- "Locale-Sensitive Run-Time Functions" on page 189

Elements of a Language Environment

The typical elements of a language environment are:

Native language

The natural language of the user.

Character sets and coded character sets

A coded character set is created by mapping the characters of a character set onto a set of code points (hexadecimal values). See the *National Language Support* for information about coded character sets.

Collating and ordering

The relative order of characters used for sorting.

Character classification

The type of a character (for example, alphabetic, numeric) in a character set.

Character case conversion

The mapping between uppercase and lowercase characters in a character set.

Date and time format

The format of date and time data (for example, order of the months, names of the weekdays).

Format of numbers and monetary quantities

The format of numbers and monetary quantities (for example, numeric grouping, decimal-point character, monetary symbols).

Format of affirmative and negative system responses

The format of affirmative and negative system responses

Locales

A locale is a system object that specifies how language-specific data is processed, printed and displayed. A locale is made up of categories that describe the character set, collating sequence, date and time representation and monetary representation of the language environment in which it is used. Using locales and locale-sensitive functions, you can create programs that are independent of language, cultural data, or character set, yet are sensitive to the language environment of the user.

POSIX Locale Definition and *LOCALE Support

Locale definition source files conform to the IEEE POSIX P1003.2 standard and are shipped with the system in the optionally installable library QSYSLOCALE. One *LOCALE object, the C locale as defined by the POSIX standard, is provided with the system. Other locales of type *LOCALE can be created with the CRTLOCALE command from the locale source definition members in the QSYSLOCALE library.

Creating Locales

On the AS/400 system, *LOCALE objects are created with the CRTLOCALE command, specifying the name of the file member containing the locale's definition source, and the CCSID to be used for mapping the characters of the locale's character set to their hexadecimal values.

A locale definition source member contains information about a language environment. This information is divided into a number of distinct categories which are described in "Categories Used in a Locale." One locale definition source member characterizes one language environment.

Characters are represented in a locale definition source member with their symbolic names. The mapping between the symbolic names, the characters they represent and their associated hexadecimal values is based on the CCSID value specified on the CRTLOCALE command.

Figure 20 on page 187 shows how a locale of type *LOCALE is created.

Categories Used in a Locale

A locale and its definition source member contain these categories:

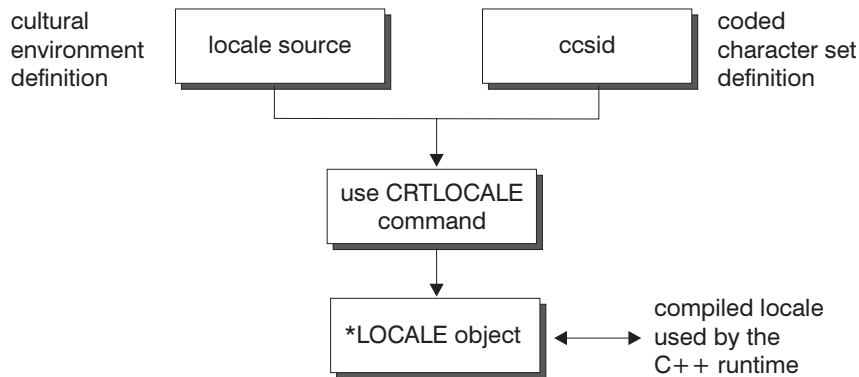


Figure 20. How a Locale of Type *LOCALE is Created

Category Purpose

LC_COLLATE

Defines the collation relations among the characters. Affects the behavior of the collating functions `strcoll`, `strxfrm`, `wscoll` and `wcsxfrm`.

LC_CTYPE

Defines character types, such as upper-case, lower-case, space, digit, and punct. Affects the behavior of character handling functions.

LC_MESSAGES

Defines the format and values for responses from the program.

LC_MONETARY

Defines the monetary names, symbols, punctuation, and other details. Affects monetary information returned by `localeconv`

LC_NUMERIC

Defines the decimal-point (radix) character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by `localeconv`.

LC_TIME

Defines the date and time conventions, such as calendar used, time zone, and days of the week. Affects the behavior of time display functions.

LC_TOD

Defines time zone difference, time zone name, and Daylight Savings Time start and end.

Setting an Active Locale for Your Program

C++ programs running on the AS/400 and using locales of type *LOCALE have an active locale which is scoped to the activation group of the program. The active locale determines the behavior of the locale-sensitive functions in the C library. The active locale can be set explicitly with a call to `setlocale()`. See the *C Library Reference* for information on using `setlocale()`.

If the active locale is not set explicitly by a call to `setlocale()`, it is implicitly set by the C++ run time at program activation time. The following information describes how the C++ run time sets the active locale when a program is activated:

If the user profile has a value for the `LOCALE` parameter other than `*NONE` (the default) or `*SYSVAL`, that value is used for the program's active locale.

If the value of the `LOCALE` parameter in the user profile is `*NONE` the default "C" locale is the active locale.

If the value of the `LOCALE` parameter in the user profile is `*SYSVAL`, the locale associated with the system value `QLOCALE` is used for the program's active locale.

If the value of `QLOCALE` is `*NONE`, the default "C" locale becomes the active locale.

See the *International Application Development* for information about how locales of type `*LOCALE` are enabled.

Using Environment Variables to Set the Active Locale

A program's active locale is set either implicitly at program startup, or explicitly by a call to `setlocale()`. The `setlocale()` function takes two arguments: an integer representing the locale category whose values are needed for the active locale, and the name of the locale from which the values are to be taken. The name of the locale can be either "C", "POSIX", the fully-qualified path name of a locale object of type `*LOCALE`, or a null string("").

When the locale argument of `setlocale()` is specified as a null string(""), `setlocale()` sets the active locale according to the environment variables defined for the job in which the program is running. You can create environment variables having the same names as the locale categories and specify the locale to be associated with each environment variable. The `LANG` environment variable is automatically created during job initiation when you specify a locale path name for the `LOCALE` parameter in your user profile or for the `QLOCALE` system value.

When a program calls `setlocale(category, "")`, the locale-related environment variables defined in the current job are checked to find the locale name or names to be used for the specified category. The locale name is chosen according to the first of the following conditions that applies:

1. If the environment variable `LC_ALL` is defined and is not null, the value of `LC_ALL` is used for the specified category. If the specified category is `LC_ALL`, that value is applied to all categories.
2. If the environment variable for the category is defined and is not null, then the value specified for the environment variable is used. For the `LC_ALL` category, if individual environment variables (for example, `LC_CTYPE`, `LC_MONETARY`, and so on) are defined and are not null, then their values are used for the categories that correspond to the environment variables. This could result in the locale information for each category being retrieved from a different locale object.

3. If the environment variable LANG is defined and is not null, the value of the LANG environment variable is used.
4. If no non-null environment variable is present to supply a locale value, the default "C" locale is used.

If the locale specified for the environment variable is found to be invalid or non-existent, `setlocale()` returns `NULL` and the program's active locale remains unchanged.

For `setlocale(LC_ALL, "")`, if the locale names found identify valid locales on the system, `setlocale()` returns a string naming the locale associated with each locale category. Otherwise, `setlocale()` returns `NULL` and the program's locale remains unchanged.

Locale-Sensitive Run-Time Functions

The `asctime()`, `asctime_r()`, `ctime()`, `ctime_r()`, `fprintf()`, `gmtime()`, `gmtime_r()`, `isalnum()` to `isxdigit()`, `iswalnum()` to `iswxdigit()*`, `localeconv()`, `localtime()`, `localtime_r()`, `mblen()`, `mbrlen()*`, `mbsinit()*`, `mbrtowc()*`, `mbsrtowcs()*`, `mbstowcs()`, `mbtowc()`, `mktime()`, `nl_langinfo()*`, `printf()`, `setlocale()`, `sprintf()`, `strcoll()`, `strfmon()*`, `strftime()`, `strptime()*`, `strxfrm()`, `time()`, `tolower()`, `toupper()`, `tolower()*`, `toupper()*`, `vfprintf()`, `vprintf()`, `vsprintf()`, `wcrtomb()*`, `wscoll()*`, `wcsrtombs()*`, `wcstombs()`, `wcswidth()*`, `wcsxfrm()*`, `wctomb()`, `wctype()*`, `wcwidth()*` are run-time functions sensitive to locales.

Note: * sensitive to *LOCALE objects only

Part 4. Working with AS/400 C++ Features

This part explains how to:

- Code program and procedure calls to other AS/400 programming languages
- Pass parameters to other AS/400 programming languages
- Qualify library calls
- Create classes to use in ILE
- Change program and procedure names
- Generate and include template function definitions
- Use try-catch-throw, ILE condition handlers, C signal function and cancel handlers
- Percolate and promote exceptions
- Understand the ILE control boundary and exception handling
- Monitor for AS/400 system exceptions
- Port ILE C and Windows C and C++ programs to VisualAge for C++ for AS/400
- Use run-time type information

Chapter 12. Calling Conventions

You can call OPM, EPM or ILE programs from C++ using program calls. A program call is a call made to a program object (*PGM). Unlike OPM and EPM programs, ILE programs are not limited to using program calls. ILE programs can use static procedure calls or procedure pointer calls to call other procedures.

This section includes:

- “Introducing Program and Procedure Calls”
- “Calling a Program Using a Linkage Specification” on page 196
- “Calling an ILE Procedure Using a Linkage Specification” on page 197
- “Passing Parameters” on page 197
- “Changing the Names of Programs and Procedures” on page 215
- “Creating C++ Classes for Use in ILE” on page 216
- “Calling OPM Programs” on page 221
- “Calling ILE Programs” on page 229
- “Calling an ILE C++ Program” on page 236
- “Calling an EPM C Program” on page 237
- “Calling ILE-Bindable APIs” on page 239
- “Passing Operational Descriptors” on page 242

Note: The terms *parameter* and *argument* are used interchangeably.

Introducing Program and Procedure Calls

C++ is one of the programming languages supported in the Integrated Language Environment (ILE). In ILE it is possible to call either a program (*PGM) or an ILE procedure. The calling program must identify whether the target of the call statement is a program or an ILE procedure. Different C++ calling conventions exist for programs and for ILE procedures.

Program processing within ILE occurs at the procedure level. ILE programs consist of one or more modules which in turn consist of one or more procedures. A C++ module may contain only one `main()` procedure, but can contain many other procedures (functions). Other ILE languages, however, may allow only one procedure. A program call is a special form of procedure call; it is a call to the program entry procedure. A program entry procedure is the procedure designated at program creation time to receive control when a program is called. This is the same as calling another program's `main()` function.

Calling Programs

When executing dynamic calls, the called program's name is resolved to an address at run time, just before the calling program passes control to the called program for the first time. Program calls are dynamic calls.

Calls to an ILE program, an EPM program, or an OPM program are program calls. A call to a non-bindable API is a program call.

When an ILE program is called, the program entry procedure receives the program parameters and is given initial control for the program. All procedures within the program become available for procedure calls.

Calling Procedures

Unlike OPM programs, ILE programs are not limited to using program calls. ILE programs can also use static procedure calls or procedure pointer calls to call other procedures. Procedure calls are bound calls.

A *static procedure call* is a call to an ILE procedure where the name of the procedure is resolved to an address during binding (static call). Run-time performance of using static procedure calls is faster than run-time performance using program calls.

```
extern "C" void foo ();
main ()
{
    foo (); // static procedure call
}
```

Note: The term static procedure call does not refer to static storage class but refers to a bound procedure call within a bound module or service program. If the static call is to a procedure written in a language other than C or C++, operational descriptors can be used to resolve the differences in the representation of character strings, if values of this data type are passed as arguments.

Procedure pointer calls provide a way to call a procedure dynamically. You can pass a procedure pointer as a parameter to another procedure which then runs the procedure specified. You can manipulate arrays of procedure names or addresses to dynamically route a procedure call to different procedures. If the called procedure is in the same activation group, the cost of a procedure pointer call is almost identical to the cost of a static procedure call.

Using either type of procedure call, you can call a procedure in a separate module within the same ILE program or service program, or a procedure in a separate ILE service program. Any procedure that can be called using a static procedure call can also be called through a procedure pointer.

Introducing the Call Stack

The *call stack* is a list of call stack entries, in a last-in-first-out (LIFO) order. A *call stack entry* is a call to a program or procedure. There is one call stack per job.

When an ILE program is called, the program entry procedure is first added to the call stack. After the program entry procedure is called, control is given to the main entry point in the program (`main()` for C or C++) which is pushed onto the stack.

Figure 21 shows a call stack for a program consisting of an OPM program which calls an ILE program consisting of two modules: a C++ module containing the program entry procedure and the associated user entry procedure, and a C module containing a regular procedure. The most recent entry is at the bottom of the stack.

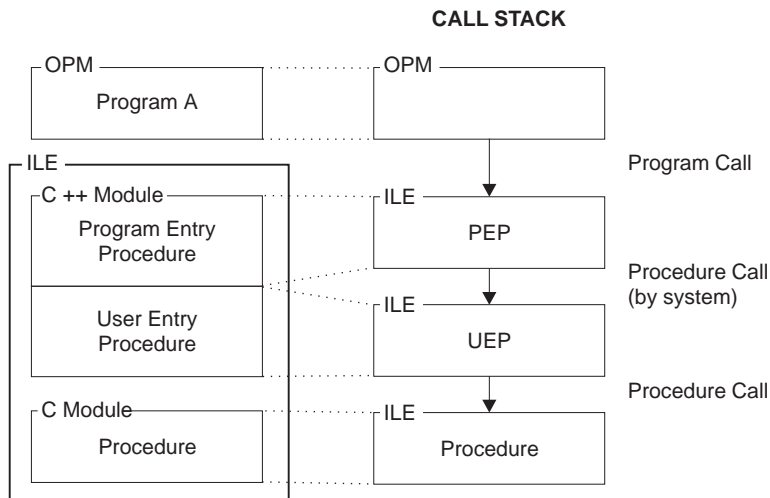
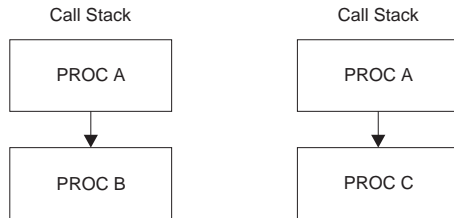


Figure 21. Program and Procedure Calls on the Call Stack

Note: In a program call, the calls to the program entry procedure and the user entry procedure (UEP) occur together, since the call to the UEP is automatic. In later diagrams involving the call stack, the two steps of a program call are combined.

An ILE C++ procedure which is on the call stack can be called before it returns to its caller. This is a *recursive* procedure call. This is different from an ILE COBOL procedure which when on the call stack cannot be called until it returns to its caller. This is a *non-recursive* procedure call. Therefore, be careful not to call from ILE C++ procedures another ILE COBOL procedure which might call an already active ILE COBOL procedure.

Assume that procedure A is an ILE C++ procedure, procedure B and C are ILE COBOL procedures, and that these procedures are in the same program. If procedure A calls procedure B, then procedure B can call neither procedure A nor B. If procedure B returns and if procedure A then calls procedure C, procedure C can call procedure B but not procedure A or C. See Figure 22 on page 196.



PROC B cannot call PROC A or PROC B; only PROC C.

PROC C cannot call PROC A or PROC C; only PROC B.

Figure 22. ILE C++ Procedures Cannot Call Other Active ILE COBOL Procedures

OPM COBOL programs already on the call stack cannot be called.

Calling a Program Using a Linkage Specification

You can call OPM, ILE, or EPM programs from a C++ program. OPM, ILE or EPM programs can also call a C++ program. See Table 1 on page 19 for a list of AS/400 programming languages you can call.

C++ provides a linkage specification to enable program calls and sharing of data between them. The syntax is:

```
extern "string-literal" { declaration-list }

extern "string-literal" declaration
```

The *"string-literal"* is used to specify the linkage associated with a particular function. The string literals used in linkage specifications are case-insensitive. The valid string literals for the linkage specification to call programs are:

"OS" OS linkage call

"OS nowiden"

OS linkage call without widened parameters. See "OS Linkage" on page 201 for details.

If you want a C++ program to call an ILE, OPM, or EPM program (*PGM), use the `extern "OS"` linkage specification in your C++ source to tell the compiler that the called program is an external program, not a bound ILE procedure. For example, if you want a C++ program to call an OPM COBOL program (*PGM) this `extern "OS"` linkage specification in your C++ source tells the compiler that `COBOL_PGM` is an external program, not a bound ILE procedure.

```
extern "OS" void COBOL_PGM(void);
```

If you want an ILE, OPM or EPM program to call a C++ program, use the ILE, OPM, or EPM language-specific call statement.

Calling an ILE Procedure Using a Linkage Specification

Unlike OPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can also use static procedure calls or procedure pointer calls.

Notes:

1. ILE procedures within an *activated* ILE program can be accessed using static procedure calls or procedure pointer calls. New ILE programs that have not been activated yet must be called by a dynamic program call. There are other situations in which dynamic calls occur.
2. EPM C and Pascal procedures or functions cannot call C++ procedures.
3. The term procedure in ILE is similar to the term function in C++. A C++ function is an ILE procedure.

C++ provides a linkage specification to enable procedure calls and the sharing of data between the C++ caller and the called procedure. See "Calling a Program Using a Linkage Specification" on page 196 for the syntax.

The valid string literals for the linkage specification to call ILE procedures are:

Linkage Specification Type of Procedure Called

"C++"	ILE C++ procedure (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
"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

Passing Parameters

To share data between programs or between procedures, you need to pass the called program or procedure parameters which both programs can use. In C++ you use the linkage specification to tell the compiler which parameter passing convention to use on the external call.

When passing parameters from C++ to a different high-level language (HLL) consider:

Parameter passing style of the HLLs

Each HLL has its own way of passing parameters. Parameters can be passed as a pointer to the parameter value, to a copy of the value, or to the value itself. C++ passes parameters in all three ways. See "Using Default Parameter Passing Styles" on page 203 for information on these styles.

Interlanguage data compatibility

Different HLLs support different ways of representing data. Pass only parameters which have a data type common to the calling and called program or procedure. If you are not sure of the exact format of the data that is passed to your program, you may specify to the users of your procedure that an operational descriptor can be passed to provide additional information regarding the format of the passed parameters. See "Using Operational Descriptors" on page 204.

Passing Parameters in C++

Table 7 shows the effect of using the using different linkage types on passing parameters:

<i>Table 7 (Page 1 of 2). Effects of Various Linkage Specifications</i>				
Linkage	Name Mangled	Parameter Passing	Parameter Widening	Comments
"C++"	Yes	C++	No	This is the default.
"C"	No	C++	Yes	Used to call a function (procedure) written in ILE C.
"C nowiden"	No	C++	No	Used to call a function (procedure) written in ILE C.
"OS"	No	OS	Yes	Used to call an external program written in any OPM/EPM/ILE language.
"OS nowiden"	No	OS	No	Used to call an external program written in any OPM/EPM/ILE language.
"RPG"	No	OS	No	Used to call a procedure written in ILE RPG. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"COBOL"	No	OS	No	Used to call a procedure written in ILE COBOL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.

Linkage	Name Mangled	Parameter Passing	Parameter Widening	Comments
"CL"	No	OS	No	Used to call a procedure written in ILE CL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"ILE"	No	OS	Yes	Used to call a procedure written in an ILE language. Identical to RPG, COBOL, and CL specifications. If the particular language in which the function was written is unknown to the programmer, use this linkage. If you have C code that uses the #pragma argument directive and you plan to port this code to C++ then use the <code>extern "ILE"</code> linkage specification.
"ILE nowiden"	No	OS	No	Used to call a procedure written in an ILE language.
"VREF"	No	OS	Yes	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.
"VREF nowiden"	No	OS	No	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.

The `extern` keyword followed by the string literals "RPG", "COBOL", or "CL" are used to specify that the function has "ILE" linkage. These string literals perform the same function as the **#pragma argument** directive in ILE C. The "VREF" linkage also performs the same as the `VREF` parameter on the **#pragma argument** directive.

The *string-literal* is case-insensitive: `extern "OS NOWIDEN"`, `extern "OS nowiden"`, and `extern "os Nowiden"`, although different in case are all handled in the same way.

The name of the function must follow the naming conventions of the other language. For OS linkage specifications, the program name and all AS/400 objects must be uppercase.

A typedef of a function can be declared to have linkage information. After the typedef is declared, it can be used to declare functions of a particular linkage. The typedef declaration must be enclosed by braces `{ }`.

A function cannot be assigned directly to a pointer to a function with a different linkage. A type cast may be used to make this assignment possible. Type casting helps you eliminate parameter mismatching problems without excess constraint.

To override a function but not to override extern "OS" use a type cast:

```
extern "ILE"
{
    typedef void (*ILE) ();
}
extern "C++"
{
    typedef void (*CPP) ();
}
ILE pILE;
CPP pCPP = (CPP) pILE;
```

Functions that take function pointers as parameters may not be overloaded based on the linkage of the function pointers:

```
// Using the typedef declarations above
void foo (OS);
void foo (CPP); // undefined behavior, foo already declared
```

Functions that are defined with non-C++ linkage specifications accept parameters using the appropriate convention for that linkage. You do not need to widen parameters:

```
extern "C" void foo (char); // chars are widened in C
// In another compilation unit we then have
extern "C" void foo (char c) // this parameter is correctly widened
{
    // implementation of foo (char);
}
```

Attempting to define a function with either extern "OS", extern "OS nowiden", or extern "builtin" linkage results in undefined behavior:

```
extern "OS" void FOOPGM (char); // declaration: OK
extern "OS" void FOOPGM (char c) // definition: undefined behavior
{
    // implementation of FOOPGM
}
```

Functions FO01(), FO02(), and FO03() are all declared as OS linkage functions. Functions FO01() and FO02() are declared by using the declaration-list syntax. FO03() is declared by using the simple declaration.

```
extern "OS" {
    void FO01 (char, char *);
    void FO02 (int, int *);
}
extern "OS" double FO03 (double, double *);
```

The declaration of a function pointer is:

```
extern "OS" {
    typedef void (*fp) (char);
}
fp F00;
```

Function `FOO()` is declared to be a function pointer of type `fp`.

Data objects can be declared inside the `extern` linkage declaration:

```
extern "OS" {
    int a1;
}
extern "OS" int a2;
```

Variable `a1` is defined while variable `a2` is only declared.

The widening rules for all the linkage specifications shown in Table 7 on page 198 are:

Any data type that is smaller than `int` is widened to `int`

`float` is widened to `double`

Address and Data pointers are not widened

`struct` has the same structure and information as the `struct` parameter passed

In C++ linkage specifications, function identifiers are mangled. In all other linkage specifications, all function identifiers are identical to the exported names unless changed by the `#pragma map` directive.

OS Linkage

The `extern` specifier followed by the *string-literal* `"OS"` or the *string-literal* `"OS nowiden"` is used to declare external programs. These programs may then be called in the same way as a regular function.

When an OS linkage function is called from a C++ program, the compiler generates code and performs the following tasks in sequence:

1. If `extern "OS"` is used, then the parameters and return value are widened.
If `extern "OS nowiden"` is used, then the parameters and return value passed between programs are not widened.
2. Parameters that are passed by value are copied to temporary variables and the addresses of the temporary variables are passed to the called program.
If a temporary variable is created for a structure, the temporary variable has the same structure and information as the `struct` parameter passed.
3. Parameters that were passed by reference are still passed by reference.
4. Arrays and pointers are passed by reference.
5. If the argument you are passing is an array name or a pointer, then the argument is passed directly, and a temporary variable is not created. The data referenced by the array or pointer can be changed by the called program.
6. The function name is not mangled.

The program name that the C++ program calls must be in uppercase. You can use the `#pragma map` directive to map an internal identifier longer than 10 characters to an

OS/400-compliant object name (10 characters or less) in your program. See "Changing the Names of Programs and Procedures" on page 215.

The return code for the program call can be retrieved by declaring the program to return an integer:

```
extern "OS" int PGMNAME(void);
```

The value returned on the call is the return code for the program call. If the program being called is a C++ program, this return code can be accessed using the `_LANGUAGE_RETURN_CODE` macro defined in the header file `<mlib.h>`. A C++ program returns four bytes in the `_LANGUAGE_RETURN_CODE`. If the program being called is an EPM or OPM program, this return code can be accessed using the AS/400 Retrieve Job Attributes (RTVJOBA) command.

When a function is called from an OS linkage `function` pointer, the compiler generates the same code sequence it does when calling an OS linkage function.

Nonpointer arguments are passed by value reference, and changes made to the variables in the called program are not reflected in the calling C++ program.

C Linkage

Specifying C linkage for a function tells the compiler:

- Parameters are passed using C++ conventions
- Parameters for functions declared with `extern "C"` are widened
- The function name is not mangled

The `extern` keyword followed by the *string-literal* `"C"` or the *string-literal* `"C nowiden"` is used to specify that the function is declared to have "C" linkage instead of "C++" linkage.

ILE, CL, COBOL, and RPG Linkage

Specifying an ILE, CL, COBOL, or RPG linkage for a function tells the compiler:

- Arguments passed as values or nonpointer arguments are copied to temporary variables and the addresses of the temporary variables are passed to the called program
- Pointer arguments are passed directly to the called program
- If `extern "ILE nowiden"` is used, then the parameters and return value passed between programs are not widened; specifying any other ILE linkage widens the parameters
- Function names are not mangled

VREF Linkage

Specifying a VREF linkage is identical to specifying an ILE linkage except that pointer parameters are stored in a temporary variable and the address of the temporary variable is passed as the actual argument.

Using Default Parameter Passing Styles

ILE C++ uses the same calling mechanisms for calling any ILE HLL program or procedure: `extern` linkage specification. ILE C++ passes and receives parameters using three passing methods: by value directly, by value indirectly, and by reference. Other ILE languages may have different methods of passing data. See Table 8.

by value, directly

The value of the data object is placed directly into the argument list.

by value, indirectly

The value of the data object is copied to a temporary location. The address of the copy, a pointer, is placed into the argument list.

by reference

A pointer to the data object is placed into the argument list. Changes made by the called procedure to the argument are reflected in the calling procedure.

Table 8 shows the common parameter-passing methods for the ILE programs.

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly or by reference	by value, directly or by reference
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly, or by value, indirectly or by reference
ILE COBOL	by reference or by value, indirectly	by reference or by value, indirectly
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

Table 9 shows the common parameter passing methods for the ILE procedures.

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly	by value, directly
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly, or by value, indirectly or by reference
ILE COBOL	by reference or by value, indirectly	by reference
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

To pass or receive parameters to or from procedure calls involving other ILE languages, especially ILE C, ILE RPG, or ILE COBOL, you must ensure that the other procedure is set up to accept data by reference.

Using Operational Descriptors

To pass a parameter to a procedure even though the data type is not precisely known to the called procedure you can use operational descriptors. *Operational descriptors* provide descriptive information to the called procedure regarding the form of the argument. This information allows the procedure to properly interpret the passed parameter. Only use operational descriptors when they are expected by the called procedure.

Note: The C++ compiler supports operational descriptors for describing null-terminated strings. A character string in C++ is defined by: `char string_name[n]`, `char * string_name`, or *string-literal*.

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.

To use operational descriptors, you specify a **#pragma descriptor** directive in your source to identify functions whose arguments have operational descriptors. Operational descriptors are then built by the calling procedure and passed as hidden arguments to the called procedure.

The syntax is:

```
# pragma descriptor (void function_name(od_specifiers))
```

See the *C++ Language Reference* for information on operational descriptors.

Understanding Data-Type Compatibility

Each high-level language has different data types. When you want to pass data between programs written in different languages, you must be aware of these differences.

Some data types in the ILE C++ programming language have no direct equivalent in other languages. You can simulate data types in other languages using ILE C++ data types.

Note: No data-type compatibility tables are shown for C. You can use the C++ tables since C and C++ data types are the same except for the packed decimal data type. In C++ the packed decimal data type is implemented as a class. The packed decimal data type in ILE C and the binary coded decimal class in C++ are binary compatible.

Table 10 on page 205 shows the ILE C++ data type compatibility with ILE RPG.

Table 10 (Page 1 of 2). ILE C++ Data-Type Compatibility with ILE RPG			
ILE C++ declaration in prototype	ILE RPG D spec, columns 33 to 39	Length	Comments
char[n]	nA	n	An array of characters where n=1 to 32766
char *	*	16	A pointer
char	1A	1	An indicator which is a variable starting with *IN
char[n]	nS 0	n	A zoned decimal
char[2n]	nG	2n	A graphic added
char[2n+2]	Not supported	2n+2	A graphic data type
_Packed struct {short i; char[n]}	data structure	n+2	A variable length field where i is the intended length and n is the maximum length
char[n]	D	8, 10	A date field
char[n]	T	8	A time field
char[n]	Z	26	A timestamp field
short int	5I 0	2	An integer field
short unsigned int	5U 0	2	An unsigned integer field
int	10I 0	4	An integer field
unsigned int	10U 0	4	An unsigned integer field
long int	10I 0	4	An integer field
long unsigned int	10I 0	4	An unsigned integer field
struct {unsigned int : n}x;	Not supported	1, 2, 4	A 4-byte unsigned integer, a bitfield
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
_DecimalT<n,p>	nP p	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
union.element	<type> with keyword OVERLAY(longest field)	length of longest union member	An element of a union
data_type[n]	<type> with keyword DIM(n)	16	An array to which C++ passes a pointer

Table 10 (Page 2 of 2). ILE C++ Data-Type Compatibility with ILE RPG

ILE C++ declaration in prototype	ILE RPG D spec, columns 33 to 39	Length	Comments
struct or class	data structure	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	* with keyword PROCPTR	16	A 16-byte pointer

Note: ¹All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.

Table 11 shows the ILE C++ data-type compatibility with ILE COBOL.

Table 11 (Page 1 of 2). ILE C++ Data-Type Compatibility with ILE COBOL

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) DISPLAY.	n	A zoned decimal
wchar_t[n]	PIC G(n) DISPLAY.	2n	A graphic data type
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	PIC X(n).	6	A date field
char[n]	PIC X(n).	5	A day field
char	PIC X.	1	A day-of-week-field
char[n]	PIC X(n).	8	A time field
char[n]	PIC X(n).	26	A time stamp field
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999
short int	PIC S9(4) BINARY.	2	A 2-byte signed integer with a range of -9999 to +9999
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	USAGE IS INDEX	4	A 4-byte integer
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999

Table 11 (Page 2 of 2). ILE C++ Data-Type Compatibility with ILE COBOL			
ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
long int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	1, 2, 4	Bitfields can be manipulated using hex literals
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n,p>	PIC S9(n-p)V9(p) COMP-3.	n/2+1	A packed decimal. In C++, this is a binary coded decimal class and not a data type.
_DecimalT<n,p>	PIC S9(n-p) 9(p) PACKED-DECIMAL.	n/2+1	A packed decimal. In C++ this is a binary coded decimal class not a data type.
union.element	REDEFINES	length of longest union member	An element of a union
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
struct or class	01 record 05 field1 05 field2	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	PROCEDURE-POINTER	16	A 16-byte pointer to a procedure
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
Not supported.	PIC S9(18) BINARY.	8	An 8-byte integer
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.			

Table 12 on page 208 shows the ILE C++ data-type compatibility with ILE CL.

ILE C++ declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. CHGVAR &V1 VALUE(&V *TCAT X'00') where &V1 is one byte bigger than &V
char	*LGL	1	Holds '1' or '0'
_Packed struct {short i; char[n]}	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported	1, 2, 4	A 1, 2 or 4 byte signed or unsigned integer
float constants	CL constants only	4	A 4 or 8 byte floating point
_DecimalT<n,p>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.
union.element	Not supported	length of longest union member	An element of a union
struct or class	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Table 13 shows the ILE C++ data-type compatibility with OPM RPG.

ILE C++ declaration in prototype	OPM RPG I spec, DS sub-field columns spec	Length	Comments
char[n]	1 10	n	An array of characters where n=1 to 32766
char	*INxxxx	1	An indicator which is a variable starting with *IN
char[n]	1 nd (d>=0)	n	A zoned decimal The limit of n is 30
char[2n+2]	Not supported	2n+2	A graphic data type
_Packed struct {short i; char[n]}	Data structure	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	char	6, 8, 10	A date field

Table 13 (Page 2 of 2). ILE C++ Data-Type Compatibility with OPM RPG			
ILE C++ declaration in prototype	OPM RPG I spec, DS sub-field columns spec	Length	Comments
char[n]	char	8	A time field
char[n]	char	26	A time stamp field.
short int	B 1 20	2	A 2-byte signed integer with a range of -9999 to +9999
int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	Not supported	1, 2, 4	A 4-byte unsigned integer, a bitfield
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	Not supported	16	A pointer
_DecimalT<n,p>	P 1 n/2+1d	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
union.element	data structure subfield	length of longest union member	An element of a union
data_type[n]	E-SPEC array	16	An array to which C++ passes a pointer
struct or class	data structure	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.			

Table 14 on page 210 shows the ILE C++ data-type compatibility with OPM COBOL.

ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) USAGE IS DISPLAY	n	A zoned decimal The limit of n is 18.
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length
char[n]	PIC X(n).	6, 8, 10	A date field
char[n]	PIC X(n).	8	A time field
char[n]	PIC X(n).	26	A time stamp field
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999.
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	1, 2, 4	Bitfields can be manipulated using hex literals.
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n,p>	PIC S9(n-p)V9(p) COMP-3.	n/2+1	A packed decimal The limits of n and p are 18. In C++, this is a binary coded decimal class and not a data type.
union.element	REDEFINES	length of longest union member	An element of a union
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer

ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
struct or class	01 record	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
Note: ¹ All structures must be packed. Classes with <code>virtual</code> functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C all nested structures are packed.			

Table 15 shows the ILE C++ data-type compatibility with CL.

ILE C++ declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. <code>CHGVAR &V1 VALUE(&V *TCAT X'00')</code> where &V1 is one byte bigger than &V. The limit of n is 9999.
char	*LGL	1	Holds '1' or '0'
_Packed struct {short i; char[n]}	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported	1, 2, 4	A 1, 2 or 4 byte signed or unsigned integer.
float constants	CL constants only	4	A 4 or 8 byte floating point
_DecimalT<n,p>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.
union.element	Not supported	length of longest union member	An element of a union

Table 15 (Page 2 of 2). ILE C++ Data-Type Compatibility with CL

ILE C++ declaration in prototype	CL	Length	Comments
struct or class	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Passing Arguments from a CL Program to an ILE C++ Program

Table 16 shows how arguments are passed from a command line CL call to an ILE C++ program.

Table 16. Arguments Passed From a Command Line CL Call to an ILE C++ Program

Command Line Argument	Argv Array	ILE C++ Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	__D("0000000123.40000")
'Hi'	argv[3]	"Hi"
Lo	argv[4]	"LO"
'1'	argv[5]	"1"

A CL character array is not null-terminated when it is passed to another program. A C++ program that receives such an argument from a CL program should not expect the strings to be null-terminated. You can use the `QCAPEXC` to ensure that all the arguments are null-terminated.

Table 17 on page 213 shows how CL constants are passed from a compiled CL program to an ILE C++ program.

Compile CL Program Argument	Argv Array	ILE C++ Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	__D("0000000123.40000")
'Hi'	argv[3]	"Hi"
Lo	argv[4]	"LO"
'1'	argv[5]	"1"

A command processing program (CPP) passes CL constants as defined in Table 17. You can create your own CL command with the Create Command (CRTCMD) command and define an ILE C++ program as the command processing program.

Table 18 shows how CL variables are passed from a compiled CL program to an ILE C++ program. All arguments are passed by reference from CL to C++.

CL Variables	C++ Arguments
DCL VAR(&v) TYPE(*CHAR) LEN(10) VALUE('123.4')	"123.4"
DCL VAR(&d) TYPE(*DEC) LEN(10 1) VALUE(123.4)	__D("0000000123.40000")
DCL VAR(&h) TYPE(*CHAR) LEN(10) VALUE('Hi')	"Hi"
DCL VAR(&i) TYPE(*CHAR) LEN(10) VALUE(Lo)	"LO"
DCL VAR(&j) TYPE(*LGL) LEN(1) VALUE('1')	"1"

CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings. Character literals and logical literals are passed as null-terminated strings but are not padded with blanks. Numeric literals such as packed decimals are passed as 15,5 (8 bytes).

The CL program CLPROG1 passes the parameters v, d, h, i, j to an ILE C++ program MYPROG1.

The parameters are null-terminated within the the CL program CLPROG1. They are passed by reference. All incoming arguments to MYPROG1 are pointers.

```

/* CLPROG1
PGM      PARM(&V &D &H &I &J)
  DCL      VAR(&V) TYPE(*CHAR) LEN(10)
  DCL      VAR(&VOUT) TYPE(*CHAR) LEN(11)
  DCL      VAR(&D) TYPE(*DEC) LEN(10 1)
  DCL      VAR(&H) TYPE(*CHAR) LEN(10)
  DCL      VAR(&HOUT) TYPE(*CHAR) LEN(11)
  DCL      VAR(&I) TYPE(*CHAR) LEN(10)
  DCL      VAR(&IOUT) TYPE(*CHAR) LEN(11)
  DCL      VAR(&J) TYPE(*LGL) LEN(1)
  DCL      VAR(&JOUT) TYPE(*LGL) LEN(2)
  DCL      VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE C++ PROGRAM */
CHGVAR    VAR(&VOUT) VALUE(&V *TCAT &NULL)
CHGVAR    VAR(&HOUT) VALUE(&V *TCAT &NULL)
CHGVAR    VAR(&IOUT) VALUE(&V *TCAT &NULL)
CHGVAR    VAR(&JOUT) VALUE(&V *TCAT &NULL)
CALL      PGM(MYPROG1) PARM(&VOUT &D &HOUT &IOUT &JOUT)
ENDPGM

```

The CL program CLPROG1 receives its input values from a CL Command Prompt MYCMD1 which prompts the user to input the desired values. The source code for MYCMD1 is:

```

CMD      PROMPT('ENTER VALUES')
  PARM    KWD(V) TYPE(*CHAR) LEN(10) +
          PROMPT('1ST VALUE')
  PARM    KWD(D) TYPE(*DEC) LEN(10 2) +
          PROMPT('2ND VALUE')
  PARM    KWD(H) TYPE(*CHAR) LEN(10) +
          PROMPT('3RD VALUE')
  PARM    KWD(I) TYPE(*CHAR) LEN(1) +
          PROMPT('4TH VALUE')
  PARM    KWD(J) TYPE(*LGL) LEN(10 2) +
          PROMPT('5TH VALUE')

```

After the CL program CLPROG1 has received the user input from the command prompt MYCMD1, it passes the input values on to a C++ program MYPROG1. The source code for this program is contained in myprog1.cpp:

```

// myprog1.cpp

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// Arguments are received by reference from CL program CLPROG1
// Incoming arguments are all pointers

int main(int argc, char *argv[])
{
    char          *v;
    char          *h;
    char          *i;
    char          *j;
    _DecimalT <10, 1> d;

    v = argv[1];
    d = *((_DecimalT <10,1> *) argv[2]);
    h = argv[3];
    i = argv[4];
    j = argv[5];
    cout << " v= " << v
         << " d= " << d
         << " h= " << h
         << " i= " << i
         << " j= " << j
         << endl;
}

```

If the CL program CLPROG1 passes the following parameters to the C++ program MYPROG1:

'123.4', 123.4, 'Hi', LO, and '1'

the output from program MYPROG1 is:

```

v= 123.4    HI        LO        1 d= 123.4 h= HI        LO        1
i= LO      1 j= 1
Press ENTER to end terminal session.

```

Changing the Names of Programs and Procedures

You may want to change the name of an ILE procedure to make it more descriptive, and to make the ILE procedure easy to identify for maintenance purposes. An ILE procedure name QRZ1233 could be renamed to `checkmod`. You may want to change the name of a program that contains an illegal character; A-B is not a valid name in a C++ program.

You can use the **#pragma map** directive to map an internal identifier to an OS/400-compliant name (10 characters or less for program names and 1 or more characters for ILE procedure names) in your program.

The syntax is:

```
# pragma map (identifier , "name")

# pragma map
(function-or-operator-identifier(argument-list) , "name")
```

This pragma tells the compiler that all references to identifier are to be converted to "name". The pragma can appear anywhere in the source file 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:

```
int func(int);

class X
{
public:
    void func(void);
#pragma map(func, "funcname1") //maps ::func
#pragma map(X::func, "funcname2") //maps X::func
};
```

There are two functions named `func()` in this code. One is a regular function, prototyped `int func(int)` and the other is a class member function `void func(void)`. To avoid confusion, they are renamed `funcname1`, and `funcname2` using the **#pragma map** directive.

Creating C++ Classes for Use in ILE

You can access existing C++ classes from other languages such as ILE C, but you need to write your own functions to display and manipulate the data members of these classes.

Mapping a C++ Class to a C Structure

A C++ class without virtual functions can be mapped to a corresponding C structure, but there are fundamental differences between both data types. The C++ class contains data members and member functions to access and manipulate these data members. The corresponding C structure contains only the data members, but not the member functions contained in the C++ class.

The class `Class1` in C++ is:

```

Class Class1
{
public:
    int m1;
    int m2;
    int m3;
    f1();
    f2();
    f3();
};

```

The corresponding C structure is:

```

struct Class1
{
    int m1;
    int m2;
    int m3;
};

```

To access a C++ class from a C program you need to write your own functions to inspect and manipulate the class data members directly.

Note: While data members in the C++ class can be **public**, **protected**, or **private**, the variables in the corresponding C structure are always publicly accessible. Be careful, you may eliminate the safeguards built into the C++ language.

You can use C++ operators on this class if you supply your own definitions of these operators in the form of member functions.

When you write your own C++ classes that you want to access from other languages:

Do not use static data members in your class, because they are not part of the C++ object that is passed to the other language.

Do not use virtual functions in your class, because you cannot access the data members since the alignment of the data members between the class and the C structure is different.

By making all data members of a class publicly accessible to programs written in other languages, you may be breaking data encapsulation.

Using C++ Objects in a C Program

This program shows how you can access the data members in C++ classes from source code written in C.

Program Structure

The program consists of these files:

A C++ source file `hourclas.cpp` which contains:

- Definitions of one base class, `HourMin`, and two derived classes, `HourMinSec1` and `HourMinSec2`

- Three function prototypes with extern "C" linkage:
 - extern "C" void CSetHour(HourMin *)
 - extern "C" void CSetSec(HourMin *)
 - extern "C" void CSafeSetHour(HourMin *)
 - The definition of a function with extern "C" linkage, extern "C" void CXXSetHour(HourMin * x)
 - A main() function containing the program logic
- A C source file hour.c which contains:
- A structure CHourMin that maps to the C++ class HourMin in file hourclas.cpp
 - Definitions of the three functions with extern "C" linkage declared in hourclas.cpp
 - The definition of a function CSafeSetHour()

Program Description

In its main() function the program:

1. Instantiates an object hm of the base class HourMin
2. Assigns a value to the h variable (hour) in the base class
3. Passes the address of the base class to the function CSetHour() defined in the C source file hour.c, which assigns a new value to h in the base class
4. Displays the value of h in the base class
5. Instantiates an object hms1 of the derived class HourMinSec1
6. Passes the address of this object class to the function CSetSec() defined in the C source file hour.c, which assigns a value to s in the object
7. Displays the value of s in the object
8. Instantiates an object hms2 of the class HourMinSec2 which contains the class HourMin
9. Passes the address of this new object to the function CSetSec() defined in the C source file hour.c, which assigns a value to s in the object
10. Displays the value of s in the object
11. Passes the address of the base class object to function SafeSetHour() defined in the C source file hour() which passes the address back to a function CXXSetHour() defined in the C++ source file hourclas.cpp

The C++ source file hourclas.cpp is:

```
#include <iostream.h>

class HourMin {          // base class
protected:
    int h;
```

```

    int m;

public:
    void set_hour(int hour) { h = hour % 24; } // keep it in range
    int get_hour()         { return h; }
    void set_min(int min)  { m = min % 60; } // keep it in range
    int get_min()         { return m; }
    HourMin(): h(0), m(0) {}
    void display() { cout << h << ':' << m << endl; }
};

// derived from class HourMin
class HourMinSec1 : public HourMin {
public:
    int s;
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int get_sec()         { return s; }
    HourMinSec1() { s = 0; }
    void display() { cout << h << ':' << m << ':' << s << endl; }
};

// has an HourMin contained inside
class HourMinSec2 {
private:
    HourMin a;
    int s;

public:
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int get_sec()         { return s; }
    HourMinSec2()         { s = 0; }
    void display() {
        cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
};

extern "C" void CSetHour(HourMin *); // defined in C
extern "C" void CSetSec(HourMin *); // defined in C
extern "C" void CSafeSetHour(HourMin *); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
    x->set_hour(99); // much like the C version but the C++
                    // member functions provide some protection
                    // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...

main() {

    HourMin hm;

```

```

    hm.set_hour(18); // supper time;
    CSetHour(&hm); // pass address of object to C function
    hm.display(); // hour is out of range

    HourMinSec1 hms1;
    CSetSec((HourMin *) &hms1)
    hms1.display();

    HourMinSec2 hms2;
    CSetSec(&hms2);
    hms2.display();

    CSafeSetHour(&hm); // pass address to a safer C function
    hm.display(); // hour is not out of range
}

```

The C source file hour.c is:

```

/* C code hour.c */

struct CHourMin {
    int hour;
    int min;
};

void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99; // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes

void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want

    // ... do things with p, but be careful
    // ...
    // use a C++ wrapper function to access C++ function members
}

```

```
CXXSetHour(p); // almost the same as p->hour = 99
}
```

Program Output

The program output is:

```
99:0
0 -:0 :45
0 -:0 :45
3 -:0
Press ENTER to end terminal session.
```

Qualifying Library Calls

You can call a program with a library qualification by using the:

- Bindable APIs with library qualification

- `system()` function in C to use `QCAPEXC` and call with library qualification

- AS/400 Resolve System Pointer (RLSYP) to resolve to the object with a library and call with the pointer

Calling OPM Programs

This program demonstrates some typical steps in creating a program that uses several ILE and OPM programming languages.

Program Description

The program is a small transaction-processing program that takes as input the item name, price, and quantity for one or more products. As output, the program displays the total cost of the items specified on the display and writes an audit trail of the transactions to a file.

Figure 23 on page 222 shows the basic flow of the program.

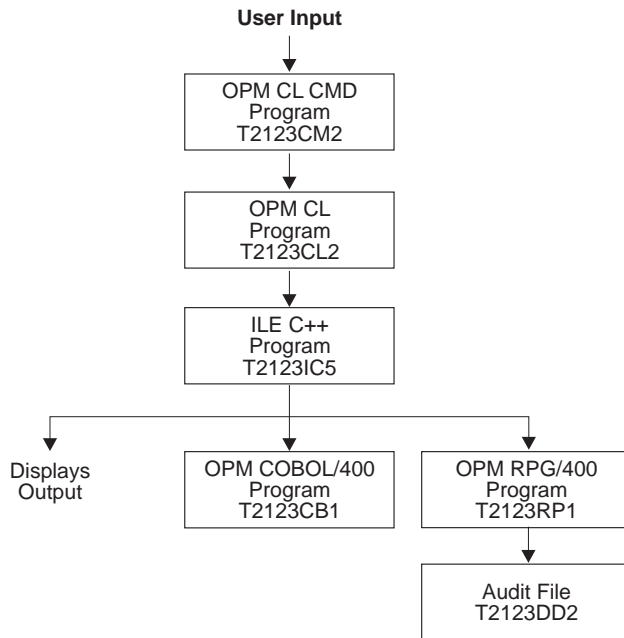


Figure 23. Basic Program Structure

Program Structure

The program consists of these components:

- A CL command T2123CM2 that accepts the users input and passes it to an OPM CL program

- An OPM CL program T2123CL2 that processes the input and passes it to an ILE C++ program

- An ILE C++ program T2123IC5 that calls an OPM COBOL program to process the input, and an OPM RPG program to write the audit trail to an externally described file

- An OPM COBOL program T2123CB1 that completes the calculation and formats the cost

- An OPM RPG program T2123RP1 that updates the audit file with each transaction

- An externally described file T2123DD2 that receives the audit trail

Program Activation

The ILE C++ program T2123IC5 is created with the CRTPGM default for the ACTGRP parameter, ACTGRP(*NEW). When the CL program calls the ILE C++ program, a new activation group is started.

The OPM CL, COBOL, and RPG programs are activated within the OPM default activation group.

Figure 24 shows the structure of this program in ILE.

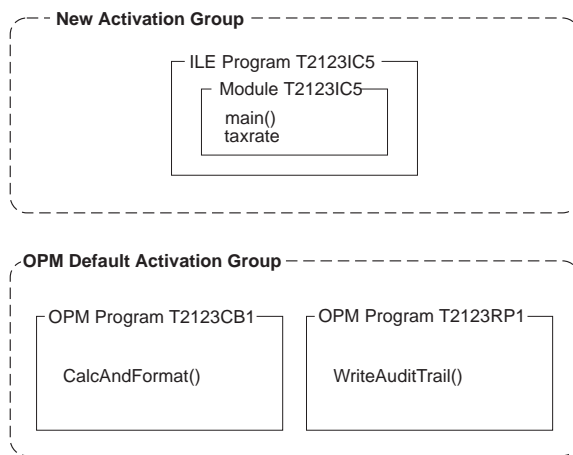


Figure 24. Structure of the Program in ILE C++

Program Files

The source code for each of the files that compose this program are an externally described file, a CL program, a CL command prompt, a C++ source file, and OPM COBOL program and an OPM RPG program.

Externally Described File T2123DD2

The file T2123DD2 contains the audit trail for the C++ program T2123IC5. The DDS source defines the fields for the audit file:

```

R T2123DD2R
  USER          10          COLHDG('User')
  ITEM           20          COLHDG('Item name')
  PRICE         10S 2        COLHDG('Unit price')
  QTY            4S          COLHDG('Number of items')
  TXRATE        2S 2        COLHDG('Current tax rate')
  TOTAL         21          COLHDG('Total cost')
  DATE           6          COLHDG('Transaction date')
K USER
  
```

CL Program T2123CL2

The CL program T2123CL2 passes the CL variables `item_name`, `price`, `quantity` and `user_id` by reference to an ILE C++ program T2123IC5.


```

PGM          PARM(&ITEMIN &PRICE &QUANTITY)
DCL          VAR(&USER) TYPE(*CHAR) LEN(10)
DCL          VAR(&USEROUT) TYPE(*CHAR) LEN(11)
DCL          VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL          VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL          VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL          VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
CHGVAR      VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
CHGVAR      VAR(&USEROUT) VALUE(&USER *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE */
RTVJOBA     USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF      FILE(T2123DD2) TOFILE(*LIBL/T2123DD2) +
            MBR(T2123DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL        PGM(T2123IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
            &USEROUT)
DLTOVR      FILE(*ALL)
ENDPGM

```

The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C++ program. The variables containing the user and item names are explicitly null-terminated in the CL program.

Note: CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings. Character literals and logical literals are passed as null-terminated strings but are not widened with blanks. Numeric literals such as packed decimals are passed as 15,5 (8 bytes). Floating point constants are passed as double precision floating point values (1.2E+15).

CL Command Prompt T2123CM2

You use the CL command prompt T2123CM2 to prompt the user to enter item names, prices, and quantities that will be used by the C++ program T2123IC5.

```

CMD          PROMPT('CALCULATE TOTAL COST')
PARM         KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
            MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM         KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
            RANGE(0.01 99999999.99) MIN(1) +
            ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM         KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
            9999) MIN(1) ALWUNPRT(*YES) +
            PROMPT('Number of items' 3)

```

C++ Source File T2123IC5

The C++ source file T2123IC5 contains a main() function which receives the incoming arguments from the CL program T2123CL2. These arguments have been verified by the CL command prompt T2123CM2 and null-terminated within the CL program T2123CL2. All the incoming arguments are pointers.

The `main()` function calls the function `CalcAndFormat()` which is mapped to a COBOL name. It passes the `price`, `quantity`, `taxrate`, `formatted_cost`, and a `success_flag` to the OPM COBOL program `T2123CB1` using the extern "OS nowiden" linkage specification, because the OPM COBOL program is not expecting widened parameters.

The `formatted_cost` and the `success_flag` values are updated in the C++ program `T2123IC5`.

If `CalcAndFormat()` returns successfully a record is written to the audit trail by `WriteAuditTrail()` in the OPM RPG program.

The `main()` function in program `T2123IC5` calls `WriteAuditTrail()` which is mapped to an RPG program name, and passes the `user_id`, `item_name`, `price`, `quantity`, `taxrate`, and `formatted_cost`, using the extern "OS" linkage specification.

Note: By default, the compiler converts a short integer to an integer unless the `nowiden` parameter is specified on the extern linkage specification. The short integer in the C++ program is converted to an integer, and then passed to the OPM RPG program. The RPG program is expecting a 4 byte integer for the quantity variable. See "Understanding Data-Type Compatibility" on page 204 for information on data-type compatibility.

```
// This program is called by a CL program that passes an item
// name, price, quantity and user ID.
// COBOL is called to calculate and format the total cost.
// RPG is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a new program name to the existing
// program name so that the purpose of the program is clear.
// Tell the compiler that there are dynamic program calls so
// arguments are passed by value-reference.

extern "OS nowiden" void CalcAndFormat(_DecimalT <10,2>,
                                     short int, _DecimalT<2,2>, char[],
                                     char *);

#pragma map(CalcAndFormat,"T2123CB1")

extern "OS" void WriteAuditTrail(char[], char[] ,
                                 _DecimalT<10,2>, short int,
                                 _DecimalT<2,2>, char[]);

#pragma map(WriteAuditTrail,"T2123RP1")

int main(int argc, char *argv[])
{
    // Incoming arguments from a CL program have been verified by
```

```

// the *CMD and null-terminated within the CL program.
// Incoming arguments are passed by reference from a CL program.

char          *user_id;
char          *item_name;
short int     quantity;
_DecimalT <10, 2> price;
_DecimalT <2,2> taxrate = __D(".15");
char          formatted_cost[22];

// Remove null terminator for RPG program. Item name is null
// terminated for C++.

char          rpg_item_name[20];
char          null_formatted_cost[22];
char          success_flag = 'N';
int           i;

// Incoming arguments are all pointers.

item_name =          argv[1];
price      = *((_DecimalT<10, 2> *) argv[2]);
quantity   = *((short *)          argv[3]);
user_id    =          argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

CalcAndFormat(price, quantity, taxrate, formatted_cost,
               &success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

// Null terminate the result.

formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {

// Remove null terminator for the RPG program.

        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = ' ';
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }
}

```

```

// Call an RPG program to write audit records.

    WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                    taxrate, formatted_cost);

    cout <<quantity <<item_name << "plus tax ="
         <<null_formatted_cost <<endl;
}
else
{
    cout <<"Calculation failed" <<endl;
}
}

```

OPM COBOL Program T2123CB1

The OPM COBOL program T2123CB1 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag.

The CalcAndFormat() function in program T2123CB1 calculates and formats the total cost. Parameters are passed from the ILE C++ program to the OPM COBOL program to do the tax calculation.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T2123CB1.
*****
* parameters:
* incoming: PRICE, QUANTITY
* returns : TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
*          SUCCESS-FLAG.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. AS-400.
OBJECT-COMPUTER. AS-400.
DATA DIVISION.
WORKING-STORAGE SECTION.

    01 WS-TOTAL-COST          PIC S9(13)V99      COMP-3.
    01 WS-TAXRATE            PIC S9V99          COMP-3.

LINKAGE SECTION.

    01 LS-PRICE              PIC S9(8)V9(2)      COMP-3.
    01 LS-QUANTITY           PIC S9(4)          COMP-4.
    01 LS-TAXRATE            PIC SV99          COMP-3.
    01 LS-TOTAL-COST         PIC $$$,$$$,$$$,$$$,$$$,$$$.$99
                                DISPLAY.
    01 LS-OPERATION-SUCCESSFUL PIC X          DISPLAY.

PROCEDURE DIVISION USING LS-PRICE
                    LS-QUANTITY

```

```

LS-TAXRATE
LS-TOTAL-COST
LS-OPERATION-SUCCESSFUL.

```

```

MAINLINE.

```

```

  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.

```

```

  CALCULATE-COST.

```

```

    MOVE LS-TAXRATE TO WS-TAXRATE.
    ADD 1 TO WS-TAXRATE.
    COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                         LS-PRICE *
                                         WS-TAXRATE

```

```

    ON SIZE ERROR

```

```

      MOVE "N" TO LS-OPERATION-SUCCESSFUL
    END-COMPUTE.

```

```

  FORMAT-COST.

```

```

    MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

OPM RPG Program T2123RP1

The OPM RPG program T2123RP1 contains the `WriteAuditTrail()` function which writes the audit trail for the program.

```

FT2123DD20  E          DISK          A
F          T2123DD2R          KRENAMEDD2R
IQTYIN      DS
I          B    1    40QTYBIN
C          *ENTRY    PLIST
C          PARM      USER    10
C          PARM      ITEM    20
C          PARM      PRICE   102
C          PARM      QTYIN
C          PARM      TXRATE   22
C          PARM      TOTAL   21
C          EXSR ADDR EC
C          SETON
C          ADDR EC    BEGSR          LR
C          MOVE LUDATE    DATE
C          MOVE QTYBIN    QTY
C          WRITEDD2R
C          ENDSR

```

Invoking the ILE-OPM Program

To enter data for the program T2123IC5 enter the AS/400 command T2123CM2 and press F4 (Prompt).

You can enter this data into T2123CM2:

```
Hammers
1.98
5000
Nails
0.25
2000
```

The output is:

```
5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.
```

The physical file T2123DD2 contains this data:

SMITHE	HAMMERS	0000000198500015	\$11,385.0007	2893
SMITHE	NAILS	0000000025200015	\$575.0007	2893

Calling ILE Programs

This program shows you some typical steps in creating a program that uses several ILE programming languages.

Program Description

This program is an ILE version of the small transaction-processing program described in the “Calling OPM Programs” on page 221.

Program Structure

The program consists of these components:

- A CL command T2123CM3 that accepts the user input and passes it to an ILE CL program

- An ILE CL program T2123CL3 that processes the input and passes it to an ILE program

- An ILE program T2123ICB in which the `main()` function of a C++ module T2123ICB calls a procedure `CalcAndFormat` in an ILE COBOL module T2123CB2

- A service program T2123SP3, created from a C++ source file `t2123icc.cpp`, that exports the variable `TAXRATE`

- A service program T2123SP4, created from an ILE RPG module object T2123RP2, that writes an audit trail of all transactions to a file

- An externally described file T2123DD2 that receives the audit trail data

Figure 25 on page 230 shows the ILE structure.

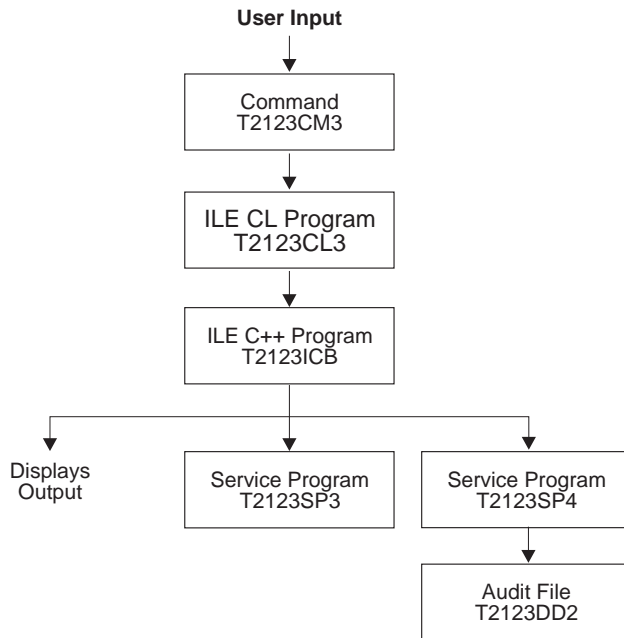


Figure 25. ILE Structure

Program Activation

The programs T2123CL3 and T2123ICB are created with the CRTPGM default for the *ACTGRP* parameter, *ACTGRP(*NEW)*. When the CL program calls the ILE C++ program, a new activation group is started.

The service programs are created with the CRTSRVPGM default for the *ACTGRP* parameter, *ACTGRP(*CALLER)*. When they are called, they are activated within the activation group of the calling program.

Figure 26 on page 231 shows the basic object structure used in this example.

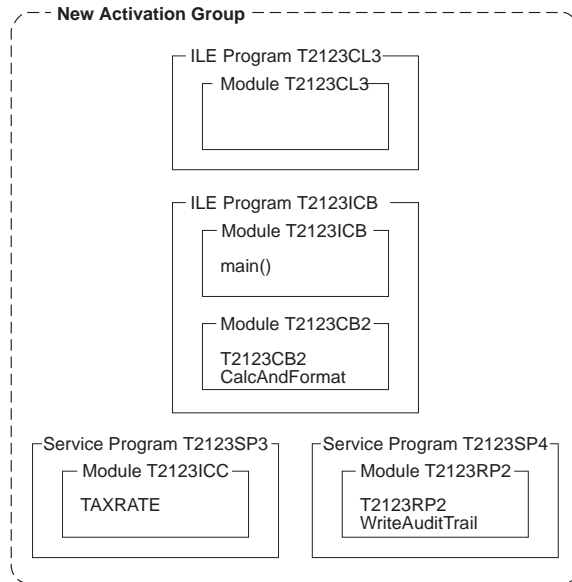


Figure 26. Basic Object Structure

Program Files

The source files for this program include an externally described file, a CL program, a command prompt, two C++ source files, and ILE COBOL source file and an ILE RPG source file.

Externally Described File T2123DD2

The file `T2123DD2` contains the audit trail for the C++ program `T2123ICB`. The DDS source defines the fields for the audit file.

See “Externally Described File T2123DD2” on page 223 for the DDS source of the audit file `T2123DD2`.

CL Program T2123CL3

The CL program `T2123CL3` passes the CL variables `item_name`, `price`, `quantity`, and `user_id` by reference to an ILE C++ program `T2123IC5`.

CL Command Prompt T2123CM3

You use the CL command prompt `T2123CM3` to prompt the user to enter item names, prices, and quantities that will be used by the C++ program `T2123ICB`.

The source code for program `T2123CL3` is identical to the source code shown in “CL Program `T2123CL2`” on page 223.

C++ Source File T2123ICB.CPP

The source for the ILE C++ program T2123ICB is almost identical to the source shown in "C++ Source File T2123IC5" on page 224. The difference lies in the linkage specifications used for interlanguage calls:

```
// This program demonstrates the interlanguage call capability
// of an ILE C++ program. This program is called by a CL
// program that passes an item name, price, quantity and user ID.
// A COBOL procedure is called to calculate and format total
// cost. An RPG procedure is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a function name to the bound
// procedure name so that the purpose of the procedure is clear.
// Tell the compiler that there are bound procedure calls and
// arguments are to be passed by value-reference.

extern "COBOL" void CalcAndFormat(_DecimalT <10,2>,
                                short int, char[],
                                char *);

#pragma map(CalcAndFormat,"T2123CB2")

extern "RPG" void WriteAuditTrail(char[],
                                  char[],
                                  _DecimalT<10,2>,
                                  short int, char[]);

#pragma map(WriteAuditTrail,"T2123RP2")

int main(int argc, char *argv[])
{
// Incoming arguments from a CL program have been verified by
// the *CMD and null-terminated within the CL program.
// Incoming arguments are passed by reference from a CL program.

    char          *user_id;
    char          *item_name;
    short int     quantity;
    _DecimalT<10, 2> price;
    char          formatted_cost[22];

// Remove null terminator for RPG program. Item name is null
// terminated for C++.

    char          rpg_item_name[20];
    char          null_formatted_cost[22];
    char          success_flag = 'N';
```

```

int          i;

//Incoming arguments are all pointers.
    item_name =          argv[1];
    price     = *((_DecimalT<10, 2> *) argv[2]);
    quantity  = *((short *)          argv[3]);
    user_id   =          argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

    CalcAndFormat(price, quantity, formatted_cost, &success_flag);

    memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

// Null terminate the result.

formatted_cost[21] = '\0';
if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
            {

// Remove null terminator for the RPG program.

                if (*(item_name+i) == '\0')
                    {
                        rpg_item_name[i] = ' ';
                    }
                else
                    {
                        rpg_item_name[i] = *(item_name+i);
                    }
            }
    }

// Call an RPG program to write audit records.

WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                formatted_cost);

cout <<"plus tax =" << quantity << item_name << null_formatted_cost
    <<endl <<endl;
    }
else
    {
        cout <<"Calculation failed" <<endl;
    }
}

```

C++ Source File T2123ICC

The source for the ILE C++ module T2123ICC shows the variable TAXRATE is exported from this module to be used by ILE COBOL and ILE RPG procedures.

Note: Weak definitions (EXTERNALs from COBOL) cannot be exported out of a service program to a strong definition language like C or C++, while C or C++ can export to COBOL. The choice of language for TAXRATE is C++.

```
// Export the tax rate data.
#include <bcd.h>
const _DecimalT <2,2> TAXRATE = __D(".15");
```

ILE COBOL Module T2123CB2

The ILE COBOL procedure in T2123CB2 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag.

The CalcAndFormat() function calculates and formats the total cost. Parameters are passed from the ILE C++ program to the ILE COBOL procedure to do the tax calculation.

The source code for T2123CB2 is:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
*****
* parameters:
* incoming: PRICE, QUANTITY
* returns : TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
* SUCCESS-FLAG.
* TAXRATE : An imported value.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. AS-400.
OBJECT-COMPUTER. AS-400.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 WS-TOTAL-COST PIC S9(13)V99 COMP-3.
01 WS-TAXRATE PIC S9V99 COMP-3
VALUE 1.
01 TAXRATE EXTERNAL PIC SV99 COMP-3.

LINKAGE SECTION.
01 LS-PRICE PIC S9(8)V9(2) COMP-3.
01 LS-QUANTITY PIC S9(4) COMP-4.
01 LS-TOTAL-COST PIC $$$,$$$,$$$,$$$,$$$.$99
DISPLAY.
01 LS-OPERATION-SUCCESSFUL PIC X DISPLAY.

PROCEDURE DIVISION USING LS-PRICE
LS-QUANTITY
LS-TOTAL-COST
LS-OPERATION-SUCCESSFUL.
```

```

MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.

CALCULATE-COST.
  ADD TAXRATE TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

  ON SIZE ERROR
    MOVE "N" TO LS-OPERATION-SUCCESSFUL
  END-COMPUTE.

FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

ILE RPG Module T2123RP2

The ILE RPG module T2123RP2 contains the `WriteAuditTrail()` function which writes the audit trail for the program:

```

FT1520DD2  O  A  E          DISK
D  TAXRATE          S          3P 2  IMPORT
D  QTYIN            DS
D  QTYBIN           1          4B 0
C   *ENTRY          PLIST
C                   PARM          USER          10
C                   PARM          ITEM          20
C                   PARM          PRICE         10 2
C                   PARM          QTYIN
C                   PARM          TOTAL         21
C                   EXSR          ADDR          LR
C                   SETON
C   ADDR          BEGSR
C                   MOVEL         UDATE          DATE
C                   MOVE          QTYBIN         QTY
C                   MOVE          TAXRATE        TXRATE
C                   WRITE         T1520DD2R
C                   ENDSR

```

Service Program T2123SP3

Service program T2123SP3 is created from the C++ module T2123ICC. It exports the variable `TAXRATE`.

Service Program T2123SP4

Service program T2123SP4 is created from the ILE RPG module T2123RP2. It exports the procedure T2123RP2.

Invoking the ILE Program

T2123ICB is considered the main program. It runs in the new activation group that is created when the CL program T2123CL3 is called.

To enter data for the program T2123ICB enter the AS/400 command: T2123CM2 and press F4 (Prompt). You can enter the sample data in “Invoking the ILE-OPM Program” on page 228.

The output is the same as for the OPM version of this program.

The physical file T2123DD2 contains the same data as shown in the OPM version in “Invoking the ILE-OPM Program” on page 228.

Calling an ILE C++ Program

This program shows how to retrieve a return value from `main`. A CL command called `SQUARE` calls an ILE C++ program `SQITF`. The program `SQITF` calls another ILE C++ program called `SQ`. The program `SQ` returns a value to program `SQITF`.

Note: Returning an integer value from an ILE C++ program may impact performance.

You use the CL command prompt `SQUARE` to enter the number you want to determine the square of for the ILE C++ program `SQITF`:

```
CMD      PROMPT('CALCULATE THE SQUARE')
PARM     KWD(VALUE) TYPE(*INT4) RSTD(*NO) RANGE(1 +
          9999) MIN(1) ALWUNPRT(*YES) PROMPT('Value' 1)
```

The ILE C++ program calls another ILE C++ program called `SQ`:

```

// This program SQITF is called by the command SQUARE. This
// program then calls another ILE C++ program SQ to perform
// calculations and return a value.

#include <iostream.h>

extern "OS" int SQ(int); // Tell compiler this is external call,
                        // do not pass by value.

int main(int argc, char *argv[])
{
    int *x;
    int result;

    x = (int *) argv[1];

    result = SQ(*x);

    // Note that although the argument is passed by value, the compiler
    // copies the argument to a temporary variable, and the pointer to
    // the temporary variable is passed to the called program SQ.

    cout <<"The SQUARE of" <<x <<"is" <<result <<endl;
}

```

The ILE C++ program `SQ` calculates an integer value and returns the value to the calling program `SQITF`:

```

// This program is called by a ILE C++ program called SQITF.
// It performs the square calculations and returns a value to SQITF.

int main(int argc, char *argv[])

{ return (*(int *) argv[1]) * (*(int *) argv[1]);
}

```

To enter data for the program `SQITF` enter the AS/400 command `SQUARE` and press F4 (Prompt). Type 10, and press Enter. The output is:

```

The SQUARE of 10 is 100
Press ENTER to end terminal session.

```

Calling an EPM C Program

If your ILE C++ program calls an EPM default entry point, use the `extern` linkage specification in your ILE C++ source to tell the compiler that `PGMNAME` is an external program, not a bound ILE procedure. `QPXXCALL` is not needed to call EPM default entry points.

The ILE C++ program T2123DL2 passes two integers and two characters to an EPM C program T2123DL3:

```
// t2123dl2.cpp

#include <string.h>
#include <iostream.h>

extern "OS" void T2123DL3(int *,int *,char *,char *);

int main(void)
{
    int      i;           // Integer parameter to pass to callee.
    int      rtn;        // Return value from main in T2123DL3.
    char     a,b;        // Character parameters to pass.
    i = 5;               // Initialize parameters to be passed.
    a = 'a';
    b = 'b';

    T2123DL3(&rtn,&i,&a,&b);

    cout << "Values returned are rtn = " <<rtn << ", i=" <<i
         << ", a=" <<a << ", b=" <<b <<endl;
}
```

The EPM C program T2123DL3 receives and return values to the ILE C++ program:

```
/* This program illustrates how this EPM C program retrieves */
/* t2123dl3.c */

#include <stdio.h>

int main( int argc, char *argv[])
{
    printf ( "integer passed was %d\n", * (int *) argv[2]);

        * (int *) argv[2] = 8;
    printf ( "character passed was %c\n", *argv[3]);
        *argv[3] = 'D';

    printf ( "integer passed was %d\n", * (int *) argv[4]);

    printf ( "as a character it is %c\n", *argv[4]);

    printf ( "returning %d\n", * (int *) argv[1]);
        * (int *) argv[1] =5;

}
```

The output is:

```
Values returned are rtn = 5, i = 8 a = D b = b
Press ENTER to end terminal session.
```

```
Start of terminal session.
integer passed was 5
character passed was a
integer passed was -2113929216
as a character it is b
returning 0
Press ENTER to end terminal session.
```

Calling ILE-Bindable APIs

The program T2123API uses DSM ILE bindable API calls to create a window and echo whatever is entered. The *System API Reference* contains information on the ILE bindable APIs. The prototypes for the DSM APIs are in the <qsnsess.h> header file. The extern "OS nowiden" return type API(arg_list); is specified for each API where return type is void or whatever type is returned by the API, and arg_list is the list of parameters taken by the API. This ensures any value argument is passed by value indirectly.

```
// This program uses Dynamic Screen Manager API calls to
// create a window and echo whatever is entered. This is an
// example of bound API calls. Note the use of extern linkage
// in the <qsnsess.h> header file. OS, nowiden ensures that a
// pointer to an unwidened copy of the argument is passed to the
// API.
// Use BNDDIR(QSNAPI) on the CRTPGM command to build this
// example.

#include <stddef.h>
#include <string.h>
#include <iostream.h>
#include <qsnapi.h>

#define BOTLINE " Echo lines until:  PF3 - exit"

// DSM Session Descriptor Structure.

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char          buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}
```



```

int main(void)
{
    int i;
    storage_t    storage;

    // Declarators for declaring windows. Types are from the <qsnssess.h>
    // header file.

    Qsn_Inp_Buf_T    input_buffer = 0;
    Q_Bin4           input_buffer_size = 50;
    char             char_buffer[100];
    Q_Bin4           char_buffer_size;

    Qsn_Ssn_T        session1;
    Qsn_Ssn_Desc_T   *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
    Qsn_Win_Desc_T   win_desc;
    Q_Bin4           win_desc_length = sizeof(Qsn_Win_Desc_T);
    char             *botline = BOTLINE;
    Q_Bin4           botline_len = sizeof(BOTLINE) - 1;
    Q_Bin4           sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                                     botline_len;

    Q_Bin4           bytes_read;

    // Initialize Session Descriptor DSM API.

    QsnInzSsnD( sess_desc, sess_desc_length, NULL);

    // Initialize Window Descriptor DSM API.

    QsnInzWinD( &win_desc, win_desc_length, NULL);

    sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
    sess_desc->cmd_key_desc_line_1_len = botline_len;
    memcpy( storage.buffer, botline, botline_len );

    sess_desc->cmd_key_desc_line_2_offset = sizeof(Qsn_Ssn_Desc_T) +
                                     botline_len;

    // Set up the session type.

    sess_desc->EBCDIC_dsp_cc = '1';
    sess_desc->scl_line_dsp = '1';
    sess_desc->num_input_line_rows = 1;
    sess_desc->wrap = '1';

    // Set up the window size.

    win_desc.top_row      = 3;
    win_desc.left_col     = 3;
    win_desc.num_rows     = 13;

```

Passing Operational Descriptors

This program shows you how to use operational descriptors in ILE C++. It shows:

- An operational descriptor for `func1` with a **#pragma descriptor** directive for the function in a header file `<op_desc.h>`

- An ILE C++ program `T2123CP1` that calls `func1` in a C++ module `t2123CP2`

- The ILE C++ source code of module `T2123CP2` that contains an ILE API that is used to get information from the operational descriptor

The source code for the header file `op_desc.h` shows an operational descriptor for `func1` with a **#pragma descriptor** directive for the function:

```
/* op_desc.h */
/* containing function prototype */
extern "C" int func1( char a[5], char b[5],
char *c );
#pragma descriptor(void func1( "", "", "" ))
```

A function `func1()` is declared. The **#pragma descriptor** directive for `func1()` specifies that the compiler must generate string operational descriptors for the three arguments the function takes.

The source code `t2123cp1.cpp` shows that the program `T2123CP1` calls function `func1()`. When the function is called, the compiler generates operational descriptors for the three arguments specified on the call:

```
// t2123cp1.cpp

#include "op_desc.h"

main()
{
    char a[5] = {'s', 't', 'u', 'v', '\0'};
    char *c;

    c = "EFGH";
    func1(a, "ABCD", c);
}
```

The source code `t2123cp2.cpp` for module `T2123CP2` defines function `func1()`. It contains the call to the ILE API that is used to determine the string type, length, and maximum length of the string arguments declared in function `func1()`.

The *System API Programming* contains information about `_FEEDBACK`. The values for `typeCharZ` and `typeCharV2` are found in the ILE API header file `<leod.h>`.

```
// t2123cp2.cpp

#include <string.h>
#include <iostream.h>
#include <leawi.h>
```

```

#include <leod.h>
#include "op_desc.h"

int func1(char a[5], char b[5], char *c)
{
    int      posn      = 1;
    int      datatype;
    int      currlen;
    int      maxlen;
    _FEEDBACK fc;

    char     *string1;

    /* Call to ILE API CEEGSI to determine string type, length*/
    /* and the maximum length.*/

    CEEGSI(&posn, &datatype, &currlen, &maxlen, &fc);

    switch(datatype)
    {
        case typeCharZ:
            string1 = a;
            break;
        case typeCharV2:
            string1 = a + 2;
            break;
    }

    /* Use string1.*/

    if (!memcmp(string1, "stuv", currlen))
        cout <<"First 4 characters are the same."<<endl;
    else
        cout <<"First 4 characters are not the same."<<endl;
    return 0;
}

```

Chapter 13. Using Templates in C++ Programs

Templates may be used in C++ to declare and define classes, functions, and static data members of template classes. The C++ language describes the syntax and meaning of each kind of template. Each compiler determines the mechanism that controls when and how often a template is expanded.

VisualAge for C++ for AS/400 offers several alternative organizations with a range of convenience and compile performance to meet the needs of any program. This information describes those alternatives and the criteria you should use to select which one is right for you.

See the *C++ Language Reference* for a general description of templates.

This section includes:

“How the Compiler Expands Templates” on page 246

“Generating Template Function Definitions” on page 247

“Including Defining Templates” on page 248

Using Template Terms

These terms describe the template constructs in C++:

class template

A template used to generate classes. Classes generated in this way are called *template classes*. A class template describes a family of related classes. It can simply be a declaration, or it can be a definition of a particular class.

function template

A template used to generate functions. Functions generated in this way are called *template functions*. A function template describes a family of related functions. It can be a declaration, or it can be a definition of a particular function.

declaring template

A class template or function template that includes a declaration but does not include a definition. A declaring function template is:

```
template<class A> void foo(A*a);
```

A declaring class template is:

```
template<class T> class C;
```

defining template

A class template or function template declaration that includes a definition. A defining function template is:

```
template<class A> void foo(A*a) {a ->Bar();};
```

A defining class template would look like this:

```
template<class T> class C : public A {public: void boo();};
```

explicit definition

A user-supplied definition that overrides a template. An explicit definition of the `foo` function is:

```
void foo(int *a) {a++;}
```

An explicit definition of a template class is:

```
class C<short> {  
    public: int moo();  
}
```

Instantiation

A defining template defines a whole family of classes or functions. An *instantiation* of a template class or function is a specific class or function that is based on the template.

How the Compiler Expands Templates

You can instantiate templates in three ways:

1. Include defining templates everywhere. See “Including Defining Templates Everywhere” on page 248 for more details.
2. Use VisualAge for C++ for AS/400's automatic facility to ensure that there is a single instantiation of the template. See “Structuring for Automatic Instantiation” on page 249 for more details.
3. Manually structure your code so that there is a single instantiation of the template. See “Manually Structuring for Single Instantiation” on page 253 for more details.

The best way to instantiate templates depends on how the compiler reacts when it encounters templates.

When you use templates in your program, the VisualAge for C++ for AS/400 compiler automatically instantiates each defining template that is:

Referenced in the source code

Visible to the compiler (included as the result of an `#include` statement)

Not explicitly defined by the programmer

If an program consists of several separate compilation units that are compiled separately, a given template may expand in two or more of the compilation units. For templates that define classes, inline functions, or static nonmember functions, this is the desired behavior. These templates need to be defined in each compilation unit where they are used.

For other functions and for static data members, which have external linkage, defining them in more than one compilation unit would normally cause an error when the program is bound. VisualAge for C++ for AS/400 avoids this problem by giving special treatment to template-generated versions of these objects. At bind time, VisualAge for

C++ for AS/400 gathers all template-generated functions and static-member definitions, plus any explicit definitions, and resolves references to them in these ways:

If an explicit definition of the function or static member exists, it is used for all references. All template-generated definitions of that function or static member are discarded.

If no explicit definition exists, one of the template-generated definitions is used for all references. Any other template-generated definitions of that function or static member are discarded.

You may have only one explicit definition of any external linkage template instance.

Generating Template Function Definitions

The class template `Stack` shows you template function generation. `Stack` implements a stack of items. The overloaded operators `<<` and `>>` are used to push items onto the stack and pop items from the stack. Assume that the declaration of the `Stack` class template is contained in the file `stack.h`:

```
template <class Item, int size> class Stack {
public:
    int operator << (Item item); // push operator
    int operator >> (Item& item); // pop operator
    Stack() { top = 0; } // constructor defined inline
private:
    Item stack[size]; // stack of items
    int top; // index to top of stack
};
```

In the template, the constructor function is defined inline. Assume the other functions are defined using separate function templates in the file `stack.c`:

```
template <class Item, int size>
int Stack<Item,size>::operator << (Item item) {
    if (top >= size) return 0;
    stack[top++] = item;
    return 1;
}
template <class Item, int size>
int Stack<Item,size>::operator >> (Item& item)
{
    if (top <= 0) return 0;
    item = stack[--top];
    return 1;
}
```

The constructor has internal linkage because it is defined inline in the class template declaration. In each compilation unit that uses an instance of the `Stack` class, the compiler generates the constructor function body. Each unit uses its own copy of the constructor.

In each compilation unit that includes the file `stack.c`, for any instance of the `Stack` class in that unit, the compiler generates definitions for the following functions (assuming there is no explicit definition):

```
Stack<item,size>::operator<<(item)
Stack<item,size>::operator>>(item&)
```

Given the source file `usrstack.cpp`:

```
#include "stack.h"
#include "stack.c"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}
```

the compiler generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)` because both those functions are used in the program, their defining templates are visible, and no explicit definitions are seen.

Including Defining Templates

There are three methods of including defining templates:

“Including Defining Templates Everywhere”

“Structuring for Automatic Instantiation” on page 249

“Manually Structuring for Single Instantiation” on page 253

Automatic instantiation is the recommended method.

Including Defining Templates Everywhere

The simplest way to instantiate templates is to include the defining template in every compilation unit that uses the template. This alternative has these disadvantages:

If you make even a trivial change to the implementation of a template, you must recompile every compilation unit that uses it.

The compilation process is slower, and the resulting modules are bigger because the templates are expanded in every compilation unit where they are used. The duplicated code for the templates is eliminated during binding, so the executable programs are not larger if you choose to include defining templates everywhere.

To use this method with the `Stack` template, include both `stack.h` and `stack.c` in all compilation units that use an instance of the `Stack` class. The compiler generates definitions for each template function. Each template function may be defined multiple times, increasing the size of the module.

Structuring for Automatic Instantiation

The recommended way to instantiate templates is to structure your program for their automatic instantiation. The advantages of this method are:

It is easy to do

Unlike the method of including defining templates everywhere, you do not get larger modules and slower compile times

Unlike the method of including defining templates everywhere, you do not have to recompile all of the compilation units that use a template if that template implementation is changed

The disadvantages of this method are:

It may not be practical in a team programming environment because the compiler may update source files that are being modified at the same time by somebody else.

The modifications that are made to source files may not be file-system-independent. Header files that are locally available may be included rather than header files that are available on a network.

There are some situations where the compiler cannot determine exactly which header files should be included.

To use this facility:

1. Declare your template functions in a header file using class or function templates, but do not define them. Include the header file in your source code.
2. For each header file, create a *template-implementation* file with the same name as the header and the extension `.c`. Define the template functions in this template-implementation file.

Note: Use the same compiler options to bind your modules that you use to compile them. If you compile with the command: `iccas /C myfile.cpp`, bind with the command:

```
iccas /B"CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/MYFILE) "
```

This is especially important for options that control libraries, linkage, and code compatibility. This does not apply to options that affect only the creation of programs (`/C` and `/Fo`).

For each header file with template functions that need to be defined, the compiler generates a template-include file. The template-include file generates **#include** statements in that file for:

The header file with the template declaration

The corresponding template-implementation file

Any other header files that declare types used in template parameters

Note: If you have other declarations that are used inside templates but are not template parameters, you must place or `#include` them in either the template-implementation file or one of the header files included as a result of the above three steps. Define any classes that are used in template arguments and that are required to generate the template function in the header file. If the class definitions require other header files, include them with the **#include** directive. The class definitions are then available in the template-implementation file when the function definition is compiled. Do not put the definitions of any classes used in template arguments in your source code.

```
foo.h
    template<class T> void foo(T*);
hoo.h
    void hoo(A*);
foo.c
    template<class T> void foo(T* t)
        {t -> goo(); hoo(t);}
other.h
    class A {public: void goo() {} };

main.cpp
    #include "foo.h"
    #include "other.h"
    #include "hoo.h"
    int main() { A a; foo(&a); }
```

This requires the expansion of the `foo(T*)` template with `class A` as the template type parameter. The compiler creates a template-include file `TEMPINC\foo.cpp`. The file contents (simplified below) are:

```
#include "foo.h"           //the template declaration header
#include "other.h"        //file defining template type parameter
#include "foo.c"          //corresponding template implementation

void foo(A*);             //triggers template instantiation
```

This does not compile properly because the header `"hoo.h"` did not satisfy the conditions for inclusion but the header file is required to compile the body of `foo(A*)`. One solution is to move the declaration of `hoo(A*)` into the `"other.h"` header file.

The function definitions in your template-implementation file can be explicit definitions, template definitions, or both. Any explicit definitions are used instead of the definitions generated by the template.

Before it invokes the binder, the compiler compiles the template-include files and generates the necessary template function definitions. Only one definition is generated for each template function.

By default, the compiler stores the template-include files in the `TEMPINC` subdirectory under the current directory. The compiler creates the `TEMPINC` directory if it does not already exist. To redirect the template-include files to another directory, use the `/Ftmdir`

compiler option, where *dir* is the directory to contain the template-include files. You can specify a fully qualified path name or a path name relative to the current directory.

If you specify a different directory for your template-include files, make sure that you specify it consistently for all compilations of your program, including the bind step.

Note: After the compiler creates a template-include file, it may add information to the file as each compilation unit is compiled. The compiler never removes information from the file. If you remove function instantiations or reorganize your program so that the template-include files become obsolete, you may want to delete one or more of these files and recompile your program. If error messages are generated for a file in the `TEMPINC` directory, you must either correct the errors manually or delete the file and recompile. To regenerate all of the template-include files, delete the `TEMPINC` directory, the modules, and recompile your program.

If you do not delete the modules, MAKEFILE rules prevent the modules from being recompiled, and the template-include files cannot be updated with all the lines needed for all the compilation units used in the program. The end result is that the bind fails.

A Template-Implementation File

In the `Stack` source, the file `stack.c` is a template-implementation file. To create a program using the `Stack` class template, `stack.h` and `stack.c` must reside in the same directory. You include `stack.h` in any source files that use an instance of the class. The `stack.c` file does not need to be included in any source files. Given the source file:

```
#include "stack.h"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}
```

the compiler automatically generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)`.

You can change the name of the template-implementation file or place it in a different directory using the **#pragma implementation** directive.

The syntax is:

```
#pragma implementation "path"
```

The *path* is used to specify the path name for the template-implementation file. If it is only a partial path name, it must be relative to the directory containing the header file.

Note: This path is a quoted string following the normal conventions for writing string literals. Backslashes must be doubled.

In the `Stack` class, to use the file `stack.def` as the template-implementation file instead of `stack.c`, add the line: `#pragma implementation("stack.def")` anywhere in the `stack.h` file. The compiler then looks for the template-implementation file `stack.def` in the same directory as `stack.h`.

A Template-Include File

A typical template-include file generated by the compiler shows this information:

```

/*0000000000*/ #pragma sourcedir("c:\swearsee\src")      0
/*0698421265*/ #include "c:\swearsee\src\list.h"        1
/*0000000000*/ #include "c:\swearsee\src\list.c"        2
/*0698414046*/ #include "c:\swearsee\src\mytype.h"      3
/*0698414046*/ #include "c:\IBMCPP\INCLUDE\iostream.h"  4
#pragma define(List<MyType>)                             5
ostream& operator<<(ostream&,List<MyType>);             6
#pragma undeclared                                       7
int count(List<MyType>);                                 8

```

- 0 This pragma ensures that the compiler looks for nested include files in the directory containing the original source file, as required by the VisualAge for C++ for AS/400 file inclusion rules.
- 1 The header file that corresponds to the template-include file. The number in comments at the start of each `#include` line (for this line `/*0698421265*/`) is a time stamp for the included file. The compiler uses this number to determine if the template-include file is current or should be recompiled. A time stamp containing only zeroes (0) as in line 2 means the compiler is to ignore the time stamp.
- 2 The template-implementation file that corresponds to the header file in line 1 .
- 3 Another header file that the compiler requires to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point.
- 4 Another header file required by the compiler. It is referenced in the function declaration in line 6 .
- 5 The compiler inserts **#pragma define** directives to force the definition of template classes. In this case, the class `List<MyType>` is to be defined and its member functions are to be generated.
- 6 The `operator<<` function is a nonmember function that matched a template declaration in the `list.h` header file. The compiler inserts this declaration to force the generation of the function definition.
- 7 The **#pragma undeclared** directive is used only by the compiler and only in template-include files. All template functions that are explicitly declared in at least one compilation unit appear before this line. All template functions that are called, but never declared, appear after this line. This division is necessary because the C++ rules for function overload resolution treat declared and undeclared template functions differently.

8 `count` is a template function that is called but not declared. The template declaration of the function is contained in `list.h`, but the instance `count(List<MyType>)` is never declared.

Note: Although you can edit the template-include files, it is not normally necessary or advisable to do so.

Manually Structuring for Single Instantiation

If you do not want to use the automatic instantiation method of generating template function definitions, you can structure your program in such a way that you define template functions directly in your compilation units. The advantage of this approach is that modules are smaller and compile times are shorter than they are when you include defining templates everywhere. When you structure your code manually for template instantiation, you avoid the potential problems that automatic instantiation can cause, such as dependency on a particular file system or file sharing problems.

There are disadvantages to structuring your code manually for template instantiation:

You have to do more work than for the other two methods. You may have to reorganize source files and create new compilation units.

You have to be aware of all of the instantiations of templates that are required by the entire program.

Note: It is recommended that you use the compiler's automatic instantiation facility. The manual structuring method is useful if you find you cannot work around the limitations of the automatic instantiation method.

Use **#pragma define** directives to force the compiler to generate the necessary definitions for all template classes used in other compilation units. Use explicit declarations of non-member template functions to force the compiler to generate them.

To use the second method, include `stack.h` in all compilation units that use an instance of the `Stack` class, but include `stack.c` in only one of the files. If you know what instances of the `Stack` class are being used in your program, you can define all of the instances in a single compilation unit:

```
#include "stack.h"
#include "stack.c"
#include "myclass.h" // Definition of "myClass" class
#pragma define(Stack<int,20>)
#pragma define(Stack<myClass,100>)
```

The **#pragma define** directive forces the definition of two instances of the `Stack` class without creating any object of the class. Because these instances reference the member functions of that class, template function definitions are generated for those functions. See the *C++ Language Reference* for information about the `pragma` directive.

You can compile and bind in one step or two, but you must use `iccas` to invoke the binder. To compile and bind `usrstack.cpp` use the command: `iccas usrstack.cpp` or the commands:

```
iccas /C usrstack.cpp  
iccas /B"CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/USRSTACK) "
```

When you use these methods, you may also need to specify the `/Ft-` option to ensure that the compiler does not also automatically create the `TEMPINC` files according to the automatic generation facility.

Chapter 14. Handling Exceptions

Exception handling is used to detect and report run-time errors and abnormal conditions. There are two types of exceptions in C++, system generated exceptions and C++ exceptions. System generated exceptions can be monitored for using direct monitor handlers, condition handlers, and signal handlers. C++ exceptions can be monitored for and handled using `try-catch-throw`.

This section explains how you can handle exceptions in C++ programs running in the ILE environment on AS/400 system, using one or several of these methods:

High-level language (HLL) exception-handling constructs:

- C++ exception handling using `try-catch-throw`
- C signal handling with `signal()`, `raise()`, and `_GetExcData()`.

Direct monitor handlers that are enabled with the **#pragma exception_handler** directive

ILE condition handlers using the ILE condition handler API `CEEHDLR`

Cancel handlers that are enabled with the **#pragma cancel_handler** directive

Note: Cancel handler functions that are enabled using **#pragma cancel_handler** are called when an activation group is brought down following an abnormal termination, such as an unhandled exception or calling `abort()`. Cancel handlers allow you to have a consistent mechanism for catching exceptions across ILE languages, but they are not invoked just for exception situations.

Introducing Exception Handling

The purpose of system exception handling is to detect and report run-time errors and abnormal conditions. It allows you to build robust programs and reduce the negative impact that unhandled exceptions may have on the overall AS/400 performance.

C++ exception handling provides a way of transferring control and information from a point in the execution of the program to an exception handler. An exception handler must be invoked by a throw-expression invoked in code executed in the handler's try block, or in functions called from the handler's try block.

Exception handling using direct monitor handlers, ILE condition handlers, signal handlers, and cancel handlers helps you:

Examine an exception message that has been issued as a result of a run-time error

Modify the exception to show that it has been received (handled, percolated, or promoted)

Recover from the exception by passing the exception information to a piece of code to take any necessary actions

Table 19 on page 256 summarizes the exception-handling methods, and shows which methods to use and when.

<i>Table 19. Summary of Exception-Handling Methods</i>			
Mechanism	Description	When Called	Why Used
Try-Catch-Throw	C++ exception handling	Not applicable	Localized, C++ only Object friendly Communicate failures among programs Does not trap any AS/400 system-generated exceptions
C Signal Function	C exception handling	Called after direct monitor handler and condition handler were called and did not handle the exception	Global to an activation group Use if performance is not a concern Monitors AS/400 system exceptions
#pragma exception_handler	Direct Monitor Handlers	Called first when an exception is generated	No run-time penalty Use for nonstandard, nonportable code Monitors AS/400 system exceptions
ILE Bindable APIs	ILE Condition Handlers	Called next after direct monitor handlers if the exception is still active at this time. Only for the ILE portion of the code	For a consistent common execution environment across LE enabled platforms Monitors AS/400 system unhandled exceptions that were mapped to signals

Introducing ILE Message Handling

Exception handling is integrated with message handling. As each job starts, it is assigned a Job Message Queue. This object is a message space where all the messages that relate to that job are sent. These messages are linked together by AS/400 in several logical chains, each chain being a different kind of queue:

A *Call Message Queue* is associated with each stack entry. When a new entry appears in the program stack, a new call message queue is created. This queue is destroyed when the procedure or the program leaves the invocation stack.

An *External Queue (*EXT)* is a message queue associated with the job. It has global scope and exists for the lifetime of the job. It is used for inter-process communication, and for communicating with a workstation operator.

A *Log Queue* is a chronological ordering of all messages sent to the job (either from within the job itself, or from one job to another) that are to appear in the job log.

The *call message queue* is the most relevant to exception handling. It has the same name as the program or procedure in the call stack entry. In case of a recursive procedure call, more than one call message queue can be assigned to a single procedure. If you refer to the call message queue by name, you refer to the nearest call stack entry instance of the related procedure.

Messages are sent to a call message queue either by AS/400 or by a user procedure. Your programs may use these ways to interact with call message queues:

Message handler APIs (QMH)

Bindable APIs (CEE)

Note: Message Handler APIs and Bindable APIs are described in the *System API Reference*.

Actions Taken when a Run-time Error Occurs

When a run-time error occurs, an exception message is generated. These are the only types of messages considered to be exceptions:

- *ESCAPE Indicates that a severe error has been detected.
- *STATUS Describes the status of work being done by a program.
- *NOTIFY Describes a condition that requires corrective action or reply a from the calling program.

Function Check Describes a terminating condition.

All of these exception message types can be monitored for using direct monitor handlers, and ILE condition handlers. You cannot monitor these exception message types using signal handlers or `try-catch-throw`.

If an *ESCAPE message is not handled by the direct monitor handler or by the condition handler, then the *ESCAPE message is mapped to the appropriate signal, and a signal handler set up for this signal can trap it and handle it at this point.

When system exceptions are generated during record I/O operations, `errno` may also be set. See Table 28 on page 323 for `errno` macro to exception-handling mapping.

Exception messages are associated with call stack entries. Each call stack entry is in turn associated with a list of exception handlers defined for that entry. When an exception occurs, the handlers associated with the call stack entry have the chance to handle the exception.

If a call stack entry is a control boundary, and the exception is not handled by any handlers in the list, it is considered unhandled, and a default action is taken as shown in Table 20 on page 258. If the call stack entry is not a control boundary, and the exception is not handled by any handlers in the list, then it is moved (percolated) to the caller's call message queue.

Table 20. Default Responses to Unhandled Exceptions

Message Type	Severity of Condition	Condition Raised by the Signal a Condition (CEESGL) Bindable API	Exception Originated from Any Other Source
Status	0 (Informative message)	Return the unhandled condition.	Resume without logging the message.
Status	1 (Warning)	Return the unhandled condition.	Resume without logging the message. For ILE COBOL procedures, resume and issue the message. For non-ILE COBOL procedures, resume without issuing message.
Notify	0 (Informative message)	Not applicable.	Log the notify message and send the default reply.
Notify	1 (Warning)	Not applicable.	Log the notify message and send the default reply. For ILE COBOL procedures, resume and issue the message. For non-ILE COBOL procedures, resume without issuing message.
Escape	2 (Error)	Return the unhandled condition.	Log the escape message and send a function check message to the call stack entry of the current resume point.
Escape	3 (Severe error)	Return the unhandled condition.	Not applicable.
Escape	4 (Critical ILE error)	Log the escape message and send a function check message to the call stack entry of the current resume point.	Log the escape message and send a function check message to the call stack entry of the current resume point.
Function check	4 (Critical ILE error)	Not applicable.	End the program, and send the CEE9901 message across the control boundary, to the caller of the program or procedure that generated the exception.

Note: When the program is ended by an unhandled function check, the activation group is deleted if the control boundary is the oldest call stack entry in the activation group.

Percolation is the action of moving an exception to its caller's call stack entry. Percolation allows a previous call stack entry to handle an exception. Percolation continues until the exception is handled, or until the control boundary is reached. A *control*

boundary is a call stack entry whose caller either runs in a different activation group, or whose caller is an original program model (OPM) program.

The default action depends on the exception type. An exception message is associated with the *procedure* that is active on the call stack. When the exception is percolated, it is not converted to a function check. Each call stack entry is given a chance to handle the original exception until the control boundary is reached. Only then is the exception converted to a function check. This is the default action for an unhandled *ESCAPE message.

At this point, the exception processing starts all over again, beginning with the procedure which received the exception. This time, each call stack entry is given a chance to handle the function check.

If the control boundary is reached and the exception is still unhandled, a generic failure message CEE9901 is sent across the control boundary to the caller of the program or procedure where the exception originated. Any call stack entry that did not handle the message is removed.

Note: If the control boundary call stack entry is the oldest call stack entry in the activation group, then the activation group is terminated for an unhandled function check. See Table 20 on page 258. The destructors for the static objects are called.

Only the system can send a function-check exception message. When this happens the call stack entry is removed, and ILE clears all the automatic storage associated with this entry. The destructors for the automatic objects are called.

ILE allows the percolation of a function-check message. From a diagnostic point of view, percolation is not recorded in the job log; everything is managed by the system. When an exception is handled, no further messages appear anywhere; if you see an exception message not immediately followed by a function check, the exception was handled.

An unhandled exception might go through a very long and costly path for the system; this is particularly true when a function check is generated and percolated.

ILE allows you to move the resume point in your program. Once the exception has been handled, the program flow starts from the resume point you have set. To create a robust exception handler, it is essential to set a proper resume point. If the handler routine returns after coping with the exception, you do not know whether the machine interface (MI) instruction that generated the exception was terminated. Your program might execute several low-level (IMPI) instructions related to the MI instruction that just failed.

The *resume cursor* is a point at which a program resumes after handling an exception. This is initially set to the instruction following the suspend point of the call stack entry that caused the exception.

Note: Many of the examples refer to the `main()` function as a control boundary. The program entry procedure (PEP) is the control boundary for the program, if the program is running in a `*NEW` activation group.

Nesting Exceptions

Exceptions can be nested. A nested exception is an exception that occurs while another exception is being handled. When this happens, the processing of the first exception is temporarily suspended. AS/400 saves all of the associated information such as the locations of the handle cursor and resume cursor. Exception handling begins again with the most recently generated exception. New locations for the handle cursor and resume cursor are set by AS/400. Once the new exception has been properly handled, handling activities for the original exception normally resume.

The *handle cursor* is a point indicating the location of the current exception handler. As the system searches for an available exception handler, it moves the handle cursor to the next handler in the exception handler list. The list may contain direct monitor handlers, ILE condition handlers, and high level language (HLL)-specific handlers.

Note: If a nested exception causes termination, the exception handler for the first exception may not complete.

When a nested exception occurs, both of the following are still on the call stack:

- The call stack entry associated with the original exception
- The call stack entry associated with the original exception handler

To reduce the possibility of exception-handling loops, AS/400 stops the percolation of a nested `*ESCAPE` exception at the original exception-handler stack entry. The call stack entry of an exception handler acts like a control boundary with regard to exception percolation. If you do not handle the nested `*ESCAPE` exception or the function check message, AS/400 ends the program by calling the Abnormal End (`CEE4ABN`) bindable API. Message `CEE9901` is sent across the control boundary to the caller of the program or procedure where the exception originated.

If you move the resume cursor while processing the nested exception, you can implicitly modify the original exception. To cause this to occur:

1. Move the resume cursor to a call stack entry earlier than the call stack entry that incurred the original exception
2. Resume processing by returning from your handler.

This code shows a nested `*ESCAPE` exception. `main()` generates an exception, which causes `main_hdlr()` to get control. The handler `main_hdlr()` generates another exception, which causes `hdlr_hdlr()` to get control. The handler `hdlr_hdlr()` handles the exception. Control resumes in `main_hdlr()`, and it handles the original exception:

```

#include <signal.h>

void hdlr_hdlr(INTRPT_Hndlr_Parms_T *parms)
{
    // Handle the exception, generated in main_hdlr, using QMHCHGEM.
}

void main_hdlr(INTRPT_Hndlr_Parms_T *parms)
{
#pragma exception_handler(hdlr_hdlr,0,0,_C2_MH_ESCAPE)

    // Generate another exception.
    // Handle exception generated in main.
}

int main(void)
{
#pragma exception_handler(main_hdlr,0,0,_C2_MH_ESCAPE)
    // Generate exception.
}

```

You can get an exception within an exception handler. To prevent exception recursion, exception-handler call stack entries act like control boundaries with regards to exception percolation. You can monitor for exceptions within your exception handlers. See the *System API Reference* for information about QMHCHGEM.

Unhandled Exceptions

If you do not handle an exception in the call stack entry that caused the exception, it is percolated (moved) to the caller's message queue. If the caller does not handle the exception message, then the message is percolated again. This continues until the exception reaches a control-boundary call stack entry, at which point AS/400 takes the default action for the unhandled exception.

If the message type is *STATUS, the program resumes without logging the unhandled exception.

If the message type is *ESCAPE, a function check is sent to the call stack pointed to by the resume cursor.

If the message type is a function check, then the call stack is canceled to the control boundary, and the message CEE9901 is sent to the caller of the control boundary. The resume cursor points to the caller of the control boundary.)

If the message type is *NOTIFY, the default reply is sent, and the program logs the message.

This program shows an unhandled exception. An exception is sent to the function fred(). The main() function is the control boundary.

Note: When an unhandled function check exception occurs or a termination verb is used in the program source, ILE transfers control to a call stack entry that represents the boundary for your program. This call stack entry is known as the *control boundary*.

Since the function `fred()` has no exception handlers, the exception is percolated to the `main()` function. Since the `main()` function has no exception handlers and `main()` is a control boundary, AS/400 takes the default action. Since, the exception is of type `*ESCAPE`, the function check is sent to `fred()`. The function check percolates to `main()`, and again the default is taken. The exception is of type function check, the call stack entries of `main()` and `fred()` are canceled, and the message CEE9901 is sent to the caller of the `main()` function.

```
// fred.cpp

#include <iostream.h>

void fred(void)

{
    char *p = NULL;
    *p = 'x';    // *ESCAPE exception
}

int main(void)
{
    fred();
}
```

Handling Exceptions in Your Programs

When you provide your own exception-handling routines rather than leaving this task to AS/400, you improve the consistency, reliability, and usability of your programs.

To provide exception handling in your program, you have to go through two steps:

1. Define which exception-handling routine you want to use, depending on your program flow.
2. Write the proper exception handler.

The exceptions that might occur in your program can be grouped into classes (floating-point exceptions, I/O exceptions). AS/400 recognizes that your program is going to use its own exception handler rather than the default one, once a handler routine is registered. The way you register your handlers determines which technique you can use.

You can use different techniques within the same program. But if you define several handlers through different techniques for the same exception class, AS/400 calls handlers in the following order:

1. Direct monitor handlers

2. ILE condition handlers (bindable APIs)
3. C++ and C exception-handling functions

Once the proper exception handler is invoked, then your code decides which action to perform. You may choose to:

- Handle the exception
- Promote the message and send the new message to a different stack entry
- Percolate the exception
- Move the resume cursor

Promoting a message is the action of modifying the exception message to a different message. This action marks the original exception message as handled, and restarts exception processing with a new exception message. This action is allowed only from direct monitor handlers and ILE condition handlers.

The exception handler priority becomes important if you use both language-specific error handling (C signal) and additional ILE exception-handler types. For the call stack entry that incurred the exception, AS/400 calls handlers in the following order:

1. Direct monitors
2. ILE condition handlers
3. `signal`

For portable code across multiple platforms, use only `signal`. Use ILE condition handlers if a consistent mechanism for handling exceptions across ILE-enabled languages is required.

Using `signal` always handles the exception implicitly (unless the signal action is `SIG_DFL`, in which case it percolates the exception). With direct monitor handlers you have to either specify one of the control actions that implicitly handles the exception (`_CTLA_HANDLE`, `_CTLA_HANDLE_NO_MSG`, `_CTLA_IGNORE`, or `_CTLA_IGNORE_NO_MSG`) or handle the exception explicitly within the handler function. To do the latter specify the control action `_CTLA_INVOKE` and `_CTLA_INVOKE_NO_MSG`, and use either `QMCHGEM` or an ILE condition-handling API.

Note: Direct monitors are usually the fastest handlers.

The HLL-specific handler, which is the signal handler in ILE C++, is global. It is enabled for all function calls in the activation group in which the `signal` function is called. ILE condition handlers and direct monitor handlers are scoped to the function that enables them, or until they are disabled in that function.

This program shows that if you do not want to change the state of a signal handler when the `signal()` function returns, then you must manage the state of the signal handler explicitly:


```

#include <signal.h>

void f(void)
{
    void (*old_state)(int);

    /* Save old state of signal action */

    old_state = signal(SIGALL,handlr);

    /* Other code in your program */
    /* Reset state of signal          */

    signal(SIGALL,old_state);
}

```

ILE condition handlers and direct monitor handlers do not have this requirement because they are not global handlers.

Note: It is not always sufficient to consider the exception handled when a handler is called. Some exception-handling methods consider an exception handled once the handler is invoked. Other methods expect the handler to change the message type explicitly, before they consider the exception handled.

Using Try-Catch-Throw

The `try-catch-throw` construct should not be used to monitor AS/400 system exceptions. It is used as a C++ construct you create in your source to communicate failures among programs. It provides type-safe transmission of information from a throw-point to a catch handler. It allows exceptions to be caught by the appropriate catch handler, and lets you write catch handlers that can catch groups of exceptions, as well as individual ones.

Use the `try-catch-throw` construct in your programs unless:

You are migrating a program that uses `signal()`

You are developing a module which must be integrated with other languages; in this case, use bindable APIs

Exception performance for a module is critical; in this case, use the **#pragma exception_handler** directive

You are developing a module that expects AS/400 system exceptions to be raised (run-time exception); in this case, use the **#pragma exception_handler** directive or condition handler, or use the `signal()` function if portability is a concern

Note: The various exception-handling methods are not all compatible with each other. The C++ `try-catch-throw` method does not interact with the other methods. To represent an AS/400 system exception with a C++ construct, you must first handle the AS/400 system exception at the point where it is generated. To do this, use either the **#pragma exception_handler** method or bindable API method, and then throw a C++ object to inform the C++ code that something is

wrong. Consider `try-catch-throw` as a method to communicate errors among your programs, but not a method to directly monitor AS/400 system exceptions.

This program shows how to use `try-catch-throw`. See the *C++ Language Reference* for information transferring control.

```
#include <stdlib.h>
#include <iostream.h>
#define MAXLEN      25
#define MAXWIDTH    80
class Err {
public:
    virtual void Print(void) { cout << "Any Error!" <<endl; }
};
class LenErr : public Err {
    int len;
public:
    LenErr (int i) { len = i; }
    virtual void Print(void) {
        cout << "Length Error!   Len = " << len << endl;
    }
};

class WidErr : public Err {
    int wid;
public:
    WidErr (int w) { wid = w; }
    virtual void Print(void) {
        cout << "Width Error!   Wid = " << wid << endl;
    }
};

class Shape {
    int len;
    int wid;
    int area;
public:
    Shape (char **arg) : len(atoi(arg-1-)), wid(atoi(arg-2-)) {
        if (len > MAXLEN)
            throw LenErr (len);
        if (wid > MAXWIDTH)
            throw WidErr (wid);
        area = len * wid;
    }
    void PrintInfo() {
        cout << "Length = " << len << endl;
        cout << "Width = " << wid << endl;
        cout << "Area   = " << area << endl;
    }
};

int main(int argc, char ** argv) {
    if (argc < 3) return 999;
    cout << "Program Start" << endl;
}
```

```

try {
    cout << "Try Start" << endl;
    Shape s (argv);
    s.PrintInfo();
    cout << "Try End" << endl;
}
catch (Err & error) {
    error.Print();
}
cout << "Program End" << endl;
return 0;
}

```

In this program `Shape` is a class and `LenError` and `WidError` are classes derived from class `Err`.

When the program executes, if `len` is greater than `MAXLEN`, the `throw LenError (len);` is executed and an exception is thrown. The thrown object is `LenError(len)`. The stack at the point where the throw occurred shows `main` at the bottom of the stack followed by `area`.

When the exception occurs, the stack is unwound until a catch clause has been found that is able to catch the thrown object. In this case, the catch clause is:

```

catch (Err &error) {
    error.Print();
}

```

During the unwinding process, all the objects that are created and going out of scope are destroyed by calling the appropriate destructors.

After all the objects that are going out of scope have been destroyed, the control is transferred from the throw point to the catch point. The `error.Print` member function is executed. Before the `error.Print()` function is executed, the stack frame shows `main` at the bottom of the stack.

If the thrown object cannot be caught after unwinding the whole stack, the `terminate()` function is called.

Try-Catch-Throw and Direct Monitor Handlers

This program shows how `try-catch-throw` can be used with the `#pragma exception_handler` directive to enhance the C++ error-handling capability. It demonstrates how to transform an AS/400 system-specific exception into a user-defined C++ exception.

In this program, class `PGMEEXEC` is designed to invoke functions through the member function `run()`. `run()` registers a `#pragma exception_handler` (`class_hdlr`) for detecting exceptions possibly generated during its execution. Upon receiving an exception, `class_hdlr` is invoked, which throws out an object of type `ESCINFO` to inform the client code of `run()`.

In `main()`, an object of class `PGMEEXEC` is constructed with a function pointer to `fred()`, which is invoked through `run()`. `fred()` incurs an AS/400 system exception (MCH3601) through dereferencing an invalid pointer. Exception MCH3601 cannot be percolated across to `main()` if there is a control boundary between `main()` and `fred()`. As a result, CEE9901 is sent to `main`. `class_hdlr` is invoked; it fills an `ESCINFO` structure with the exception information and throws it out to `main()`. This object is then caught by a catch handler, where the structure must be examined to understand what exception has failed the execution of `fred()`.

```
// File main.cpp

#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <except.h>

typedef int (*_FUNCPTR) ();
typedef struct {
    char    rcvmsg[8];
} ESCINFO;

extern "OS" void QMHCHGEM(_INVPTR*, int, unsigned int,
                        char*, char*, int, void*);

int fred(void);

void class_hdlr(_INTRPT_Hndlr_Parms_T* parm)
{
    ESCINFO msgdata;
    int ec=0;

    // Mark the exception as handled. Avoid having it percolate up the stack
    // and cancel the entire invocation

    QMHCHGEM(&(parm->Target),0,parm->Msg_Ref_Key, "*HANDLE  ",
            "",0,&ec);
    memcpy(msgdata.rcvmsg, parm->Msg_Id, 7);
    msgdata.rcvmsg[7] = '\\0';
    throw msgdata;
}

class PGMEEXEC {
private:
    _FUNCPTR    funcptr;

public:
    PGMEEXEC(_FUNCPTR fptr) { funcptr = fptr; }

    void run() {
        #pragma exception_handler(::class_hdlr, 0, 0,    \
                                _C2_MH_ESCAPE)
        funcptr();
        #pragma disable_handler
    }
};
```

```

    }
};

int main() {
    _FUNCPTR funcptr = fred;
    PGMEXEC PGM_EXECUTOR(funcptr);
    try {
        PGM_EXECUTOR.run();
    }
    catch (ESINFO &info) {
        cout << info.rcvmsg << " was sent to main" << endl;
    }
}

// File fred.cpp

int fred() {
    char a;
    char *p =0;
    a = *p;
    return 0;
}

```

Try-Catch-Throw and ILE Condition Handlers

This program shows how to use `try-catch-throw` with ILE condition handlers to enhance the C++ error-handling capability. It demonstrates how to transform an AS/400 system-specific exception into a user-defined C++ exception.

In this program, class `PGMEXEC` is designed to invoke functions through the member function `run()`. `run()` registers a condition handler (`class_hdlr`) for detecting exceptions possibly generated during its execution. Upon receiving an exception, `class_hdlr` is invoked, which throws out an object of type `ESINFO` to inform the client code of `run()`.

In `main()`, an object of class `PGMEXEC` is constructed with a function pointer to `fred()`, which is to be invoked through `run`. `fred()` incurs an AS/400 system exception (MCH3601) through dereferencing an invalid pointer. Exception MCH3601 cannot be percolated across to `main()` if there is a control boundary between `main()` and `fred()`. As a result, CEE9901 is sent to `main()`. `class_hdlr` is invoked; it fills an `ESINFO` structure with the exception information and throws it out to `main()`. This object is then caught by a catch handler, where the structure must be examined to understand what exception has failed the execution of `fred()`.

```

// File main.c

#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <leawi.h>

typedef int (*_FUNCPTR) ();
typedef struct {

```

```

        char    msg_prefix[3];
        int     msg_no;
    } ESCINFO;

int fred(void);

extern "C" void cond_hdlr(_FEEDBACK *cond, _POINTER *token, _INT4 *rc,
                        _FEEDBACK *xxx) {
    ESCINFO msgdata;
    memcpy(msgdata.msg_prefix, (const char*)cond->Facility_ID, 3);
    msgdata.msg_no = cond->MsgNo;
    *rc = CEE_HDLR_RESUME;
    throw msgdata;
}

class PGMEEXEC {
private:
    _FUNCPTR  funcptr;

public:
    PGMEEXEC(_FUNCPTR fptr) { funcptr = fptr; }
    void run() {
        _HDLR_ENTRY cond_hdlr_entry = cond_hdlr;
        CEEHDLR(&cond_hdlr_entry, NULL, NULL);
        funcptr();
        CEEHDLU(&cond_hdlr_entry, NULL);
    }
};

int main() {
    _FUNCPTR funcptr = fred;
    PGMEEXEC PGM_EXECUTOR(funcptr);
    try {
        PGM_EXECUTOR.run();
    }
    catch (ESCINFO &info) {
        cout << info.msg_prefix << hex << info.msg_no
             << " was sent to main" << endl;
    }
}

// File fred.cpp

int fred() {
    char a;
    char *p=0;
    a = *p;
    return 0;
}

```

Cases In Which Try-Catch-Throw Does Not Catch An Exception

Not all exceptions can be caught. This program shows a case in which a handler might not be able to catch a thrown exception. The catch handler is not located in the same activation group as the throw operation:

```
// File a.cpp

int a() {
    throw 7;
}

// File main.cpp

#include <stdlib.h>
#include <iostream.h>
#include <terminat.h>

int a(void);

int main() {
    try {
        a();
    }
    catch(int) { cout << "In catch(int)" <<endl; }
}
```

File `a.cpp` is to be created as a service program with the `ACTGRP(*NEW)` parameter so it will not run in the same activation group as `main`. Function `a` cannot throw any object across to `main()` because a thrown exception is not permitted to cross a control boundary. The call stack entry of `a` is a control boundary. As a result, CEE9901 is sent to `main` instead.

The various exception handling methods are not all compatible. The C++ `try-catch-throw` method does not interact with the other methods.

Direct Monitor Intercepts a Thrown Exception: This program shows a case where a catch handler is not able to catch a thrown exception. The function containing the catch handler also registers a direct monitor handler for `*STATUS` message. This direct monitor intercepts the thrown exception because direct monitor handlers are handled first by the system, before other types of exception handlers.

```
#include <stdlib.h>
#include <iostream.h>
#include <except.h>
#include <terminat.h>

const int MARK_HNDL = 0;
const int MARK_UNHNDL = 1;

// Indicate to pragma_hdlr whether to mark an exception as handled

int mark_except = 0;
```

```

// The following API is used by pragma_hdlr to mark an AS/400
// system exception as handled

extern "OS" {
    void QMHCHGEM(_INVPTR*, int, unsigned int, char*, char*,
                 int, void*);
}

void pragma_hdlr(_INTRPT_Hndlr_Parms_T* parm)
{
    int ec=0;
    cout << "In #pragma handler" <<endl;

    if (mark_excpt == MARK_HNDL) {
        cout << "The thrown exception is marked as handled" <<endl;
        QMHCHGEM(&(parm->Target),0,parm->Msg_Ref_Key, \
                 "*HANDLE ", "", 0, &ec);
    }
}

void a(int i) {
    throw i;
}

void my_terminate() {
    cout << "my_terminate is invoked to end the program"
    <<endl;
}

typedef void(*PFV)();
void main() {
    PFV old_term = set_terminate(my_terminate);

    // Monitoring for a "_C2_MH_STATUS" type exception
    // intercepts a thrown exception as #pragma exception_handler
    // is handled first by the system

    #pragma exception_handler(pragma_hdlr,0,0,_C2_MH_STATUS);
    mark_excpt = MARK_UNHNDL;
    try {
        a(1234);
    }
    catch (int i) {
        cout << "In catch handler, thrown integer = " << i << endl;
    }
    catch (...) {
        cout << "In catch(...), failed" <<endl;
        exit(1);
    }
    mark_excpt = MARK_HNDL;
}

```



```

try {
    a(2456);
}

// pragma_hdlr does intercept the thrown exception. It
// then marks the exception as handled.
// There is no exception object for the catch handler.
// Consequently, the run-time behavior is undefined.

catch (...) {
    cout << "In catch(...), failed" <<endl;
    exit(1);
}
}

```

The output is:

```

In #pragma handler
In catch handler, thrown integer = 1234
In #pragma handler
The thrown exception is marked as handled

```

ILE Condition Handler Intercepts a Thrown Exception: This program shows a case where a handler is not able to catch a thrown exception.

The function containing the catch handler also registers an ILE condition handler for *STATUS message. The condition handler intercepts the thrown exception because condition handlers are handled second by the system.

```

#include <stdlib.h>
#include <iostream.h>
#include <lecond.h>
#include <terminat.h>

const int MARK_HNDL = 0;
const int MARK_UNHNDL = 1;

// Indicate to cond_hdlr whether to mark an exception as handled

int mark_excpt = 0;

extern "C" void cond_hdlr(_FEEDBACK *cond, _POINTER *token, \
                        _INT4 *rc, _FEEDBACK *xxx) {
    cout << "In condition handler" <<endl;

    if (mark_excpt == MARK_HNDL) {
        cout << "The thrown exception is marked as handled" <<endl;
        *rc = CEE_HDLR_RESUME;
    }
}

void a(int i) {

```

```

        throw i;
    }

void my_terminate() {
    cout << "my_terminate is invoked to end the program"
        <<endl;
}

typedef void(*PFV)():
void main() {
    PFV old_term = set_terminate(my_terminate);
    _HDLR_ENTRY handler_entry = cond_hdlr;
    CEEHDR(&handler_entry, NULL, NULL);
    mark_except = MARK_UNHNDL;

    try {
        a(1234);
    }
    catch (int i) {
        cout << "In catch handler, thrown integer = " << i << endl;
    }
    catch (...) {
        cout << "In catch(...), failed" <<endl;
        exit(1);
    }
    mark_except = MARK_HNDL;
    try {
        a(2456);
    }

    // cond_hdlr intercepts the thrown exception. It
    // then marks the exception as handled.
    // There is no exception object for the catch handler.
    // Consequently, the run-time behavior is undefined.

    catch (...) {
        cout << "In catch(...), failed" <<endl;
        exit(1);
    }
}

```

The expected output is:

```

In condition handler
In catch handler, thrown integer = 1234
In condition handler
The thrown exception is marked as handled

```

Using the C Language Signal Function

Using the `signal()` function for exception handling contributes to the portability of your programs, since it is implemented by a wide range of different platforms.

Only the basic functions (`signal()` and `raise()`) provided by the standard C library are supported by most C and C++ compilers, while the exception analysis routines are platform dependent.

AS/400 System Exceptions

AS/400 system exceptions are mapped to C signals by the run time. The C mechanism that determines the course of action for a signal is called a signal handler.

Note: You cannot register a signal handler in an activation group that is different from the one you wish to invoke it from. If a signal handler is in a different activation group from the occurrence of the signal it is handling, the behavior is undefined.

Signals are raised either implicitly or explicitly. To explicitly raise a signal, use the `raise()` function. Signals are implicitly raised by AS/400 when an exception occurs. If you call a program that does not exist, an implicit signal is raised, indicating that the program object could not be found.

The header file `<signal.h>` contains a number of function prototypes associated with signal handling.

The `raise()`, `signal()`, and `_GetExcData()` functions can be used with signal handling in your program.

The `_GetExcData()` function allows you to obtain information about the exception message associated with the signal, and returns a structure containing information about the exception message. The `_GetExcData()` function returns the same structure that is passed to the **#pragma exception_handler** directive.

Setting up a Signal Handler: This program shows that, when there is no signal handler set up, the default action for SIGIO is SIG_IGN. The exception is ignored. When a signal handler is set up for SIGIO, the signal handler is called.

```
#include <iostream.h>
#include <stdio.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

#define FILE_NAME "QTEMP/MY_FILE"
#define RCD_LEN 80
#define NUM_RCD 5
#define BUFFERSIZE 256

// The signal handler for SIGIO.

static void handler_SIGIO(int sig)
{
    cout << "In SIGIO handler " << endl;
    cout << "Exception message ID is " << _EXCP_MSGID << endl;
}
```

```

    cout << "Signal raised is " << sig <<endl;
}

// The signal handler for SIGALL.

static void handler_SIGALL(int sig)
{
    _INTRPT_Hndlr_Parms_T data;
    _GetExcData(&data);

    cout << "In SIGALL handler " << endl;
    cout << "Exception message ID is " << data.Msg_Id << endl;
    cout << "Signal raised is " << sig << endl;
}

// Since this code does not re-register this handler
// for SIGIO, the action for this handler is SIG_DFL.

int main(void)
{
    _RFILE *fp;
    int i;
    char buf[RCD_LEN];
    char cmd[100];

    // Create a file.

    streambuf x(cmd, BUFFERSIZE);
    ostream strout(100,cmd);
    strout << "CRTPF FILE(" << FILE_NAME << ")" << "RCDLEN("
        << RCD_LEN << ")" ;// Store command as string in cmd
    system(cmd);

    // Open the file for write.

    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        cout << "Open for write fails" << endl;
        exit(1);
    }

    // Write some data into the file.
    memset(buf, '1', RCD_LEN);

    for ( i = 0; i < NUM_RCD; i++ )
    {
        _Rwrite(fp, buf, RCD_LEN);
    }

    _Rclose(fp);

    // Open the file for the first read.

```

```

    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        cout << "Open for the first read fails" << endl;
        exit(2);
    }

// Read until end-of-file.
// Since there is no signal handler set up and the default
// action for SIGIO is SIG_IGN.

    i = 1;
    cout << "The first read starts" << endl;
    for (i=_Ropnfbk(fp)->num_records; i>0; --i)
    {
        _Rreadn(fp, buf, RCD_LEN, __DFT);
        cout << "Read record " << i++ << endl;
    }

    _Rclose(fp);

    cout << "The first read finishes" <<endl;

// Set up a signal handler for SIGIO.

    signal(SIGIO, handler_SIGIO);

// Open the file for the second read.

    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        cout << "Open for the second read fails" << endl;
        exit(3);
    }

// Read until end of file.
// Since a signal handler is set up for SIGIO, the signal
// handler is called.

    i = 1;
    cout << "The second read starts" << endl;
    for (i=_Ropnfbk(fp)->num_records; i>0; --i)
    {
        _Rreadn(fp, buf, RCD_LEN, __DFT);
        cout << "Read record " << i++ << endl;
    }
    _Rclose(fp);
    cout << "The second read finishes" << endl;

// Set up a signal handler for SIGALL.
    signal(SIGALL, handler_SIGALL);

```

```

// Open the file for the third read.
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        cout << "Open for the third read fails" << endl;
        exit(4);
    }

// Read until end of file.
// Since there is no signal handler for SIGIO but there is a
// signal handler for SIGALL, the signal handler for SIGALL
// is called. But
// the signal ID passed to the SIGALL signal handler is still
// equal to SIGIO.

    i = 1;
    cout << "The third read starts" << endl;
    for (i=(_Ropnfbk(fp))->num_records; i>0; --i)
    {
        _Rreadn(fp, buf, RCD_LEN, __DFT);
        cout << "Read record " << i++;
    }
    _Rclose(fp);
    cout << "The third read finishes" << endl;
}

```

The output is:

```

The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1

```

```

Read record 2
Read record 3
Read record 4
Read record 5
In SIGIO handler
Exception message ID is CPF5001
Signal raised is 9
The second read finishes
The third read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGALL handler
Exception message ID is CPF5001
Signal raised is 9
The third read finishes
Press ENTER to end terminal session.
==>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window

```

Writing a Separate Exception-Handling Program: Since the signal vector has activation group scope, if you register an exception handler in one program, it is visible to separate ILE programs running in the same activation group.

This program shows how to write a separate exception-handling program. The second program EHI1 sets the exception handlers, which are then visible in the activation group.

```

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>

extern "OS" int EHI1(jmp_buf *);

int main(void)
{
    float a, b, c;
    jmp_buf env;

    if (setjmp(env)) {
        cout << "You entered wrong parameters...." << endl;
        cout << "Enter first value: " << endl;
        cin >> a;
        cout << "Enter second value: " << endl;
        cin >> b;
    }
    else {
        EHI1(&env);
        c = a/b;
        cout << "Result:" << c;
    }
}

```

```

    }
}

```

The first program, shown above, might generate a floating-point exception. The handlers in the second program, shown below, are invoked.

```

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>

static jmp_buf env;

void Abend_Handler(int);
void I_O_Err_Handler(int);
void F_P_Err_Handler(int);

int main(int argc, char **argv)
{
    if (argc < 1)
        cout << "No environment saved..." << endl;
    else
        memcpy(&env, (jmp_buf *) argv[1], sizeof(jmp_buf));
    signal(SIGABRT, &Abend_Handler);
    signal(SIGIO, &I_O_Err_Handler);
    signal(SIGFPE, &F_P_Err_Handler);
}

void Abend_Handler(int sig)
{
    _INTRPT_Hndlr_Parms_T s;
    cout << "Exception handling by Abort_Handler..." << endl;
    _GetExcData(&s);

    char buf[8];
    memcpy(buf, Message_ID, 7); buf[7]=0;
    cout << "Message_ID:" << s.Msg_Id;
    char buf1[8];
    memcpy(buf1, Exception_Id, 7); buf1[7]=0;
    cout << "Exception Id:" << s.Exception_Id << endl;
    char buf2[49];
    memcpy(buf2, Exception_Id, 48); buf2[48]=0;
    cout << "Exception Data:" << s.Ex_Data << endl;

    signal(SIGABRT, &Abend_Handler);
}

void F_P_Err_Handler(int sig)
{
    _INTRPT_Hndlr_Parms_T s;

```



```

cout << "Exception handling by F_P_Handler..." << endl;
_GetExcData(&s);
char buf[8];
memcpy(buf, Message_ID, 7); buf[7]=0;
cout << "Message_ID:" << s.Msg_Id;
char buf1[8];
memcpy(buf1, Exception_Id, 7); buf1[7]=0;
cout << "Exception Id:" << s.Exception_Id << endl;
char buf2[49];
memcpy(buf2, Exception_Id, 48); buf2[48]=0;
cout << "Exception Data:" << s.Ex_Data << endl;

signal(SIGFPE, &F_P_Err_Handler);
longjmp(env, 1);
}

void I_O_Err_Handler(int sig)
{
    _INTRPT_Hndlr_Parms_T s;
    cout << "Exception handling by I_O_Err_Handler..." << endl;
    _GetExcData(&s);
    char buf[8];
    memcpy(buf, Message_ID, 7); buf[7]=0;
    cout << "Message_ID:" << s.Msg_Id;
    char buf1[8];
    memcpy(buf1, Exception_Id, 7); buf1[7]=0;
    cout << "Exception Id:" << s.Exception_Id << endl;
    char buf2[49];
    memcpy(buf2, Exception_Id, 48); buf2[48]=0;
    cout << "Exception Data:" << s.Ex_Data << endl;

    signal(SIGIO, &I_O_Err_Handler);
}

```

The Signal Function

The `signal()` function specifies the action that is performed when a signal is raised. There are ten signals represented as macros in the `<signal.h>` header file:

SIGABRT	Abnormal program end
SIGFPE	Arithmetic operation error, such as dividing by zero
SIGILL	Illegal instruction
SIGINT	AS/400 system interrupt, such as receiving an interactive attention signal
SIGIO	Record file error condition
SIGOTHER	All other *ESCAPE and *STATUS messages that do not map to any other signals
SIGSEGV	The access to storage is not valid
SIGTERM	A termination request is sent to the program

SIGUSR1 Reserved for user-defined signal handler

SIGUSR2 Reserved for user-defined signal handler

SIG_IGN and SIG_DFL are signal actions that are included in the `<signal.h>` header file.

SIG_IGN Ignore the signal

SIG_DFL Default action for the signal

SIGALL allows you to register your own default handling function for all signals whose action is SIG_DFL. This default handling function can be registered by using the `signal()` function with SIGALL. SIGALL cannot be signaled by the `raise()` function.

When a signal is received, the run time chooses one of three ways to handle the signal:

If the value of the function is SIG_IGN, the signal is ignored because the exception is handled by the run time and no signal handler is called. If the message mapped to the signal is an *ESCAPE or *NOTIFY message, it is placed in the job log.

If the value of the function is a pointer to a function, the function addressed by the pointer is called.

If the value of the function is SIG_DFL, AS/400 uses the value registered for SIGALL (choosing one of the three ways described here). If the value of the function for SIGALL is SIG_DFL, then the exception is percolated.

Note: The value of the function is the function argument on the call to the `signal()` function.

Signal Handling-Action Definitions

This table shows the initial state of the C signal values and their handling-action definitions. SIG_DFL always percolates the condition to the next handler. Resume indicates that the exception is handled, and the program continues.

Table 21. Handling-Action Definitions for Signal Values

Signal Value	Initial State	SIG_DFL	SIG_IGN	Return from Handler
SIGABRT ¹	SIG_DFL	Percolate	Ignore	Resume
SIGALL ²	SIG_DFL	Percolate	Ignore	Resume
SIGFPE	SIG_DFL	Percolate	Ignore ³	Resume
SIGILL	SIG_DFL	Percolate	Ignore ³	Resume
SIGINT	SIG_DFL	Percolate	Ignore	Resume
SIGIO	SIG_IGN	Percolate	Ignore	Resume
SIGOTHER	SIG_DFL	Percolate	Ignore ³	Resume
SIGSEGV	SIG_DFL	Percolate	Ignore ³	Resume
SIGTERM	SIG_DFL	Percolate	Ignore	Resume
SIGUSR1	SIG_DFL	Percolate	Ignore	Resume
SIGUSR2	SIG_DFL	Percolate	Ignore	Resume
Note:				
1	Can only be signaled by the <code>raise</code> or <code>abort</code> functions.			
2	SIGALL can not be signaled by the <code>raise</code> function.			
3	If the value of signal is SIGFPE, SIGILL, or SIGSEGV, the behavior is undefined. If the signal is hardware-generated, the behavior is undefined.			

Signal to AS/400 Exception Mapping

Table 22 on page 283 shows what signal the AS/400 system exception messages map to. All *ESCAPE exception messages are mapped to signals. The *STATUS and *NOTIFY messages that map to SIGIO are defined in Table 28 on page 323.

<i>Table 22. Signal to AS/400 Exception Mapping</i>	
Signal	Message
SIGABRT	C2M1601
SIGALL	C2M1610 (if explicitly raised)
SIGFPE	C2M1602, MCH1201 to MCH1204, MCH1206 to MCH1215, MCH1221 to MCH1224, MCH1838 to MCH1839,
SIGILL	C2M1603, MCH0401, MCH1002, MCH1004, MCH1205, MCH1216 to MCH1219, MCH1801 to MCH1802, MCH1807 to MCH1808, MCH1819 to MCH1820, MCH1824 to MCH1825, MCH1832, MCH1837, MCH1852, MCH1854 to MCH1857, MCH1867, MCH2003 to MCH2004, MCH2202, MCH2602, MCH2604, MCH2808, MCH2810 to MCH2811, MCH3201 to MCH3203, MCH4201 to MCH4211, MCH4213, MCH4296 to MCH4298, MCH4401 to MCH4403, MCH4406 to MCH4408, MCH4421, MCH4427 to MCH4428, MCH4801, MCH4804 to MCH4805, MCH5001 to MCH5003, MCH5401 to MCH5402, MCH5601, MCH6001 to MCH6002, MCH6201, MCH6208, MCH6216, MCH6220, MCH6403, MCH6601 to MCH6602, MCH6609 to MCH6612,
SIGINT	C2M1604
SIGIO	C2M1609, See Table 28 on page 323 for the exception mappings.
SIGOTHER	C2M1611 (if explicitly raised)
SIGSEGV	C2M1605, MCH0201, MCH0601 to MCH0606, MCH0801 to MCH0803, MCH1001, MCH1003, MCH1005 to MCH1006, MCH1220, MCH1401 to MCH1402, MCH1602, MCH1604 to MCH1605, MCH1668, MCH1803 to MCH1806, MCH1809 to MCH1811, MCH1813 to MCH1815, MCH1821 to MCH1823, MCH1826 to MCH1829, MCH1833, MCH1836, MCH1848, MCH1850, MCH1851, MCH1864 to MCH1866, MCH1898, MCH2001 to MCH2002, MCH2005 to MCH2006, MCH2201, MCH2203 to MCH2205, MCH2401, MCH2601, MCH2603, MCH2605, MCH2801 to MCH2804, MCH2806 to MCH2809, MCH3001, MCH3401 to MCH3408, MCH3410, MCH3601 to MCH3602, MCH3603 to MCH3604, MCH3802, MCH4001 to MCH4002, MCH4010, MCH4212, MCH4404 to MCH4405, MCH4416 to MCH4420, MCH4422 to MCH4426, MCH4429 to MCH4437, MCH4601, MCH4802 to MCH4803, MCH4806 to MCH4812, MCH5201 to MCH5204, MCH5602 to MCH5603, MCH5801 to MCH5804, MCH6203 to MCH6204, MCH6206, MCH6217 to MCH6219, MCH6221 to MCH6222, MCH6401 to MCH6402, MCH6404, MCH6603 to MCH6608, MCH6801
SIGTERM	C2M1606
SIGUSR1	C2M1607
SIGUSR2	C2M1608

Using ILE Condition Handlers

Condition handlers are exception handlers that are used to handle, percolate, or promote exceptions. An exception is presented to the condition handlers in the form of an ILE condition.

You can register one or more condition handlers at run time, using the Register ILE Condition Handler (`CEEHDLR`) bindable API. ILE condition handlers may be unregistered by calling the Unregister ILE Condition Handler (`CEEHDLU`) bindable API. Include the `<lecond.h>` header file in your source code when using these bindable APIs.

Use the ILE bindable API CEEHDLR if you want to have a consistent mechanism of condition handling across several ILE languages. Or use it for scoping exception handling to a call stack entry. Unlike the signal handler, which is scoped to the activation group, the ILE bindable API CEEHDLR is scoped to the function that calls it.

The procedure `main()` in the program `MYPGM` registers the condition handler `main_handler()` followed by a call to `fred()`, which registers the condition handler `my_handler()`. The functions `my_handler()` and `main_handler()` are defined in the service program `HANDLERS`.

`fred()` gets an exception, which causes `my_handler()` to get control, followed by `main_handler()`. The `main()` function is a control boundary. Although the exception is considered unhandled, a function check is sent to `fred()`. The handlers `my_handler()` and `main_handler()` are called again, this time for the function check. Neither of them handles the function check, the program terminates abnormally, and CEE9901 is sent to the caller of `main()`:

```
// MYPGM

#include <lecond.h>
#include <iostream.h>

void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *newl );

void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *newl );

void fred(void)
{
    _HDLR_ENTRY hdlr=my_handler;
    char *p = NULL;
    CEEHDLR(&hdlr, NULL,NULL);
    *p = 'x';    // exception
    CEEHDLU(&hdlr,NULL);
}

int main(void)
{
    _HDLR_ENTRY hdlr=main_handler;
    CEEHDLR(&hdlr, NULL,NULL);
    fred();
}
```

```

// HANDLERS *SRVPGM (*CALLER)

#include <signal.h>
#include <iostream.h>
#include <lecond.h>

// HANDLERS *SRVPGM (*CALLER)

void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new1)
{
    return;
}

void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new1)
{
    cout << "In main_handler" << endl;
}

```

Coding an ILE Condition Handler

This program shows how to use a condition handler to handle an exception. In this program, the ILE condition handler `cond_hdlr` is registered in the `main()` function using `CEEHDLR`. A MCH1211 (divide-by-zero) exception then occurs. The handler `cond_hdlr` is called, and it indicates that the exception should be handled. Control then resumes in `main`.

```

// This program uses the ILE bindable API CEEHDLR to handle a
// condition.

#include <iostream.h>
#include <stdlib.h>
#include <leawi.h>

// A condition handler registered by a call to CEEHDLR in main.

void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new1 )
{
    *rc = CEE_HDLR_RESUME; // handle the condition
    cout << "condition was raised: Facility_ID = " << cond->Facility_ID
        << "MsgNo = " << cond->MsgNo << endl;
}

int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;

    // Register the condition handler.

    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)

```

```

    {
        cout << "Failed to register the condition handler\n";
        exit(88);
    }

// Cause a divide by zero condition.

    x = y / zero;

// The code resumes here after the condition has been handled.

    cout << "The condition was handled.\n";
}

```

The output is:

```

condition was raised: Facility_ID = MCH, MsgNo = 0x1211
The condition was handled.
Press ENTER to end terminal session.

```

Coding Two ILE Condition Handlers

This program shows the use of two condition handlers. Using the bindable API CEEHDLR, `main_hdlr()` is registered in `main()`, and `fred_hdlr()` is registered in `fred()`. A MCH1211 (divide-by-zero) exception then occurs.

The handler `fred_hdlr()` is called, which then tests to see if the exception is an MCH1211. The result code in the condition handler is set to percolate to the next condition handler. The handler `fred_hdlr()` returns without handling the exception. Therefore `main_hdlr()` gets called.

The user-supplied token is then updated to the value '1', and the result code is set to handle the exception. The handler `main_hdlr` returns and the exception is handled.

Control resumes in `fred` following the statement that caused the divide-by-zero.

```

// This program uses the ILE bindable API CEEHDLR to enable handlers
// that percolate and handle a condition.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>

// A condition handler registered by a call to CEEHDLR in fred

void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *newl )
{
    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {

```

```

        *rc = CEE_HDLR_PERC;           // ... let it percolate to main ...
        cout << "in fred_hdlr, percolate exception." << endl;
    }
    else
    {
        *rc = CEE_HDLR_RESUME;         // ... otherwise handle it.
        cout << "in fred_hdlr, handle exception." << endl;
    }
}

// A condition handler registered by a call to CEEHDLR in main
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *newl )
{
    cout << "in main_hdlr: Facility_ID = " << cond->Facility_ID
         << cond->MsgNo << endl;
    **(_INT4 **)token = 1;             // Update the user's token.
    *rc = CEE_HDLR_RESUME;           // Handle the condition
}

int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;

    // Register the handler without a token.

    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        cout << "Failed to register the condition handler" << endl;
        exit(88);
    }

    // Cause a divide-by-zero condition.
    x = y / zero;
    cout << "Resume here because resume cursor not moved and main_hdlr"
         << " handled the exception" <<endl;
}

int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;

    // Register the handler with a token of type _INT4.

    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {

```



```

        cout << "Failed to register the condition handler" << endl;
        exit(99);
    }
    fred();

// See if the condition handler for main updated the token.

    if (*tokenp == 1)
        cout << "A condition was percolated from fred to main and"
            << " was then handled." << endl;
    }

```

The output is:

```

in fred_hdlr, percolate exception.
in main_hdlr: Facility_ID = MCH, MsgNo = 0x1211
Resume here because resume cursor not moved and main_hdlr handled the excepti
on
A condition was percolated from fred to main and was then handled.
Press ENTER to end terminal session.

```

Using Direct Monitor Handlers

Direct monitor handlers allow you to register an exception monitor directly around a limited number of C++ source statements. This exception-handling method is language-specific and platform-dependent, since the definition of the handling routines occurs at compile time, through a pragma preprocessor directive, rather than through a call to a bindable function. For C++ this capability is enabled through the **#pragma exception_handler** and **#pragma disable_handler** directives. Include the `<except.h>` header file in your source code when using these directives.

The **#pragma exception_handler** is a directive that enables a direct monitor handler from **#pragma exception_handler** to **#pragma disable_handler** without considering the program logic in between.

The direct monitor handler may either be a function, or code following a label within the function containing the **#pragma exception_handler**. A `communications` area variable may be specified on the **#pragma exception_handler**, and has two different uses:

1. If the handler is a label, then the communications area is used as storage for the standard exception handler parameter block of type `_INTRPT_Hndlr_Parms_T` (which is defined in `<except.h>`). AS/400 fills in the structure prior to giving control to the label.¹

¹ If the storage required for the exception handler parameter block exceeds the storage defined by `com_area`, then the remaining bytes are truncated.

2. If the handler is a function, then AS/400 passes a pointer to a structure of type `_INTRPT_Hndlr_Parms_T` to the function. A pointer to the communications area is available inside the structure. The definition of this structure is:

```
typedef _Packed struct {
    unsigned int    Block_Size;        // Size of the parameter block
    _INVFLAGS_T    Tgt_Flags;        // Target invocation flags
    char           reserved[8];       // reserved
    _INVPTR        Target;            // Current target invocation
    _INVPTR        Source;            // Source invocation
    _SPCPTR        Com_Area;          // Communications area
    char           Compare_Data[32];  /* Compare Data
    char           Msg_Id[7];         /* Message ID
    char           reserved1;        // 1 byte pad
    _INTRPT_Mask_T Mask;              // Interrupt class mask
    unsigned int   Msg_Ref_Key;       // Message reference key
    unsigned short Exception_Id;     // Exception ID
    unsigned short Compare_Data_Len; // Length of Compare Data
    char           Signal_Class;      // Internal signal class
    char           Priority;           // Handler priority
    short          Severity;          // Message severity
    char           reserved3[4];
    int            Msg_Data_Len;      // Length of available message data
    char           Mch_Dep_Data[10]; // Machine dependent data
    char           Tgt_Inv_Type;      // Invocation type (in MIMCHOBS.H)
    _SUSPENDPTR    Tgt_Suspend;      // Suspend pointer of target
    char           Ex_Data[48];       // First 48 bytes of exception data
} _INTRPT_Hndlr_Parms_T;
```

Element Description

`Block_Size` The size of the parameter block passed to the exception handler.

`Tgt_Flags` Contains flags used by AS/400.

`reserved` An 8-byte reserved field.

`Target` An invocation pointer to the call stack entry that enabled the exception handler.

`Source` An invocation pointer to the call stack entry that caused the exception. If that call stack entry no longer exists, then this is a pointer to the call stack entry where control resumes when the exception is handled.

`Com_Area` A pointer to the communications area variable specified as the second parameter on the **#pragma exception_handler**. If a communications area is not specified, specify zero for this parameter.

`Compare_Data`

The compare data consists of 4 bytes of message prefix (CPF, MCH) followed by 28 bytes, which are taken from the message data of the related message. Where the message data is greater than 28, these are the first 28 bytes. For MCH messages, these are the first 28 bytes of the exception-related data that is returned by AS/400.

Msg_Id A message identifier (CPF123D).
Mask This is an 8-byte exception mask, identifying the type of the exception that occurred, such as decimal data error, function check.
reserved1 A 1-byte pad.
Msg_Ref_Key
 A key used to uniquely identify the message.
Exception_Id
 Binary value of the exception ID, 0x123D.
Compare_Data_Len
 The length of the compare data.
Signal_Class
 Internal signal class.
Priority The handler priority.
Severity The message severity.
reserved3 A 4-byte reserved field.
Msg_Data_Len
 The length of available message data.
Mch_Dep_Data
 Machine dependent data.
Tgt_Inv_Type
 Invocation type. Macros are defined in the header file <mimchobs.h>.
Tgt_Suspend
 Suspend pointer of the target.
Ex_Data The first 48 bytes of exception data.

The direct monitor handlers are scoped at compile time to the code between the **#pragma exception_handler** directive and the **#pragma disable_handler** directive, regardless of the C++ control flow at run time.

The **#pragma exception_handler** directive is scoped to a block of code independent of the program logic. The **#pragma exception_handler** is enabled around the call to the `raise()` function. The conditional expression `if (ca != 0)` has no effect on enabling the direct monitor handler. The logic path for the conditional expression `if (ca != 0)` is never taken, but `my_handler` is enabled.

```

volatile int ca=0;
if (ca != 0){
    #pragma exception_handler(my_handler, ca,0,_C2_MH_ESCAPE)
}
else {
    raise(SIGINT);            // Signal is caught by my_handler
}
#pragma disable_handler
  
```

There are two programs: the source for a program MYPGM and the source for the service program HANDLERS. The procedure `main()` in MYPGM registers the direct monitor handler `main_handler()` followed by a call to `fred()`, which registers the direct monitor handler `my_handler()`. `fred()` gets an exception, which causes `my_handler()` to get control, followed by `main_handler()`. The `main()` function is a control boundary. Since the exception is considered unhandled, a function check is sent to `fred()`. Since the handlers `my_handler()` and `main_handler()` only handle *ESCAPE messages, neither is called again. Since the function check goes unhandled at `main()`, the program terminates abnormally and CEE9901 is sent to the caller of `main()`.

```
// MYPGM *PGM

#include <except.h>
#include <iostream.h>

void my_handler(_INTRPT_Hndlr_Parms_T *parms);

void main_handler(_INTRPT_Hndlr_Parms_T *parms);

void fred(void)
{
    char *p = NULL;
    #pragma exception_handler(my_handler, 0,0,_C2_MH_ESCAPE)
    *p = 'x';    // exception
    #pragma disable_handler
}

int main(void)
{
    #pragma exception_handler(main_handler, 0,0,_C2_MH_ESCAPE)
    fred();
}

// HANDLERS *SRVPGM (created with activation group *CALLER)

#include <signal.h>
#include <iostream.h>

void my_handler(_INTRPT_Hndlr_Parms_T *parms)
{
    return;
}

void main_handler(_INTRPT_Hndlr_Parms_T *parms)
{
    cout <<"In main_handler" <<endl;
}
```

Exception Classes

Exception classes indicate the type of exception (*ESCAPE, *NOTIFY, *STATUS, function check) and, for machine exceptions, the low-level type (pointer not valid, divide by zero). Direct monitor handlers monitor for exceptions based on exception classes and

message identifiers. The handler gets control if the exception falls into one or more of the exception classes specified on the **#pragma exception_handler**.

All machine exceptions are mapped to the *ESCAPE type exception. To monitor for machine exceptions you can either specify the machine exception class, specify all *ESCAPE exceptions, or specify both:

```
#include <except.h>

// Just monitor for pointer not valid exceptions

#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, 0)

// Monitor for all *ESCAPE messages

#pragma exception_handler(eh, 0, 0, _C2_MH_ESCAPE)

// Although this is valid, there is no need to specify
// _C1_POINTER_NOT_VALID because it is covered by
// _C2_MH_ESCAPE

#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, _C2_MH_ESCAPE)

// To monitor for only specific messages, use the extended form of
// pragma exception_handler.
// This #pragma only monitors for MCH3601 *ESCAPE msg.

#pragma exception_handler (eh, 0, 0, _C2_MH_ESCAPE, _CTLA_HANDLE, "MCH3601")
```

Macros for AS/400 machine exception classes are defined in the header file `<except.h>`.

You can monitor for the exception class values for *class1* and *class2* parameters. The value of *class2* can only be one of `_C2_MH_ESCAPE`, `_C2_MH_STATUS`, `_C2_MH_NOTIFY`, or `_C2_MH_FUNCTION_CHECK` as defined in the `<except.h>` header file. Table 23 defines all the exception classes you can specify.

Table 23 (Page 1 of 2). Exception classes

Bit position	Header File Constant in <code><except.h></code>	Exception class
0	<code>_C1_BINARY_OVERFLOW</code>	Binary overflow or divide by zero
1	<code>_C1_DECIMAL_OVERFLOW</code>	Decimal overflow or divide by zero
2	<code>_C1_DECIMAL_DATA_ERROR</code>	Decimal data error
3	<code>_C1_FLOAT_OVERFLOW</code>	Floating-point overflow or divide by zero
4	<code>_C1_FLOAT_UNDERFLOW</code>	Floating-point underflow or inexact result
5	<code>_C1_INVALID_FLOAT_OPERAND</code>	Floating-point invalid operand or conversion error

Table 23 (Page 2 of 2). Exception classes

Bit position	Header File Constant in <except.h>	Exception class
6	_C1_OTHER_DATA_ERROR	Other data error, for example, edit mask
7	_C1_SPECIFICATION_ERROR	Specification (operand alignment) error
8	_C1_POINTER_NOT_VALID	Pointer not set/pointer type invalid
9	_C1_OBJECT_NOT_FOUND	Object not found
10	_C1_OBJECT_DESTROYED	Object destroyed
11	_C1_ADDRESS_COMP_ERROR	Address computation underflow/overflow
12	_C1_SPACE_ALLOC_ERROR	Space not allocated at specified offset
13	_C1_DOMAIN_OR_STATE_VIOLATION	Domain/State protection violation
14	_C1_AUTHORIZATION_VIOLATION	Authorization violation
15-28	_C1_VLIC_RESERVED	VLIC reserved
29	_C1_OTHER_MI_EXCEPTION	Remaining MI-generated exceptions (other than function check)
30	_C1_MI_GEN_FC_OR_MC	MI-generated function check/machine check
31	_C1_MI_SIGEXP_EXCEPTION	Message generated via Signal Exception instruction
32-39	n/a	reserved
40	_C2_MH_ESCAPE	*ESCAPE
41	_C2_MH_NOTIFY	*NOTIFY
42	_C2_MH_STATUS	*STATUS
43	_C2_MH_FUNCTION_CHECK	function check
44-63	n/a	reserved

Notes:

1. The macro `C1_ALL` defined in the `<except.h>` header file can be used as the equivalent of all the valid `class1` exception masks.
2. 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.

Control Actions

The `#pragma exception_handler` directive allows you to specify a control action that is to be taken during exception processing. The five control actions that can be specified, as defined in the `<except.h>` header file, are:

`_CTLA_INVOKE`

This control action causes the function named on the directive to be invoked, and does not handle the exception. The exception remains active and must be handled by using `QMCHGEM` or one of the ILE condition-handling APIs.

`_CTLA_HANDLE`

This control action causes the function or label named on the directive to get control, and it handles and logs the exception implicitly. The exception is no longer active when the handler gets control.

`_CTLA_HANDLE_NO_MSG`

This control action is the same as `_CTLA_HANDLE`, except that the exception is not logged. The message reference key in the parameter block passed to the handler is zero.

`_CTLA_IGNORE`

This control action handles and logs the exception implicitly and does not pass control to the handler function named on the directive; the function named is ignored. The exception is no longer active and execution resumes at the instruction immediately following the instruction that caused the exception.

`_CTLA_IGNORE_NO_MSG`

This control action is the same as `_CTLA_IGNORE`, except only exceptions of class `*NOTIFY` are logged.

This program shows how to specify the *control action* parameter on the **`#pragma exception_handler`** directive. The code causes an MCH3601 (pointer not set) exception. The control action `_CTLA_IGNORE` causes the exception to be handled without calling the handler function. The output of this code is the message "Passed the exception."

```
#include <except.h>
#include <iostream.h>

void myhandler(void) {
    cout <<"In handler - something's wrong!" <<endl;
    return;
}

int main(void)
{
    int *ip;
    volatile int com_area;
    #pragma exception_handler(myhandler, com_area, 0, _C2_ALL, \
                             _CTLA_IGNORE)
    *ip = 5;

    cout <<"Passed the exception." <<endl;
}
```

Specifying Message Identifiers

The `#pragma exception_handler` directive also gives you the capability to specify one or more specific or generic message identifiers on the directive. When one or more identifiers are specified on the directive, the direct monitor handler takes effect only when an exception occurs whose identifier matches one of the identifiers on the directive.

To specify message identifiers on the directive, you have to specify a control action to be taken. The class of the exception must be in one of the classes specified on the directive. This code shows a `#pragma exception_handler` directive that enables a monitor for a single specific message, MCH3601:

```
#pragma exception_handler (myhandler, com_area, 0, _C2_ALL, \
                           _CTLA_HANDLE, "MCH3601")
```

This source shows a `#pragma exception_handler` directive that enables a monitor for several floating-point exceptions:

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \
                           _CTLA_IGNORE, "MCH1206 MCH1207 MCH1209 MCH1213")
```

The ability to specify generic message identifiers can be used to simplify the directive in the previous example. This source shows that a monitor is enabled for any exception whose identifier begins with MCH12":

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \
                           _CTLA_IGNORE, "MCH1200")
```

Using ILE Condition Handlers to Promote an Exception: This program shows how to use condition handlers to promote an exception. Using the bindable API CEEHDLR, `main_hdlr` is registered in `main()`, and `fred_hdlr()` is registered in `fred()`. A MCH1211 (divide-by-zero) exception then occurs.

The handler `fred_hdlr` is called because of the MCH1211 exception. This handler moves the resume cursor to the resume point in `main` using the bindable API CEEMRCCR. It builds a condition token for CEE9902, and the result code is set to promote. The handler `fred_hdlr` returns, and the original MCH1211 is promoted to a CEE9902.

The handler `main_hdlr` is called because of the CEE9902 exception. The result code is set to handle the condition. The handler `main_hdlr` returns and the CEE9902 is handled.

Control resumes in the statement following the call to `fred()` in `main()`.

```
// This program uses the ILE bindable API CEEHDLR to promote a
// divide-by-zero condition to a CEE9902.
```

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
```

```
// A condition handler registered by a call to CEEHDLR in fred
```



```

void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new1 )
{
    _INT4 type=1;
    CEEMRCR(&type, NULL); // Move the resume cursor to the resume point
                          // in main

    // If it's a divide-by-zero error ...

    cout << "in fred_hdlr: moving resumes. ";
    cout << "Facility_ID = " << cond->Facility_ID << cond->MsgNo << endl;

    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {
        *rc = CEE_HDLR_PROM; //... Promote the condition to unexpected error.*/
        *new = *cond;
        memcpy(new->Facility_ID, "CEE", 3);
        new->MsgNo = 9902;
        cout << "promoting condition...." << endl;
    }
    else
    {
        *rc = CEE_HDLR_PERC; //...Otherwise, percolate to the next handler.
        cout << "percolating condition...." << endl;
    }
}

// A condition handler registered by a call to CEEHDLR in main

void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new1 )
{
    if (!memcmp(cond->Facility_ID, "CEE", 3) && cond->MsgNo == 9902;)
        **(_INT4 **)token = 1;           // Got the promoted CEE9902.
    else
        **(_INT4 **)token = 2;           // It is not a CEE9902.
    *rc = CEE_HDLR_RESUME;               // Handle the condition.
}

int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;

    // Register the handler without a token.

    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        cout << "Failed to register the condition handler" << endl;
        exit(88);
    }
}

```

```

// Cause a divide-by-zero condition.
x = y / zero;

// This is not the resume point because of the call to CEEMRCR in
// fred_hdlr.

{
    cout << "This is not the resume point: should not get here" << endl;
}

}

int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;

// Register the handler with a token of type _INT4.

    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {
        cout << "Failed to register the condition handler" << endl;
        exit(99);
    }
    fred();

// See if the condition handler for main received the promoted
// condition.

    if (*tokenp == 1)
        cout << "A condition was promoted from MCH1211 to CEE9902 by "
            << "fred and was handled by the condition handler enabled "
            << "in main" << endl;
}

```

The output is:

```

in fred_hdlr: moving resumes. Facility_ID = MCH, MsgNo = 0x1211
promoting condition...
A condition was promoted from MCH1211 to CEE9902 by fred and was handled by
the condition handler enabled in main
Press ENTER to end terminal session.

```

Direct Monitor Handlers Using Labels: This program shows direct monitor handlers using labels instead of functions as the handlers.

```

#include <except.h>
#include <signal.h>
#include <stdlib.h>
#include <iostream.h>

void sig_hdlr(int sig){
    cout <<"Signal handler should not have been called" <<endl;
}

int main(void)
{
    int a=0;
    char *p=NULL;
    volatile _INTRPT_Hndlr_Parms_T ca;

    // Set up signal handler for SIGFPE. The signal handler function
    // is never invoked, since the exception is handled
    // by the direct monitor handlers.

    if( signal(SIGFPE,sig_hdlr) == SIG_ERR )
    {
        cout <<"Could not set up signal handler for SIGFPE" <<endl;
    }

    // The following direct monitor
    // traps and handles any *ESCAPE exceptions.

    #pragma exception_handler(LABEL_1, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE)

    // Generate exception(divide by zero). The CTL_ACTION specified
    // should take effect (exception handled and logged), execution
    // resumes at LABEL_1.

    a/=a;
    cout <<"We should never reach this point" <<endl;
    LABEL_1: cout <<"The MCH1211 exception was handled" <<endl;
    #pragma disable_handler

    // The following direct monitor
    // only traps and handles MCH3601 exceptions

    #pragma exception_handler(LABEL_2, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE, "MCH3601")

    // Generate MCH3601(*ESCAPE message). The CTL_ACTION specified
    // should take effect (exception handled and logged), execution
    // resumes at LABEL_2.

    *p='X';
    cout <<"We should never reach this point" <<endl;
}

```

```

    LABEL_2: cout <<"The MCH3601 exception was handled" <<endl;
}

```

The output is:

```

The MCH1211 exception was handled
The MCH3601 exception was handled

```

Direct Monitor Handler and Signal Function: This program shows you how to use the **#pragma exception_handler** and the `signal()` function together. It shows how an exception is handled using SIGIO. An end-of-file message is mapped to SIGIO. The default for SIGIO is SIG_IGN. It also shows that when both a HLL-specific handler and direct monitor handler are defined, the direct monitor handler is called first.

```

// This program illustrates how to use direct monitor handlers.

#include <stdio.h>
#include <iostream.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <except.h> // Include except.h even though it is included
                  // in the signal.h header file.

#define FILE_NAME  "QTEMP/MY_FILE"
#define RCD_LEN    80
#define NUM_RCD    5
#define BUFFERSIZE 256

typedef struct error_code{
    int    byte_provided;
    int    byte_available;
    char   exception_id[7];
    char   reserve;
    char   exception_data[1];
    error_code_t;
}

static int handle_flag;

extern "OS" void QMCHGEM(_INVPTR *, int, unsigned int, char *,
                        char *, int, error_code_t *);

// The signal handler.

static void sig_handler(int sig)
{
    cout << "In signal handler" << endl;
    cout << "Exception message ID is " << _EXCP_MSGID<< endl;
}

// The direct monitor handler.

```

```

static void exp_handler(_INTRPT_Hndlr_Parms_T *exp_info)
{
    error_code_t      error_code;
    cout << "In direct monitor handler" <<endl;
    cout << "Exception message ID is" << exp_info->Compare_Data
        << (unsigned) exp_info->Exception_Id << endl;

    // Call QMHCHGEM API to handle the exception.

    if ( handle_flag )
    {
        error_code.byte_provided = 8;
        QMHCHGEM(&(exp_info->Target), 0, exp_info->Msg_Ref_Key,
            "HANDLE ", "", 0, &error_code);
    }
}

// The function to read a file.

static void read_file(_RFILE *fp)
{
    int i = 1;
    for (i=(_Ropnfbk(fp))->num_records; i>0; --i)
    {
        _Rreadn(fp, NULL, RCD_LEN, __DFT);
        cout << "Read record " << i++<< endl;
    }
}

int main(void)
{
    _RFILE      *fp;
    int         i;
    volatile int com;
    char        buf[RCD_LEN];
    char        cmd[100];

    // Create a file.

    streambuf x(cmd, BUFFERSIZE);
    ostream strout(100,cmd);
    strout << "CRTPF FILE(" << FILE_NAME << ")" << "RCDLEN("
        << RCD_LEN << ")" ;// Store command as string in cmd
    system(cmd);

    // Open the file for write.

    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        cout << "Open for write fails" <<endl;
        exit(1);
    }
}

```

```

// Write some data into the file.
memset(buf, '1', RCD_LEN);
for ( i = 0; i < NUM_RCD; i++ )
{
    _Rwrite(fp, buf, RCD_LEN);
}
_Rclose(fp);

// Open the file for the first read.

if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    cout << "Open for the first read fails" <<endl;
    exit(2);
}

// Read until end-of-file.
// Since no signal handler or direct monitor handler is set up,
// the default value for SIGIO is
// SIG_IGN.

i = 1;
cout << "The first read starts" <<endl;
    for (i=(_Ropnfbk(fp))->num_records; i>0; --i)
    {
        _Rreadn(fp, buf, RCD_LEN, __DFT);
        cout << "Read record " << i++ << endl;
    }
_Rclose(fp);
cout <<"The first read finishes" <<endl;

// Set up a direct monitor handler and a signal handler.
// Tell the direct monitor handler to handle the exception.
// The direct monitor handler exp_handler calls the message
// handler API QMHCHGEM with the parameter *HANDLE. This marks the
// exception as handled.
// Use exception classes to handle machine exceptions.

handle_flag = 1;
#pragma exception_handler(exp_handler, com, 0,
    _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGIO, sig_handler);

// Open the file for the second read.

if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    cout <<"Open for the second read fails" <<endl;
    exit(3);
}

```

```

// Read until end of file.
// The direct monitor handler
// is called first. Since it marks the exception as handled,
// the signal handler is not called.

    i = 1;
    cout << "The second read starts" <<endl;
        for (i=(_Ropnfbk(fp))->num_records; i>0; --i)
        {
            _Rreadn(fp, buf, RCD_LEN, __DFT);
            cout <<"Read record " << i++ << endl;
        }
    _Rclose(fp);
    cout << "The second read finishes" <<endl;

// Disable the direct monitor handler.

    #pragma disable_handler

// Set up a direct monitor handler and a signal handler.
// Set the global variable handle_flag to zero so that the
// direct monitor does not handle the exception.

    handle_flag = 0;
    #pragma exception_handler(exp_handler, com, 0,
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS) \
    signal(SIGALL, sig_handler);

// Open the file for the third read.

    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        cout << "Open for the third read fails" <<endl;
        exit(4);
    }

// Read until end-of-file.
// The direct monitor handler
// is called first. Since the exception is not marked as
// handled, the signal handler is then called.

    i = 1;
    cout <<"The third read starts" <<endl;
        for (i=(_Ropnfbk(fp))->num_records; i>0; --i)
        {
            _Rreadn(fp, buf, RCD_LEN, __DFT);
            cout << "Read record " << i++ <<endl;
        }
    _Rclose(fp);
    cout << "The third read finishes" <<endl;

// Disable the direct monitor handler.

```

```

        #pragma disable_handler

// Set up a direct monitor handler and a signal handler.

        #pragma exception_handler(exp_handler, com, 0,
                _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS) \
        signal(SIGIO, sig_handler);

// Open the file for the fourth read.

        if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
        {
            cout << "Open for the fourth read fails" <<endl;
            exit(5);
        }

// Read until end-of-file.
// Since there is no direct monitor handler for the read_file function,
// the signal handler is called.
//
// The direct monitor handler in main is not called because the
// exception was mapped to SIGIO and the signal handler gets called
// at function read_file.

        cout << "The fourth read starts" <<endl;
        read_file(fp);
        _Rclose(fp);
        cout << "The fourth read finishes" <<endl;

// Disable the direct monitor handler.
        #pragma disable_handler
    }

```

The output is:


```

The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
The second read finishes
The third read starts
Read record 1

```

```

Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
In signal handler
Exception message ID is CPF5001
The third read finishes
The fourth read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In signal handler
Exception message ID is CPF5001
The fourth read finishes
Press ENTER to end terminal session.
===>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window

```

Using Cancel Handlers

A cancel handler may be enabled around a body of code inside a function. It only gets control if the suspend point of the call stack entry is inside that code (within the **#pragma cancel_handler** and **#pragma disable_handler** directives), and the call stack entry is canceled.

The **#pragma cancel_handler** directive provides a way to register a cancel handler statically within a call stack entry (or suspend point within a call stack entry). The Register Call Stack Entry Termination User Exit Procedure (**CEERTX**) and the Unregister Call Stack Entry Termination User Exit Procedure (**CEETUTX**) ILE bindable APIs provide a way to register a user-defined routine dynamically to be executed when the call stack

entry for which it is registered is canceled. The *System API Reference* contains information on these ILE bindable APIs.

On the **#pragma cancel_handler** directive, the name of the cancel handler routine (a bound ILE procedure) is specified, along with a user-defined communications area. It is through the communications area that information is passed from the program to the handler function.

When the cancel handler function is called, it is passed a pointer to a structure of type `_CNL_Hndlr_Parms_T`, which is defined in the `<except.h>` header file. This structure contains a pointer to the communications area in addition to some other useful information passed by AS/400. This additional information includes the reason the call was canceled.

The definition of this structure is:

```
typedef _Packed struct {
    unsigned int    Block_Size;
    _INVFLAGS_T    Inv_Flags;
    char            reserved[8];
    _INVPTR        Invocation;
    _SPCPTR        Com_Area;
    _CNL_Mask_T    Mask;
} _CNL_Hndlr_Parms_T;
```

Element	Description
Block_Size	The size of the parameter block passed to the cancel handler.
Inv_Flags	Contains flags used by AS/400.
reserved	An 8-byte reserved field.
Invocation	An invocation pointer to the invocation's cancellation.
Com_Area	A pointer to the handler communications area defined by the cancel handler.
Mask	A 4-byte value indicating the reason for cancellation.

Cancel Handler Reason Codes

Table 24 on page 306 lists the bits that are set in the reason code. Table 25 on page 307 describes the reason codes for canceling invocations. If the activation group is to be terminated, then the *activation group is terminating* bit is also set in the reason code. These bits must be correlated to `_CNL_MASK_T` in `_CNL_Hndlr_Parms_T` in `<except.h>`. Column 2 contains the macro constant defined for the cancel reason mask in `<except.h>`.

Table 24. Determining Canceled Invocation Reason Codes

Function	Bits in reason code	Rationale
Library routines		
exit	_EXIT_VERB	Since the definition of <code>exit</code> is normal termination, invocations canceled by this function are done with a reason code of <i>normal</i> .
abort	_ABNORMAL_TERM_EXIT_VERB	Since the definition of <code>abort</code> is abnormal termination, invocations canceled by this function are done with a reason code of <i>abnormal</i> .
longjmp	_JUMP	The general use of <code>longjmp</code> is to return from an exception handler, although it may be used in non-exception situations as well. Since it is used as part of the "normal" path for a program, and therefore any invocations canceled because of it are canceled with a reason code of <i>normal</i> .
Unhandled function check	_ABNORMAL_TERM_UNHANDLED_EXCP	Not handling an exception that is an abnormal situation.
AS/400 system APIs		
CEEMRCR	_ABNORMAL_TERM_EXCP_SENT	This API is only used during exception processing. It is typically used to cancel invocations where a resume is not possible, or at least where the behavior would be undefined if control were resumed in them. Also, these invocations have had a chance to handle the exception but did not do so. Invocations canceled by this API are done with a reason code of <i>abnormal</i> .
QMHSNDPM /QMHRSNEM (escape messages) Message Handler APIs	_ABNORMAL_TERM_EXCP_SENT	All invocations down to the target invocation are canceled without any chance of handling the exception. The <i>System API Reference</i> contains information on these APIs.
AS/400 commands		
Process termination	_ABNORMAL_TERM_PROCESS_TERM _AG_TERMINATING	Any externally initiated shutdown of an activation group is considered abnormal.
RCLACTGRP	_ABNORMAL_TERM_RCLRSC	The default is abnormal termination. The termination can be normal if a normal/abnormal flag is added to the command.

Table 25. Common Reason Code for Canceling Invocations		
Bit	Description	Header File Constant <except.h>
Bits 0	Reserved	
Bits 1	Invocation canceled due to sending exception message	_EXCP_SENT
Bits 2-15	Reserved	
Bit 16	0 - normal termination 1 - abnormal termination	_ABNORMAL_TERM
Bit 17	Activation Group is terminating	AG_TERMINATING
Bit 18	Initiated by Reclaim Activation Group (RCLACTGRP)	_RCLRSC
Bit 19	Initiated by process termination	_PROCESS_TERM
Bit 20	Initiated by an <code>exit</code> function	_EXIT_VERB
Bit 21	Initiated by an unhandled functioncheck	_UNHANDLED_EXCP
Bit 22	Invocation canceled due to a <code>longjmp</code> function	_JUMP
Bit 23	Invocation canceled due to a jump because of exception processing	_JUMP_EXCP
Bits 24-31	Reserved (0)	

Setting up Cancel Handlers: This program shows the use of the ILE cancel-handler mechanism. This capability allows a program the opportunity to call a user-provided function to perform things such as error reporting, error logging, or file closing, when a particular function invocation is canceled. The usual ways to cause cancellation to occur are: using the `exit()` or `abort()` functions, using the `longjmp()` function to jump to an earlier invocation, and having a CEE9901 Function Check generated from an unhandled exception.

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <except.h>

// The following function is called a "cancel handler". It is
// registered for a particular invocation (function) with the
// #pragma cancel_handler directive. The variable identified
// on this directive as the "communications area" can be accessed
// using the 'Com_Area' member of the _CNL_Hndlr_Parms_T structure.

void CancelHandlerForReport( _CNL_Hndlr_Parms_T *cancel_info ) {
    cout << "In Cancel Handler for function 'Report' ..." <<endl;

    // Changing the value in the communications area updates the
    // 'return_code' variable in the invocation being canceled
    // (in function 'Report' in this example). Note that the
    // compiler issues a warning for the following
    // statement since it uses a non-ANSI C compliant technique.
    // However, this does not affect the expected run-time behavior.
    // Set "return_code" in Report to an arbitrary number.
```

```

*( (volatile unsigned *)cancel_info->Com_Area ) = 500;

cout << "Communications Area now has the value: "
    << *( (volatile unsigned *)cancel_info->Com_Area) << endl;
cout << "Leaving Cancel Handler for function 'Report'..." <<endl;
}

// The following function is also called a cancel handler but has
// been registered for the 'main' function. That is, when the
// main function is canceled, this function is automatically
// called by AS/400.

void CancelHandlerForMain( _CNL_Hndlr_Parms_T *cancel_info ) {
    cout << "In Cancel Handler for function 'main' ..." <<endl;

    // Changing the value in the communications area updates the
    // 'return_code' variable in the invocation being canceled
    // (in function main in this example). Note that the
    // compiler issues a warning for the following
    // statement since it uses a non-ANSI C compliant technique.
    // However, this does not affect the expected run-time behavior.
    // Set "return_code" in main to an arbitrary number.

    *( (volatile unsigned *)cancel_info->Com_Area ) = 999;
    cout << "Communications Area now has the value: "
        << *( (volatile unsigned *)cancel_info->Com_Area) << endl;
    cout << "Leaving Cancel Handler for function 'main'..." <<endl;
}

// The following is a simple function that registers another function
// (named 'CancelHandlerForReport' in this example) as its "cancel
// handler". When exit is used from this function, then this
// invocation and all prior invocations are canceled by AS/400
// and any registered cancel handlers functions are automatically
// called.

void Report( void ) {
    volatile unsigned return_code;           // communications area
    #pragma cancel_handler( CancelHandlerForReport, return_code )
    cout << "in function Report...about to call 'exit'..." <<endl;

    // Using the exit function causes this function invocation
    // and all function invocations within this program to be
    // canceled. If any of the functions being canceled have
    // cancel handlers enabled, then those cancel handler functions
    // are called by AS/400 after each cancellation.

    exit( 99 );                             // exit with an arbitrary value
    cout << "in function Report just after calling 'exit'..." <<endl;
    #pragma disable_handler
}

```

```

// In the main function a cancel handler is registered so that
// the function CancelHandlerForMain is called if main is
// canceled.

int main( void ) {
volatile unsigned return_code;           // communications area
#pragma cancel_handler( CancelHandlerForMain, return_code )
    return_code = 0;                     // initialize return code which
                                         // eventually is set in the cancel handler
    cout << "In main about to call Report..." <<endl;
    Report();
    cout << "...back from calling Report " <<endl;
    cout << "return_code = " <<return_code << endl;
#pragma disable_handler
}

```

Understanding the Control Boundary in ILE

Use this legend for the next three figures:

Activation Group

Dashed frame with rounded corners

Program/Service Program

Dotted frame with square corners

Call stack entry

Solid frame with rounded corners

Control Boundary

A shaded call stack entry

Figure 27 on page 310 shows how a control boundary is defined in a simple C++ program that runs in a single *NEW activation group. This activation group is also called an AS/400 system named activation group. The program consists of a C++ program called TEST, which is bound by reference to a service program SRV1. The service program SRV1 was created with the ACTGRP(*CALLER) option.

There is only one control boundary at procedure main in program TEST. This call stack entry is a control boundary since it is the first call stack entry in the activation group.

If an escape exception occurs in function f() and no handlers are enabled, then the exception percolates from the call stack entry for f() to the call stack entry for g(), and then to the call stack entry for main(). Since main() is the control boundary, and the exception is unhandled, it turns into a function check and is re-driven. The function check starts at call stack entry f() (where the exception occurred), and is percolated to the call stack entry for g(), and then to the call stack entry for main(). At this point, the function check has reached a control boundary and has not been handled. Therefore, a CEE9901 *ESCAPE message is sent to the caller of main(), and the activation group is terminated.

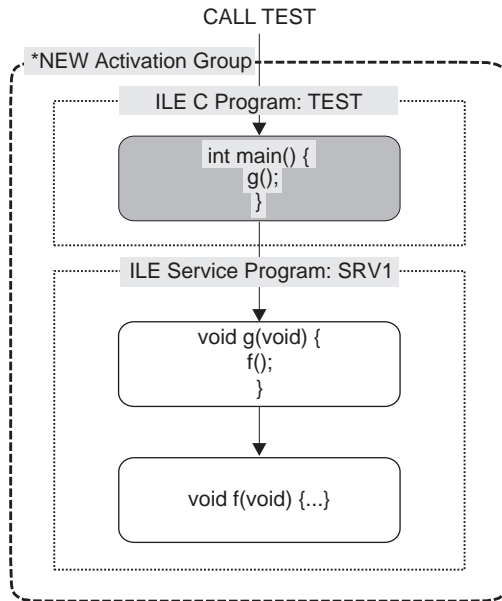


Figure 27. A Control Boundary When There is One Activation Group

Figure 28 on page 311 shows how control boundaries are set when calls are made between different activation groups. This program consists of one C++ program and two service programs (SRV1 and SRV2) that run in named activation groups (AG1 and AG2). The functions `main()`, `a()`, `b()`, and `e()` are all potential control boundaries. When you are running in procedure `c()`, then `b()` would be the nearest control boundary.

The call stack entry for `main` is a control boundary since it is the first call stack entry in the activation group. The call stack entries for `a()`, `b()`, and `e()` are control boundaries, since the immediately preceding call stack entry for each runs in a different activation group.

If an escape exception occurs in function `d()` and no handlers are enabled, then the exception percolates from the call stack entry for `d()` to the call stack entry for `c()`, and then to the call stack entry for `b()`. Since the call stack entry `b()` is a control boundary, the exception is turned into a function check and is re-driven. The function check starts at function `d()`, and is percolated to the call stack entry for `c()`, and then to the call stack entry for `b()`. At this point, the function check has reached a control boundary and has not been handled. A CEE9901 *ESCAPE message is sent to the call stack entry for function `a()`, and the activation group AG2 is terminated.

If `exit()` was called from within function `a()`, then it cancels up the call stack entry for `a()`, control returns to the call stack entry for `main()`, and the activation group AG1 is terminated. At this time, `atexit()` routines for activation group AG1 are called, if any exist.

If `exit()` was called from within function `f()`, then it cancels the call stack entries until it reaches the control boundary at the call stack entry for `e()` (for example, the call stack entries for both `e()` and `f()` are canceled). Since this control boundary is not the first one in activation group `AG1`, the activation group is not taken down, and control returns to the call stack entry for function `d()`.

Note: If an `atexit()` is registered by function `f()`, it is not called, since the activation group is not terminated.

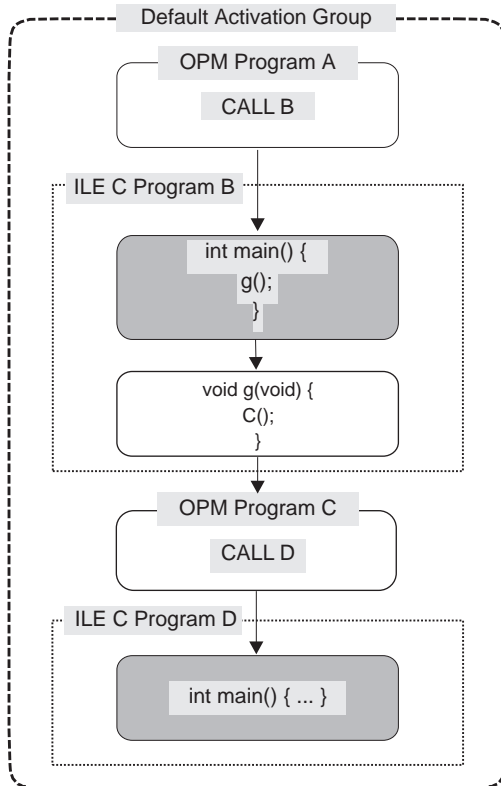


Figure 28. Control Boundaries When There are Multiple Activation Groups

Figure 29 on page 312 shows how control boundaries are set for a program that is running in the default activation group. The program consists of two OPM programs, A and C, and two ILE C++ programs, B and D, which were created using the `ACTGRP(*CALLER)` option on the `AS/400 CRTPGM` command. In this program, the function `main()` in program B and the function `main()` in program D are potential control boundaries.

The call stack entries for `main()` in programs B and D are control boundaries since the immediately preceding call stack entry for each is that of an OPM program.

If an escape exception occurs in function `g()` in program B and no handlers are enabled, then the exception percolates from the call stack entry for `g()` to the call stack entry for `main()`. Since the call stack entry for `main()` is a control boundary, the exception is turned into a function check and is re-driven. The function check starts at function `g()`, and is percolated to the call stack entry for `main()`. At this point, the function check has reached a control boundary and has not been handled. The call stack entries for `g()` and `main()` are canceled, and a CEE9901 escape message is sent to the call stack entry for program A.

Note: The activation group is not taken down, since the default activation group only goes away at job termination time.

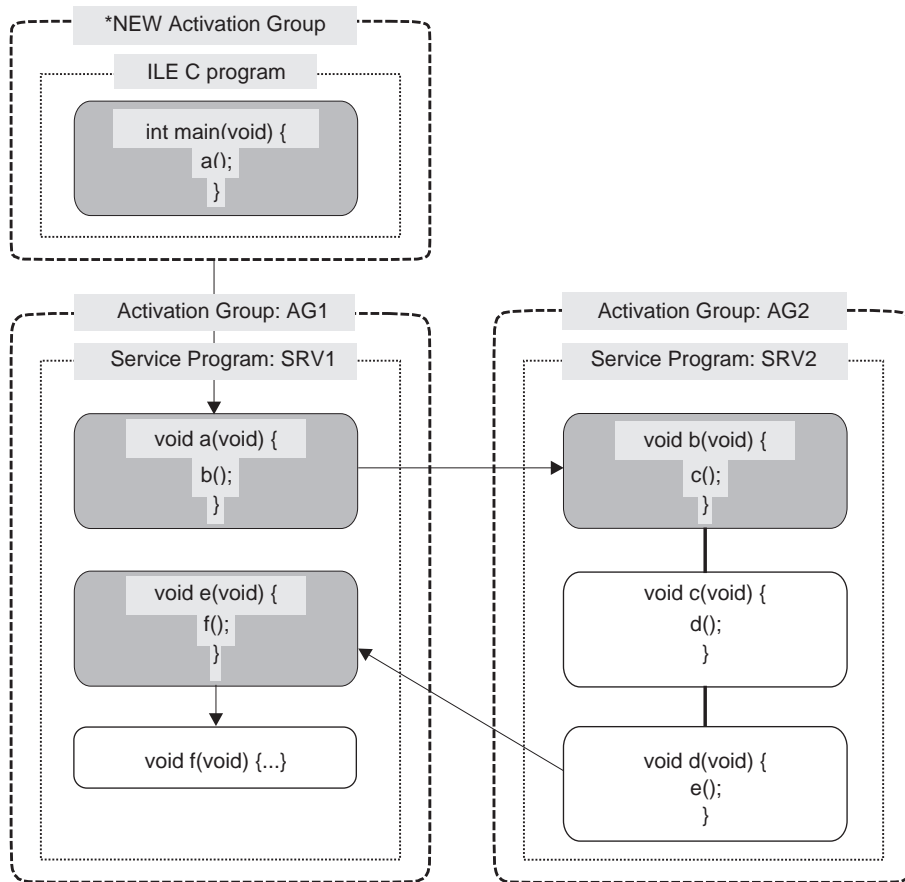


Figure 29. Control Boundaries in the Default Activation Group

Percolating Exceptions

This is the sequence of exceptions and handling actions that occur when the source code is executed:

1. An escape exception occurs in function `fred()`.
2. `handler1()` gets control since it is monitoring for an escape message, and since it is the closest nested monitor to the exception. As a direct handler, it has the highest priority.
3. `handler2()` gets control since it is monitoring for an escape message, and since it has a higher priority than a CEEHDLR.
4. `handler3()` gets control (from CEEHDLR).
5. `signal()` handler gets control. Even though it is registered in `main()`, `signal()` is scoped to the activation group and, therefore, gets control. It gets control after `handler1()`, `handler2()`, and `handler3()` since it has a lower priority than either direct handlers or CEEHDLRs. Since the action is `SIG_DFL`, the exception is not handled.
6. The exception is percolated to `main()`.
7. `handler4()` gets control.
8. Exception is still not handled. Therefore, when it hits the control boundary (the PEP for `main()`), it is turned into a function check and is re-driven.
9. `handler1()` does not get control since it is not monitoring for a function check.
10. `handler2()` gets control since it is monitoring for function check.
11. `handler3()` gets control since CEEHDLRs get control for all `*ESCAPE`, `*STATUS`, `*NOTIFY`, and `*FC` messages.
12. `signal()` handler does not get control since `signal()` does not recognize function checks.
13. The function check is percolated to `main()`.
14. `handler4()` gets control since it is monitoring for function check.
15. The function check percolates to the control boundary and causes termination.
16. (CEE9901) `*ESCAPE` is sent to the caller of `main()`.

```
#include <iostream.h>
#include <except.h>
#include <signal.h>
#include <lecond.h>
```

```
void handler1(_INTRPT_Hndlr_Parms_T *parms)
{
    cout <<"In handler1: does not handle the exception" <<endl;
}
```

```
void handler2(_INTRPT_Hndlr_Parms_T *parms)
{
```

```

    cout <<"In handler2: does not handle the exception" <<endl;
}

void handler3(_FEEDBACK *condition, _POINTER *token, _INT4 *result_code,
             _FEEDBACK *new_condition)
{
    cout <<"In handler3: does not handle the exception" <<endl;
}

void handler4(_INTRPT_Hndlr_Parms_T *parms)
{
    cout <<"In handler4: does not handle the exception" <<endl;
}

void fred(void)
{
    _HDLR_ENTRY hdlr = handler3;
    char *p = NULL;
    #pragma exception_handler(handler2, 0, 0, \
                             _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    CEEHDLR(&hdlr, NULL, NULL);
    #pragma exception_handler(handler1, 0, 0, _C2_MH_ESCAPE)
    *p = 'x'; // exception
}

int main(void)
{
    signal(SIGSEGV, SIG_DFL);
    #pragma exception_handler(handler4, 0, 0, \
                             _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    fred();
}

```

Handling Errors within Your Program in Other Ways

While designing your program, you can use other techniques to recover from situations that might otherwise lead to exceptions at run time. You can write code that checks the:

- Return value of a function

- errno value

- Major and minor return code to detect stream file errors

- Values in the `_RIOFB_T` structure to detect record file errors.

Note: If you do not check the `errno` value or return codes, you may find it difficult to locate the errors. It may also prove time-consuming to check all return codes in all files.

Checking the Return Value of a Function

Many C standard library functions have a return value associated with them for error-checking purposes. The `fopen()` function returns `NULL` if a file is not opened successfully. The *C Library Reference* contains information about the C function return values.

Your program should check the function return value to verify that the function has completed successfully. This is accomplished by checking for the return value that indicates that the function failed:

```
// Source to Check for the Return Value of fopen.

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

int main(void)
{
    FILE *fp;

    if (( fp = fopen ( "MYLIB/QCSRC(TEST)", "ab" )) == NULL )
    {
        cout << "Cannot open file QCSRC(TEST)"<< endl;
        exit (99);
    }
}
```

Checking the Errno Value

The `<errno.h>` header file contains various declarations for defined error conditions. Many functions set `errno` to specific values, depending on the type of error. These values are also defined in the `<errno.h>` header file. The implementation of `errno` contains a function call.

Since the ILE implementation of `errno` uses a function call, it cannot be displayed as a variable in the ILE System Debugger. To display the value returned by `errno` in the debugger:

1. Declare a global variable that is a pointer to integer `static int *errno_temp:`
2. At the entry point of the program, which is the `main()` function, assign the `errno_temp` variable to the storage location that contains `errno`.

```
main() {
    errno_temp = &errno;
}
```

3. At any point in the debug session, you may display the current value of `errno` by displaying the integer to which `errno_temp` is pointing. From the command line, enter `eval *errno_temp`

Your program can use the `strerror` and `perror()` functions to print the value of `errno`. The `strerror()` function returns a pointer to an error message string associated with `errno`. The `perror()` function prints a message to `stderr`. Both functions should be

used immediately after a function is called, since subsequent calls might alter the `errno` value.

Note: Your program should always initialize `errno` to zero before calling a function, because `errno` is not reset by any of the C library functions. Initialize `errno` to zero after an error has occurred.

This source code shows how to check the `errno` value:

```
// Check the errno Value for fopen

#include <errno.h>

errno = 0;

fp = fopen ( "MYLIB/QCSRC(TEST)", "ab" );

if ( errno !=0 )
{
    perror("Error occurred while opening file.\n");
    exit (1);
}
```

Errno Macros

Table 26 lists the error macros that the C library functions can set `errno` to:

<i>Table 26 (Page 1 of 3). Errno Macros</i>		
Error Macro	Description	Set by Function
EBADDATA	The message data is not valid	perror, strerror
EBADKEYLN	The key length specified is not valid	_Rreadk, _Rlocate
EBADMODE	The file mode specified is not valid	fopen, freopen, _Ropen
EBADNAME	Bad file name specified	fopen, freopen, _Ropen
EBADPOS	The position specified is not valid	fsetpos
EBADSEEK	Bad offset for a seek operation	fgetpos, fseek
EBUSY	The record or file is in use	perror, strerror
EDOM	Domain error in math function	acos, asin, atan2, cos, exp, fmod, gamma, hypot, j0, j1, jn, y0, y1, yn, log, log10, pow, sin, strtol, strtoul, sqrt, tan
EGETANDPUT	An illegal read operation occurred after a write operation	fgetc, fread, getc, getchar
EINVAL	The signal is not valid	signal
EIO	Consecutive calls of I/O occurred	I/O
EIOERROR	A nonrecoverable I/O error occurred	All I/O functions

Table 26 (Page 2 of 3). Erno Macros

Error Macro	Description	Set by Function
EIORECERR	A recoverable I/O error occurred	All I/O functions
ENODEV	Operation attempted on a wrong device	fgetpos, fsetpos, fseek, ftell, rewind
ENOENT	File or library is not found	perror, strerror
ENOPOS	No record at specified position	fsetpos
ENOREC	Record not found	fread, perror, strerror
ENOTDLT	File is not opened for delete operations	_Rdelete
ENOTOPEN	File is not opened	clearerr, fclose, fflush, fgetpos, fopen, freopen, fseek, ftell, setbuf, setvbuf, _Ropen, _Rclose
ENOTREAD	File is not opened for read operations	fgetc, fread, ungetc, _Rread, _Rreadf, _Rreadindv, _Rreadk, _Rreadl, _Rreadn, _Rreadnc, _Rreadp, _Rreads, _Rlocate
ENOTUPD	File is not opened for update operations	_Rrslck, _Rupdate
ENOTWRITE	File is not opened for write operations	fputc, fwrite, _Rwrite, _Rwrited, _Rwriterd
ENUMMBS	More than one member	ftell
ENUMRECS	Too many records	ftell
EPAD	Padding occurred on a write operation	fwrite
EPERM	Insufficient authorization for access	perror, strerror
EPUTANDGET	An illegal write operation occurred after a read operation	fputc, fwrite, fputs, putc, putchar
ERANGE	Range error in math function	cos, cosh, gamma, exp, j0, j1, jn, y0, y1, yn, log, log10, ldexp, pow, sin, sinh, strtod, strtol, strtoul, tan
ERECIO	File is opened for record I/O, so character-at-a-time processing functions cannot be used	fgetc, fgetpos, fputc, fread, fseek, fsetpos, ftell
ESTDERR	stderr cannot be opened	fgetpos, fputc, fseek, fsetpos, ftell, fwrite
ESTDIN	stdin cannot be opened	fgetc, fgetpos, fread, fseek, fsetpos, ftell
ESTDOUT	stdout cannot be opened	fgetpos, fputc, fseek, fsetpos, ftell, fwrite

Table 26 (Page 3 of 3). Errno Macros

Error Macro	Description	Set by Function
ETRUNC	Truncation occurred on I/O operation	Any I/O function that reads or writes a record sets <code>errno</code> to ETRUNC

Table 27 describes the settings that are possible when using integrated-file-system-enabled stream I/O.

Table 27 (Page 1 of 2). Errno Values for Integrated-File-System-Enabled C Stream I/O

C Stream Function	Possible Errno Values
<code>clearerr</code>	EBADF
<code>fclose</code>	EAGAIN, EBADF, EIO, EUNKOWN
<code>feof</code>	EBADF
<code>ferror</code>	EBADF
<code>fflush</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fgetc</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>fgetpos</code>	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
<code>fgets</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>fopen</code>	EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR
<code>fprintf</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fputc</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fputs</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fread</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>freopen</code>	EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR
<code>fscanf</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>fseek</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN
<code>fsetpos</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN

<i>Table 27 (Page 2 of 2). Errno Values for Integrated-File-System-Enabled C Stream I/O</i>	
C Stream Function	Possible Errno Values
ftell	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
fwrite	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
getc	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
getchar	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
gets	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
perror	EBADF
printf	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
putc	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
putchar	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
puts	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
remove	EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROOBJ, EUNKNOWN, EXDEV
rename	EACCES, EAGAIN, EBADNAME, EBUSY, ECONVERT, EDAMAGE, EEXIST, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOTEMPTY, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EMLINK, EPERM, EUNKNOWN, EXDEV
rewind	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN
scanf	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
setbuf	EBADF, EINVAL, EIO
setvbuf	EBADF, EINVAL, EIO
tmpfile	EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, EPERM, EROOBJ, EUNKNOWN, EXDEV
tmpnam	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOENT, ENOSYSRSC, EUNATCH, EUNKNOWN
ungetc	EBADF, EIO
vfprintf	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
vprintf	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
Note: Errno is defined in <errno.h> in the include directory stdincl.	

Checking the AS/400 System Exceptions for C Stream Files

In addition to checking the return value of a function or the `errno` value, you can check the major and minor return codes to detect C stream file errors. If your program processes display files, ICF files, or printer files as C stream files, you can check the external variable `_C_Maj_Min_rc`, which is defined in `<stdio.h>`. The definition of this structure is:

```
typedef struct _Major_Minor_rc
{
    char major_rc[2];
    char minor_rc[2];
} _Major_Minor_rc;

extern _Major_Minor_rc _C_Maj_Min_rc;
```

The *Data Management* contains information about major and minor return codes.

Stream I/O functions trap the `SIGIO` signal.

Note: Signal handlers registered for `SIGIO` are not called for exceptions generated when processing stream files.

These `errno` macros indicate an AS/400 system exception:

```
EIOERROR: a nonrecoverable I/O error has occurred
EIORECERR: a recoverable I/O error has occurred
```

The global variable `_EXCP_MSGID` is set whenever a stream or record I/O function gets an exception. The global variable `_EXCP_MSGID`, declared in the `<stddef.h>` header file, contains the exception message ID. See Table 28 on page 323 for information about the `_EXCP_MSGID` setting after an OS/400 exception.

AS/400 Exception Handling

This program shows a number of ways to handle errors, including checking the major and minor return code, checking `errno`, getting error information from the `_EXCP_MSGID` global variable, and signal handling with the `signal()` function.

The DDS source for the display file is:

```
A          DSPSIZ(24 80 *DS3)
A          INDARA
A          R PHONE
A          CF03(03 'EXIT')
A          CF05(05 'REFRESH')
A          7 28'Name:'
A          NAME          11A B 7 34
A          9 25'Address:'
A          ADDRESS      20A B 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A B 11 34
A          1 35'PHONE BOOK'
A          DSPATR(HI)
```

```

A          16 19'<ENTER> : Saves changes'
A          17 21'f3   : Exits with changes saved'
A          18 21'f5   : Brings back original field
A          values'
A 05      21 32'Screen refreshed'
A 05          DSPATR(HI)

```

The program source is:

```

// This program illustrates how to:
//      - check the major and minor return codes
//      - check the errno global variable
//      - get error information from the
//      _EXCP_MSGID global variable
//      - use the signal function.

#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <signal.h>
#include <recio.h>

#define IND_ON '1'
#define IND_OFF '0'
#define HELP 0
#define EXIT 2
#define REFRESH 4
#define FALSE 0
#define TRUE 1

typedef struct PHONE_LIST_T{
    char name[11];
    char address[20];
    char phone[8];
}PHONE_LIST_T;

void error_check(void);

// The error-checking routine.

void error_check(void)
{
    if (errno == EIOERROR || errno == EIORECERR)
        cout << _EXCP_MSGID << endl;
    if (strncmp(_Maj_Min_rc.major_rc,"00",2) ||
        strncmp(_Maj_Min_rc.minor_rc,"00",2))
        cout << "Major : " << _Maj_Min_rc.major_rc
            << "Minor : " << _Maj_Min_rc.minor_rc << endl;
    errno = 0;
}

```

```

// The signal handler routine.

void sighd(int sig)
{
    signal(SIGALL,&sighd);
}

// M A I N   L I N E

int main(void)
{
    FILE *dspf;
    PHONE_LIST_T phone_inp_rec,
                phone_out_rec = { "Smith, Joe",
                                "2711 Wests Rd.  ",
                                "5559729" };
    _SYSindara indicator_area;

    int ret_code;
    errno = 0;
    signal(SIGALL,&sighd); /* Register sighd as a handler for all exceptions */

    if ((dspf = fopen("MYLIB/T2123DDJ", "ab+ type=record indicators=y"))
        == NULL)
    {
        cout << "Display file could not be opened" << endl;
        exit(1);
    }

    _Rindara((_RFILE *) dspf,indicator_area);
    _Rformat((_RFILE *) dspf,"PHONE");

    memset(indicator_area,IND_OFF,sizeof(indicator_area));

    do
    {
        ret_code = fwrite(&phone_out_rec,1,sizeof(phone_out_rec),dspf);
        error_check(); // Write the records to the display file.
        ret_code = fread(&phone_inp_rec,1,sizeof(phone_inp_rec),dspf);
        error_check(); // Read the records from the display file.
        if (indicator_area[EXIT] == IND_ON)
            phone_inp_rec = phone_out_rec;
    }

    while (indicator_area[REFRESH] == IND_ON);
    _Rclose((_RFILE *)dspf);
}

```

The output is:

```

PHONEBOOK
Name: Smith, John
Address: 2711 Wests Rd.
Phone #: 721-9729
<ENTER> : Saves changes
f3      : Exits with changes saved
f5      : Brings back original field values

```

Record Input and Output Error Macro to Exception Mapping

Table 28 describes what occurs if the signal SIGIO is raised. It also describes the errno settings and the _EXCP_MSGID setting after an AS/400 system exception. This is only for *ESCAPE, *NOTIFY and *STATUS messages.

<i>Table 28 (Page 1 of 2). Record Input and Output Error Macro to Exception Mapping¹</i>		
Description	Messages	errno setting
*STATUS and *NOTIFY	CPF4001 to CPF40FF, CPF4401 to CPF44FF, CPF4901 to CPF49FF	errno is not set, a default reply is returned to the AS/400 operating system.
Recoverable I/O error	CPF4701 to CPF47FF, CPF4801 to CPF48FF, CPF5001 to CPF50FF	EIORECERR
Nonrecoverable I/O error ²	CPF4101 to CPF41FF, CPF4201 to CPF42FF, CPF4301 to CPF43FF, CPF4501 to CPF45FF, CPF4601 to CPF46FF, CPF5101 to CPF51FF, CPF5201 to CPF52FF, CPF5301 to CPF53FF, CPF5401 to CPF54FF, CPF5501 to CPF55FF, CPF5601 to CPF56FF	EIOERROR
Truncation occurred at I/O operation	C2M3003	ETRUNC
File is not opened	C2M3004	ENOTOPEN
File is not opened for read operations	C2M3005	ENOTREAD
File is not opened for write operations	C2M3009	ENOTWRITE
Bad file name specified	C2M3014	EBADNAME
The file mode specified is not valid	C2M3015	EBADMODE
File is not opened for update operations	C2M3041	ENOTUPD
File is not opened for delete operations	C2M3042	ENOTDLT
The key length specified is not valid	C2M3044	EBADKEYLN

Table 28 (Page 2 of 2). Record Input and Output Error Macro to Exception Mapping¹

Description	Messages	errno setting
A nonrecoverable I/O error occurred	C2M3101	EIOERROR
A recoverable I/O error occurred	C2M3102	EIORECERR
<p>Note:</p> <p>¹ Since the error is percolated to you, your direct monitor handlers, ILE condition handlers, and signal handler may get control. The initial setting for SIGIO is SIG_IGN.</p> <p>² The type of device determines whether the error is recoverable or nonrecoverable. These IBM publications contain information about recoverable and nonrecoverable AS/400 system exceptions for each specific file type:</p> <ul style="list-style-type: none"> <i>ICF Programming</i> <i>ADTS/400: Advanced Printer Function</i> <i>Application Display Programming</i> <i>SQL Database Programming</i> <i>Tape and Diskette Device Programming</i> 		

Checking the AS/400 System Exceptions for C Record Files

To check the return value of a function or the `errno` value, you can check the values in the `_RIOFB_T` structure, defined in `<recio.h>`, to detect C record file errors. If your program processes display files, ICF files, or printer files as C record files, you can check the `num_bytes` field in the `_RIOFB_T` structure, and the major and minor return code fields in the `sysparm` area of the `_RIOFB_T` structure. If your program processes database files as C stream files, you can check the `num_bytes` field in the `_RIOFB_T` structure, which is defined in `<recio.h>`.

The definition of the `_RIOFB_T` structure is:

```
typedef struct {
    unsigned char    *key;
    _Sys_Struct_T   *sysparm;
    unsigned long    rrn;
    long             num_bytes;
    short           blk_count;
    char            blk_filled_by;
    int             dup_key    : 1;
    int             icf_locate: 1;
    int             reserved1  : 6;
    char            reserved2[20];
} _RIOFB_T;
```

The `num_bytes` field and the `sysparm` field contain information regarding record file I/O errors.

The `num_bytes` field indicates whether the I/O operation was successful. See the descriptions of the C record I/O functions in *C Library Reference* to find out the value to which the `num_bytes` field is set by each record I/O function.

The `sysparm` field points to a structure that contains the major and minor return codes for display files, ICF files or printer files. The definition of `_Sys_Struct_T` structure is:

```
typedef struct {          // System specific information
    void          *sysparm_ext;
    _Maj_Min_rc_T _Maj_Min;
    char          reserved1[12];
} _Sys_Struct_T;
```

The definition of `_Maj_Min_rc_T` structure is:

```
typedef struct {
    char major_rc[2];
    char minor_rc[2];
} _Maj_Min_rc_T;
```

Chapter 15. Porting Programs to VisualAge for C++ for AS/400

This section describes the differences between ILE C and ILE C++ on AS/400. It also describes differences between C++ on Windows and C++ on AS/400. Some examples are provided to illustrate how to port code.

Note: If you are developing new code, then follow the ANSI guidelines as much as possible and avoid platform-specific extensions. In general, your code should be portable. Platform specific extensions, for example, `_Far16`, `_Pascal16` are platform-specific pointers that are not portable.

This section presents the following topics:

“Differences Between C and C++ on AS/400”

“Differences Between C++ on Windows and C++ on AS/400” on page 343

Differences Between C and C++ on AS/400

This section describes the differences between C and C++ on AS/400.

Inter-Language Calls

ILE C source code containing inter-language calls must be modified to run under VisualAge for C++ for AS/400. The `extern` linkage specification with the function definition or declaration must be used instead of the **#pragma linkage** or **#pragma argument** directives. The **#pragma map** directive has some semantic differences.

There is a difference in the way the **#pragma argument** directive and the `extern` linkage specification handle function definitions. Both generate the same code when processing a function call but the **#pragma argument** directive does not affect parameters within the function definition. The `extern` linkage specification does affect parameters within the function definition.

These two modules in ILE C

Module1.C

```
extern void foo (int *i, char **s)
{
    *s = *i ? "Not Zero" : "Zero";
}
```



```

Module2.C

extern void foo (int, char *)

#pragma argument (foo, VREF)

int main()

{
    char *s;
    foo (1, s);
}

```

are equivalent to this ILE C++ code:

```

extern "VREF" void foo (int i, char *s)
{
    s = i ? "Not Zero" : "Zero";
}

int main()
{
    char *s;
    foo (1, s);
}

```

This code shows how `extern "OS"` with a function definition is used to replace the **#pragma linkage** directive. See Chapter 12, "Calling Conventions" on page 193 for additional information. Instead of using

```

typedef void (FUNC)(int);
#pragma linkage (FUNC, OS)

```

you should use

```

extern "OS" typedef void (FUNC) (int); // works

not

typedef void (FUNC)(int)
extern "OS" FUNC; //error

```

Binary Coded Decimal Class Library for OS/400

The Binary Coded Decimal Class Library for OS/400 is provided so that you can create binary coded decimal objects that are compatible with the packed decimal data types on AS/400. To move the ILE C packed decimal data type to the `_DecimalT` class template you need to consider some differences between the two.

Macros Defined in `bcd.h`

These macros are used by the `_DecimalT` class template to maintain compatibility with ILE C:

```

#define decimal    _Decimal
#define digitsof  __digitsof
#define precisionof __precisionof
#define _Decimal(n,p) _DecimalT<n,p>
#define __digitsof(DecName) (DecName).DigitsOf()
#define __precisionof(DecName) (DecName).PrecisionOf()

```

This program shows the source to code a packed decimal data type in ILE C:

```

// ILE C program

#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

This program shows the same source written in C++:

```

// C++ program

#include <bcd.h>

void main()
{
    int dig, prec;
    _DecimalT<9,3> d93;
    dig = d93.DigitsOf();
    prec = d93.PrecisionOf();
}

```

This program shows you that by using the macros defined in <bcd.h>, you can use the same ILE C shown in the first program:

```

// C++ program using the macro

#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

Note: The <decimal.h> header file is specified since <decimal.h> includes the <bcd.h> header file.

Member of a Union

Since an object of a class with a constructor cannot be a member of a union, the `_DecimalT` class template in ILE C++ cannot be used as a member of a union.

Member of a Structure

In ILE C++ if a `_DecimalT` class template is a member of a `struct`, that `struct` cannot be initialized with an initializer list. The structure in ILE C is:

```
typedef struct {
    char      s1;
    decimal(5,3) s2;
}s_type;

s_type s ={'+', 12.345d};
```

In ILE C++ you need to rewrite the code as follows:

```
struct s_type {
    char s1;
    decimal(5,3) s2;
    s_type (char c, decimal(5,3) d ) : s1(c), s2(d) {}
};
s_type s ('+', __D("12.345")) ;
```

Decimal Constant

The decimal constant defined using the suffix `D` or `d` is not supported by the C++ `_DecimalT` class template. Instead, a string literal embraced by `__D` is used to represent a packed decimal constant. The decimal constant `123.456D` defined in ILE C equivalent to `__D("123.456")` in ILE C++.

Decimal Constant and Case Statements

The `__D` macro is used to simplify code that requires the frequent use of the `_ConvertDecimal` constructor. Because the `__D` macro is equivalent to the `_ConvertDecimal` constructor, the `__D` macro cannot be used with a case statement. A valid case statement uses an integral constant expression. This code results in a compiler error:

```
decimal(4,3) op;

switch int(op) {
    case int(__D("1.3")):
        ....
        break;
}
```

The compiler flags the case statement indicating that the case expression is not an integral constant expression.

_DecimalT Class Template Specifiers

C++ uses these macros:

```
#define decimal    _Decimal
#define _Decimal(n,p) _DecimalT<n,p>
```

to map the template class instantiation to the desired ILE C syntax. ILE C code using the `decimal(n,p)` specifier can be ported to C++ without any modification. The second type specifier supported by ILE C is not supported by the C++ compiler:

```
decimal (constant-expression)
```

In this case, you need to insert a zero explicitly at the type specifier:

change `decimal(10)` to `decimal(10,0)`

Error Checking

In ILE C packed decimal is implemented as a native data type. This allows an error such as invalid decimal format to be detected at compile time. In C++ detection of a similar error is deferred until run time:

```
#define _DEBUG 1

#include <bcd.h>

void main()
{
    _DecimalT<10,20> b= __D("ABC"); // Run-time exception is raised
}

and

#define _DEBUG 1

#include <bcd.h>

void main()
{
    _DecimalT<33,2> a;           // Max. dig. allow is 31. Again,
                                // run-time exception is raised
}
```

Some errors can occur at compile time. When `n<1`, (`_Decimal<-33,2>`) then the error is detected at compile time.

Mathematical Operators: ILE C provides additional error checking on the sign or the digit codes of the packed decimal operand. Valid signs are hex A-F and the valid digit range is hex 0-9. If the decimal operand is in error, ILE C generates an error message. This additional checking is not present in the `_DecimalT` template class. This code results in an error message in ILE C but not in ILE C++.

```

#include <decimal.h>

void main()
{
    _Decimal(10,2) a, b;
    int c;
    c = a > b; // a and b are not valid packed decimals because
              // a and b are not initialized
}

```

Extra Precision

The `_DecimalT` class template allows a maximum of 62 and 93 digits as the internal results for the multiplication and division operations respectively. This is different from the ILE C packed decimal data type in which a maximum of 31 digits is used for both operations.

This internal result is different from the intermediate result. The internal result is designated to store the temporary result during the operation. After the operation is completed, the internal result is converted to the intermediate result and returned to the caller.

Header File

In ILE C, the header file `<decimal.h>` must be included in the source prior any usage of the packed decimal data type. In ILE C++ `<bcd.h>` must be included instead.

Digits of an Object

In ILE C, when you use the `__digitsof` operator with a packed decimal data type the result is an integer constant. The `__digitsof` operand can be applied to a packed decimal data type or a packed decimal constant expression. The `__digitsof` operator returns the total number of digits *n* in a packed decimal data type.

To determine the number of digits *n* in a packed decimal data type:

```

#include <decimal.h>

int n,n1;
decimal (5, 2) x;

n = __digitsof(x);           /* the result is n=5 */
n1 = __digitsof(1234.567d); /* the result is n1=7 */

```

In ILE C++ when you use the member function `DigitsOf()` with a `_DecimalT` class template the result is an integer constant. The member function `DigitsOf()` can be applied to a `_DecimalT` template class object. The member function `DigitsOf()` returns the total number of digits *n* in a `_DecimalT` template class object.

To determine the number of digits *n* in a `_DecimalT` template class object:

```

#include <bcd.h>

int n,n1;
_DecimalT <5, 2> x;

n = x.DigitsOf();           // the result is n=5

```

Precision of an Object

When you use the `__precisionof` operator with a packed decimal data type the result is an integer constant. The `__precisionof` operand can be applied to a packed decimal data type or a packed decimal constant expression. The `__precisionof` operator tells you the number of decimal digits p of the packed decimal data type.

To determine the number of decimal digits p of the packed decimal data:

```

#include <decimal.h>

int p,p1;
decimal (5, 2) x;

p=__precisionof(x);           /* The result is p=2 */
p1=__precisionof(123.456d);   /* The result is p1=3 */

```

In ILE C++ when you use the member function `PrecisionOf()` with a `_DecimalT` class template the result is an integer constant. The member function `PrecisionOf()` can be applied to a `_DecimalT` template class object. The member function `PrecisionOf()` tells you the number of decimal digits p of the `_DecimalT` template class object.

To determine the number of decimal digits p of the `_DecimalT` class template object:

```

#include <bcd.h>

int p,p1;
_DecimalT <5, 2> x;

p=x.PrecisionOf();           // The result is p=2

```

Using Formatted C Input and Output Functions

The behavior of the `fprintf()`, `sprintf()`, `vfprintf()`, `vprintf()` and `vsprintf()` functions is the same as the `printf()` function. The behavior of the `fscanf()` and `sscanf()` functions is the same as the `scanf()` function.

To control the format of the printout use the *flags*, *width* and *precision* fields of the `printf()` function.

To control the format of the `scanf()` function use the *** and *width* fields of the `scanf()` function. See the *C Library Reference* for information on these fields.

Print Function Flags: Table 29 on page 334 describes the flag characters and their meanings for $D(n,p)$ conversions.

Table 29. Flag Meanings for Printing the Value of a `_DecimalT` Template Class Object

Flag	Meaning
#	The result always contains a decimal-point character, even if no digits follow it.
0	Leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed.
-	The result is always left-justified within the field.
+	The result always begins with a plus or minus sign.
space	The result is always prefixed with a space where the result of a signed conversion is no sign or the signed conversion results in no characters.

Print Function Field Width: The optional minimum field width for the `printf()` function indicates the minimum number of digits to appear in the integral part, fractional part or both parts of a `_DecimalT` template class object. If there are fewer characters than the field width, then the field is padded with spaces. The field width can be an `*`. If `n` is an `*`, the value of `n` is derived from the corresponding position in the parameter list.

Print Function Field Precision: The optional precision for the `printf()` function indicates the number of digits to appear in the fractional part of a `_DecimalT` template class object. The default precision is `p`. If precision is greater than `p`, extra zeros are padded. If precision is less than `p`, rounding is performed. The field precision can be an `*`. If `p` is an `*`, the value of `p` is derived from the corresponding position in the parameter list.

Conversion Specifiers: The conversion specifier for the `printf()` function is:

`D(n,p)` The `_DecimalT` template class object is converted in the style `[-] ddd.ddd` where the number of digits after the decimal-point character is equal to the precision of the specification. If the precision is missing, it is taken as 0; if the precision is zero and the `#flag` is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is truncated to the appropriate number of digits. The `(n,p)` descriptor is used to describe the characteristic of the `_DecimalT` template class object. Both `n` and `p` have to be in the form of decimal integers. If `p` is missing, a default value of zero is assumed. If the specifier is in another form not stated above, the behavior is undefined.

If `n` and `p` of the variable to be printed do not match with the `n` and `p` in the conversion specifier `%D(n,p)`, the behavior is undefined. Use the unary operators `__digitsof` (`expression`) and `__precisionof` (`expression`) in the argument list to replace the `*` in `D(*,*)` whenever the size of the resulting class of a `_DecimalT` template class object expression is not known.

The conversion specifier for the `scanf()` function is as follows:

`D(n,p)` Matches a decimal number, the expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal point. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject

sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

Conditional Operator

In C++ both the second and third expressions must be of the same class. In ILE C if both the second and third operands have an arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. In C++ if either expression has a different class, then you must cast the second or third expression so that it has the same class. The following illustrates an example where the conditional expression fails because the second and third expressions are not of the same class.

```
#include <bcd.h>

main()
{
  int var_1;
  decimal(4,2) op_1_1 = __D("12.34");
  decimal(10,2) op_1_2 = __D("123.45");
  var_1 = (op_1_1 < op_1_2) ? (op_1_1 + 3) : op_1_2;
}
```

To use the conditional operator with the `_DecimalT` template class you can do one of the following:

- Use an explicit cast on the second expression so that it has the same type as the third expression

- Use the same type of variables

This program shows an explicit cast on the second expression so that it has the same class as the third expression:

```
#include <bcd.h>

main()
{
  int var_1;
  decimal(4,2) op_1_1 = __D("12.34");
  decimal(10,2) op_1_2 = __D("123.45");
  var_1 = (op_1_1 < op_1_2) ? (_DecimalT<10,2>).__D(op_1_1 + 3) : op_1_2;
}
```

This program shows how to use the same type of variables:


```

#include <bcd.h>

main()
{
int var_1;
decimal(10,2) op_1_1 = __D("12.34");
decimal(10,2) op_1_2 = __D("123.45");
var_1 = (op_1_1 < op_1_2) ? op_1_1 : op_1_2;
}

```

Note: The + 3 was removed from the second expression since (op_1_1 + 3) results in `_Decimal<13,2>`.

Porting ILE C Packed Decimal Data Type to `_DecimalT` Class Template

In the class template `_DecimalT`, neither the constructor nor the assignment operator are overloaded to take any of the template class instantiated from `_DecimalT`. For this reason, explicit type casting that involves conversion from one `_DecimalT` template class object to another cannot be done directly. Instead, the macro `__D` must be used to embrace the expression that requires explicit type casting. This program is written in ILE C:

```

#include <decimal.h>

void main ()
{
decimal(4,0) d40 = 123D;
decimal(6,0) d60 = d40;
d60 = d40;
decimal(8,0) d80 = (decimal(7,0))1;
decimal(9,0) d90;
d60 = (decimal(7,0))12D;
d60 = (decimal(4,0))d80;
d60 = (decimal(4,0))(d80 + 1);
d60 = (decimal(4,0))(d80 + (float)4.500);
}

```

This source needs to be rewritten as:

```

#include<bcd.h>

void main ()
{
    _DecimalT<4,0> d40 = __D("123"); // OK
    _DecimalT<6,0> d60 = __D(d40); // Because no constructor
    // exists that can convert d40 to d60.
    // macro __D is needed to convert d40
    // into an intermediate type first.
    d60 = d40; // OK. This is different from the
    // previous statement in which
    // the constructor was called.
    // In this case, the assignment
    // operator is called and the
    // compiler converts d40 into the
    // intermediate type automatically.
    _DecimalT<8,0> d80 = (_DecimalT<7,0>)1;
    // OK
    // Type casting an int, not a decimal(n,p)

    _DecimalT<9,0> d90; // OK
    d60 = (_DecimalT<7,0>).__D("12"); // OK
    d60 = (_DecimalT<4,0>).__D(d80);
    d60 = (_DecimalT<4,0>).__D(d80 + 1);

    // In both cases, the resultant classes
    // of the expressions are _DecimalT<n,p>.
    // macro __D is needed to convert them
    // to an intermediate type first.

    d60 = (_DecimalT<4,0>)(d80 + (float)4.500);
    // OK because the resultant type
    // is a float
}

```

Header Files that Work with Both C and C++

C header files are not generally usable by C++. Structures, unions and type definitions may be all right as well as variables. Care must be used in function prototypes and pragmas.

C/C++ Enablement

Wrap your header files in the following construct:

```

.
.
.
#ifdef __cplusplus
    extern "C" {
        #pragma info(none)
    }
#else
    //only if you have #pragma
    #pragma nomargins nosequence //nomargin and #pragma checkout in the
    #pragma checkout(suspend) //header file
#endif
.
.
.
#ifdef __cplusplus
    #pragma info(restored)
    {
    }
#else
    #pragma checkout(resume)
#endif
}

```

The linkage specification `extern "C"` informs the compiler that all functions prototyped will have C linkage. C++ linkage functions cannot be called from C using the C++ internal name. See Chapter 12, "Calling Conventions" on page 193 for information on calling functions from other languages.

The macro `__ILEC400__` can replace `__cplusplus` but `__cplusplus` is preferred because it is portable to other implementations.

Packed Structures

The `_Packed` keyword tells the compiler to ignore the padding and pack the structure as much as possible. In ILE C this keyword can be used in a structure definition and typedef. The same keyword can only be used in a typedef in the C++ compiler.

<i>Table 30. Comparing Packed Structures</i>		
	ILE C/400	ILE C++
<pre>typedef _Packed struct { . . }ps_t;</pre>	ok	ok
<pre>_Packed struct { . . }ps_v;</pre>	ok	error

Therefore, you must make sure the `_Packed` keyword is only used in typedefs in the header file.

In VisualAge for C++ for AS/400 there is only one **#pragma pack** directive and that is for AS/400. The AS/400 **#pragma pack** directive is not compatible with the Windows **#pragma pack** directive.

Function Prototype

To allow your header files to be used by ILE C and ILE C++ compilers, all functions with "OS" linkage type must be dual prototyped:

```
#ifdef __cplusplus
    extern "linkage-type" //linkage type "OS"
#else
    #pragma linkage(function_name, linkage_type)
#endif
void function_name(...);
```

If you have a list of functions that need dual prototypes, you can:

```
#ifdef __cplusplus
    extern "linkage-type" { //linkage type "OS"
#else
    #pragma linkage(function_name1, linkage_type)
    .
    .
    #pragma linkage(function_nameN, linkage_type)
#endif
void function_name1(...);
.
.
void function_nameN(...);
#ifdef __cplusplus
}
#endif
```

Enum Data Type Size

In ILE C++ the enum size is always the size of an integer unless the **#pragma enumsize** is used. If the size of the enum type is critical, the following code can be used to resolve the problem:

```
.
.
#pragma enumsize (2)
enum { a=0xffff} A; //sizeof(A)=2;
#pragma enumsize ()
.
.
```

Including QSYSINC Header Files

To use the QSYSINC header files in ILE C++ you need to use the following convention `#include <file/header.h>`. To include QSYSINC/MIH/SYSEPT you can use `#include <mih/sysept.h>`

This form also works for ILE C.

Type Checking

Type checking is stricter in C++ than it is in C. C++ allows `void` pointers to be assigned only to other `void` pointers. In ANSI C, a pointer to `void` can be assigned to a pointer of any other type without an explicit cast. See the *C++ Language Reference* for information on compatibility.

Assume some source uses `memcmp()` to compare a constant `char` array to a volatile `char` array. In ILE C this code compiles without errors. Using `iccas` the same code generates error message CTT3055: Volatile unsigned char cannot be converted to a const void pointer. You cannot use a constant pointer where a volatile pointer is expected unless you cast it. You can cast a `void` pointer to the appropriate pointer type before compiling the code.

C memory functions such as `malloc()`, `calloc()`, `realloc()`, that return `void` pointers must be cast to an appropriate pointer type before compiling the code. You can use the `new` and `delete` operators instead of `malloc()` and `free()` to take advantage of C++.

Name Mangling

All C++ function names are mangled to enable function overloading. You receive an undefined names error when you bind ILE C functions with mangled names, for example, `LocateSpaces__FPc`. In ILE C the service program name is `LocateSpace`.

If you are porting ILE C code and you want to stop function name mangling then use `extern "C"` around the function name.

File Inclusion

File inclusion follows Windows include file processing. The include file name must be a valid workstation file name, for example `"file_name"` or `<file_name>`. Include files cannot reference `*LIBL` or `*CURLIB` values. You can use these values in ILE C include names. For example, `("*LIBL/ABC", *LIBL/ABC/*All) "...)`. The include name `"*LIBL/ABC"` is not valid in Windows.

If you use a back slash (`\`) character in your include file name, you must use a double back slash (`\\`). For example, `"c:\\abc\\aaa.h"` for a fully qualified path name or `"proj1\\aaa.h"` for a relative path include name. See the *C++ Language Reference* for information on file inclusion.

Function Prototypes, Declarations and Pointers

C++ requires full prototype declarations. ANSI C allows non-prototyped functions. See the *C++ Language Reference* for information on function declarations.

In C++, all declarations of a function must match the unique definition of a function. ANSI C has no such restriction.

The type defined in a pointer declaration must match the type defined in the function prototype in C++. You can assign a pointer to a different type than what is defined in the function prototype by using an explicit cast expression:

```

void (*oldsig) (); // void pointer "void (*) ()"
extern "C" void (*signal (int, void(*) (int))) (int): // function
                                                    // prototype
                                                    // void(*) (int)
                                                    // returned

oldsig = signal (SIGALL, sig_handler);

```

CTT3055: "extern "C" void(*) (int) cannot be converted to "void (*) ()" results from the type mismatch between the type defined in void pointer ("void(*) ()") and the type defined in the function prototype (void (*) (int)). The *int* parameter does not exist in the function *oldsig* or the void pointer declaration.

In another example, QXX functions return unsigned char pointers. ILE C allows you to assign a signed char to an unsigned char pointer. This is not valid in C++. Unsigned char pointers must be declared as unsigned char variables in the source code as shown in the example source:

```

#include <xxcvt.h> //void(QXXITOP(unsigned char *pptr, int digits, int
                    //fraction, int value);

#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;
    QXXITOP (pptr, digits, fraction, value);
}

```

You can assign the pointer to a variable of a different type by using a cast expression. You can also use the /J+ compiler option when you compile the C++ source to set the default char type. See the *C++ User's Guide* for more information on these options.

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 C, space for the trailing '\0' can be omitted in this type of information. The following initialization, for instance, is not valid in C++:

```

char v[3] = "asd"; //not valid in C++, valid in ANSI 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). The following initialization, for instance, is valid in C++:

```

char v[4] = "asd"; //valid in C++

```

See the *C++ Language Reference* for information on compatibility.

String Literals

In ILE C strings are placed into read/write memory by default. In ILE C++ you must use the **#pragma strings** directive to place strings into read/write memory. The syntax of this pragma is:

```
                writeable
# pragma strings (  readonly  )
```

C strings are read/write by default. C++ strings are read only by default. This pragma must appear before any C or C++ code in a file.

Integrated File System

The integrated file system provides a common interface to store and operate on information in stream files. The C stream I/O functions and the C++ stream I/O classes are implemented through the integrated file system. There are seven file systems in the integrated file system. The library (QSYS.LIB) file system maps to the AS/400 file system but using this system under the integrated file system presents some limitations:

- Logical files are not supported

- The only types of physical files supported are program-described files containing a single field and source physical files containing a single text field

- Byte-range locking is not supported; see the *System API Reference*

- If any job has a database file member open, only one job is given write access to that file at any time; other jobs are allowed only read access

The C and C++ stream I/O default is integrated file system enabled using the VisualAge for C++ for AS/400 compiler. The ILE C compiler defaults to C data management stream I/O. If you have programs that use database or DDM files, your best choice is to use the /ASi- compiler option. This ensures that you compile your existing programs using the AS/400 data management file system and not the integrated file system. Compiling programs that use restricted database or DDM files under the integrated file system will result in a run-time error.

Set_Terminate is Scoped to an Activation Group

The effect of the `set_terminate()` function is scoped to an activation group. In the following example both `my_terminate()` and `a()` reside in a service program which runs in activation group "B". `main()` runs in another activation group, for example "A". In this scenario, the thrown exception by `a()` cannot percolate up to `main()`. This uncaught exception cannot cause `my_terminate()` to be invoked. As a result, CEE9901 is sent to `main()`.

```

// File main.c

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <terminat.h>

void a();
void my_terminate();

int main() {
    set_terminate(my_terminate);
    try {
        a();
    }
    catch(...) cout << "failed" << endl;
}

// File term.c

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

void my_terminate() {
    cout << "failed" << endl;
}
void a() { throw 7; }

```

But if the `set_terminate` statement is relocated in a prior to the throw operation, `my_terminate` can then be invoked. This is because the effect of `set_terminate` is scoped to an activation group. Aswell the activation group where `set_terminate` is executed will be terminated (not the entire application, if the application encompasses several activation groups).

Differences Between C++ on Windows and C++ on AS/400

This section describes the differences between C++ on Windows and C++ on AS/400.

IBM Open Class Run-Time Libraries

The V3R6 IBM Open Class run-time library is shipped with VisualAge for C++ for AS/400. Run your existing V3R6 Open Class library programs using the V3R6 run-time library. When you create new V3R7 Open Class library programs use the V3R7 IBM Open Class run-time library which is also shipped with VisualAge for C++ for AS/400. Specify the `INCLUDE_ASV3R6` environment variable and the compiler option `/ASv3r6-` to target an AS/400 V3R6 run-time environment.

Correspondence Between Windows and OS/400 Objects

The following table shows the correspondence between Windows and OS/400 objects:

Windows	OS/400
Object module (.OBJ)	Module object (*MODULE)
Executable (.EXE)	Program object (*PGM)
Dynamic link library (.DLL)	Service program (*SRVPGM)
Library (.LIB)	N/A
Help file (.HLP)	N/A
Directory	Library (*LIB)
Linker	Binder

Inter-Language Calls

VisualAge C++ for Windows code containing inter-language and system calls must be modified before being used with VisualAge for C++ for AS/400 as these are system dependant.

Record Input and Output

You can use record I/O for database access and stream I/O. On AS/400 the database is accessed using record I/O. You can use the DB2/400 database or the AS/400 database. Both can be accessed using record I/O.

File I/O on Windows is stream I/O. Windows does not have a database integrated into the operating system. You can access a database using a database product on Windows.

You can port C++ applications on Windows using stream I/O to AS/400, but you should change the Windows programs to use AS/400 record I/O. This way you realize performance gains and can take advantage of AS/400 extensions.

Abort

The AS/400 implementation for the `abort()` function invokes destructors of all user-defined class objects. This does not happen when you use the Windows `abort()` function.

User Interface and GUI Applications

On AS/400, you cannot use the Windows graphical user interface class libraries. To display panels on an AS/400 display you must write your C++ program to use either Data Description Specifications (DDS), User Interface Manager (UIM), Dynamic Screen Manager (DSM) or User Defined Data Stream(s) (UDDS).

Externally Described Files

C++ supports externally described files through the use of the **#pragma mapinc** directive. This pragma points the compiler to the library and file and tells the compiler the name of the `#include` file to create that will contain the compiler-generated type definition. This `typedef` defines a `struct` which represents the field layout of the file. The name of the `typedef` is determined by the library, file and `record` format name.

This support helps you in that you do not need to code structures by hand which match the database format. The compiler can generate header files which contain structures matching the database format. In addition, the compiler allows you to save the header file whereas ILE C does not allow you to do this.

Operational Descriptors

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. ILE C++ defines a string as a contiguous sequence of characters terminated by and including the first null character. In another language, for example RPG, a string may be defined as consisting of a length identifier and a character sequence. Therefore when passing a string from a C++ function to a function written in another language, for example RPG, an operational descriptor can be provided with the arguments to allow the called function to determine the length and the type of the string being passed.

You can use the **#pragma descriptor** directive to specify the function name and the function's arguments that are to have operational descriptors. The generated descriptor contains the descriptor type, data type and length for each argument identified as requiring an operational descriptor.

Exception Handler

You can use the **#pragma exception_handler** directive to define your own function to be enabled as an ILE exception handler.

Pointers

In applications running on Windows, `long` and `pointer` are often used interchangeably but on AS/400 `pointer` and `long` have different implications.

`pointer` is 16 bytes and `long` is 4 bytes. An AS/400 pointer cannot be assigned to any `integer` type and an `integer` type cannot be assigned to an AS/400 pointer. An assignment of an `integer` to a `pointer` yields `NULL` regardless of the integer value.

Note: Pointer-integer conversions are generally non-portable, not just from Windows to AS/400. Anything that depends on the size of a data type is considered non-portable. While `integer` data types (such as `short` which is 2 bytes and `long` which is 4 bytes) are the same on both the Windows and AS/400 platforms, there may be differences for some other non-integer data types:

Table 32. Non-Integer Data Types

Data Type	Windows	OS/400
short	2 bytes	2 bytes
long	4 bytes	4 bytes ¹
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	10 bytes ²	8 bytes
Note:		
¹ Always defaults to signed unless specified as unsigned.		
² For long double the size is 16 bytes on Windows but floating point data is 10 bytes.		

See Appendix A, “ANSI Notes on Implementation-Defined Behavior” on page 357 for the maximum and minimum boundaries for all data types.

Data types for AIX are the same as those specified in Table 32 for AS/400.

Overloading New and Delete Operators

If you are going to overload `new` or `delete` operators, then be aware that `new` or `delete` should return a pointer to a location which is aligned on a 16 byte boundary. An AS/400 pointer is aligned on a 16 byte boundary.

Header File Names

VisualAge for C++ for AS/400 truncates header files to 8 characters. For example, `strstream.h` is truncated to `strstrea.h` and `stdiostream.h` is truncated to `stdiostr.h`.

File Names

File names must be OS/400-compliant, that is, 10 characters or less. Compiling files with names greater than 10 characters results in a compile time error.

Global Scope Operator Functions

When the global scope operator (`::`) is used in front of C++ run-time functions that are also macros a compile-time error occurs. For example,

```
size_t len= :: strlen(str) +1;
```

Since `strlen()` is a macro, this line is expanded to

```
size_t len= ::(_strlen(str)+1;
```

A compile-time error occurs because `:: (: is incorrect syntax.`

Use this code at the top of the source that used the `::strlen` syntax to avoid this error:

```
#ifdef strlen
#undef strlen
#endif
```

Chapter 16. Casting with Run-Time Type Information

Run-Time Type Information (RTTI) is an extension to the C++ language made by the ANSI/ISO standard committee. You can classify extensions to the language as either minor (those extensions that simply provide a more convenient way of implementing traditional designs) or major (extensions that fundamentally affect the organization of programs). RTTI is a major extension — similar in impact to templates, exceptions and name spaces.

This section describes:

“Introducing RTTI”

“Using C++ Language Defined RTTI” on page 348

“Understanding VisualAge for C++ for AS/400 Extensions to RTTI” on page 352

Introducing RTTI

The RTTI language extension was designed to address the difficulty encountered by builders and users of major C++ libraries. The builders and users needed a mechanism for extending base classes provided in libraries. The RTTI mechanisms implemented by builders of the major C++ libraries were incompatible with each other. This meant it could be difficult to use more than one library for a given program, or to use the same program with different libraries without major changes to the program. Since the C++ language supports the reuse of code and the building of programs from parts, the incompatibilities of the RTTI mechanisms used internally by the various C++ libraries presented a challenge to the purpose of C++. A language-supported mechanism was needed.

There are three parts to RTTI support, each corresponding to increasing involvement and knowledge of the language's RTTI implementation. The part of the mechanism that requires the least involvement and knowledge also serves the majority of needs. This is the part of the RTTI construct that you can focus on — **dynamic_cast**.

The parts of the C++ language support for RTTI are:

dynamic_cast operator

This operator combines type-checking and casting in one operation. It checks whether the requested cast is valid, and performs the cast only if it is valid.

typeid operator

This operator returns the run-time type of an object. If the operand provided to the `typeid` operator is the name of a type, the operator returns an object that identifies it. If the operand provided is an expression, `typeid` returns the type of the object that the expression denotes.

type_info class

This class describes the RTTI available, and is used to define the type returned by the `typeid` operator. This class provides to users the possibility of shaping and extending RTTI to suit their own needs. This ability is of most interest to implementers of object I/O systems such as debuggers or database systems.

Using C++ Language Defined RTTI

To use RTTI you need to be familiar with the *dynamic cast* expression and the **typeid** operator.

The dynamic_cast Operator

A *dynamic cast* expression is used to cast a base class pointer to a derived class pointer. This is referred to as *downcasting*.

The `dynamic_cast` operator makes downcasting much safer than conventional static casting. It obtains a pointer to an object of a derived class that is given a pointer to a base class of that object. The operator **dynamic_cast** returns the pointer only if the specific derived class actually exists. If not, it returns zero.

Dynamic casts have the form:

```
dynamic_cast < type_name > ( expression )
```

The operator converts the *expression* to the desired type *type_name*. The *type_name* can be a pointer or a reference to a class type. If the cast to *type_name* fails, the value of the *expression* is zero.

Dynamic Casts with Pointers

A dynamic cast using pointers is used to get a pointer to a derived class in order to use a detail of the derived class that is not otherwise available:

```
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
    virtual int bonus();
};
```

In this class hierarchy, dynamic casts can be used to include the `manager::bonus()` function in the manager's salary calculation but not in the calculation for a regular employee. The **dynamic_cast** operator uses a pointer to the base class `employee`, and gets a pointer to the derived class `manager` in order to use the `bonus()` member function.

```

void payroll::calc (employee *pe) {
    // employee salary calculation
    if (manager *pm = dynamic_cast<manager*>(pe)) {
        // use manager::bonus()
    }
    else {
        // use employee's member functions
    }
}

```

If `pe` actually points to a `manager` object at run time, the dynamic cast is successful, `pm` is initialized to a pointer to a `manager`, and the `bonus()` function is used. If not, `pm` is initialized to zero and only the functions in the `employee` base class are used.

Note: In the above program, dynamic casts are needed only if the base class `employee` and its derived classes are not available to users (as in part of a library where it is undesirable to modify the source code). Otherwise, adding new virtual functions and providing derived classes with specialized definitions for those functions is a better way to solve this problem.

Dynamic Casts with References

The **dynamic_cast** operator can be used to cast to reference types. C++ reference casts are similar to pointer casts: they can be used to cast from references to base class objects to references to derived class objects.

In dynamic casts to reference types, *type_name* represents a type and *expression* represents a reference. The operator converts the *expression* to the desired type *type_name&*.

Since there is no such thing as a *zero* reference, it is not possible to verify the success of a dynamic cast using reference types by comparing the result (the reference that results from the dynamic cast) with zero. A failing dynamic cast to a reference type throws a **bad_cast** exception.

A dynamic cast with a reference is a good way to test for a coding assumption. The `employee` example above using reference casts is:

```

void payroll::calc (employee &re) {
    // employee salary calculation
    try {
        manager &rm = dynamic_cast<manager&>(re);
        // use manager::bonus()
    }
    catch (bad_cast) {
        // use employee's member functions
    }
}

```

Note: This program is only intended to show the **dynamic_cast** operator used as a test. This example does not demonstrate good programming style, since it uses

exceptions to control execution flow. Using **dynamic_cast** with pointers, as shown above is a better way.

The typeid Operator

The **typeid** operator identifies the exact type of an object that is given a pointer to a base class. It is typically used to gain access to information needed to perform some operation where no common interface can be assumed for every object manipulated by the system. Object I/O and database systems often need to perform services on objects where no virtual function is available to do so. The **typeid** operator enables this.

A **typeid** operation has this form:

```
typeid (type_name expression )
```

The result of a **typeid** operation has type `const type_info&`.

This table summarizes the results of various **typeid** operations.

<i>Table 33. typeid operations</i>	
Operand	typeid Returns
<i>type_name</i>	A reference to a type_info object that represents it.
<i>expression</i>	A reference to a type_info that represents the type of the <i>expression</i> .
Reference to a polymorphic type	The type_info for the complete object referred to.
Pointer to a polymorphic type	The dynamic type of the complete object pointed to. If the pointer is zero, the typeid expression throws bad_typeid exception.
Nonpolymorphic type	The type_info that represents the static type of the <i>expression</i> . The <i>expression</i> is not evaluated.

These examples use the **typeid** operator in expressions that compare the run-time type of objects in the `employee` class hierarchy:

```
// ...
employee *pe = new manager;
employee& re = *pe;
// ...
typeid(pe) == typeid(employee*) // 1. True - not a polymorphic type
typeid(&re) == typeid(employee*) // 2. True - not a polymorphic type
typeid(*pe) == typeid(manager) // 3. True - *pe represents a polymorphic type
typeid(re) == typeid(manager) // 4. True - re represents the object manager

typeid(pe) == typeid(manager*) // 5. False - static type of pe returned
typeid(pe) == typeid(employee) // 6. False - static type of pe returned
typeid(pe) == typeid(manager) // 7. False - static type of pe returned

typeid(*pe) == typeid(employee) // 8. False - dynamic type of expression is manager
typeid(re) == typeid(employee) // 9. False - re actually represents manager
typeid(&re) == typeid(manager*) // 10. False - manager* not the static type of re
// ...
```

In comparison 1, `pe` is a pointer (that is, not a polymorphic type); therefore, the expression `typeid(pe)` returns the static type of `pe`, which is equivalent to `employee*`.

Similarly, `re` in comparison 2 represents the address of the object referred to (that is, not a polymorphic type); therefore, the expression `typeid(re)` returns the static type of `re`, which is a pointer to `employee`.

The type returned by **typeid** represents the dynamic type of the expression only when the expression represents a polymorphic type, as in comparisons 3 and 4.

Comparisons 5, 6, and 7 are false because it is the type of the *expression* (`pe`) that is examined, not the type of the *object* pointed to by `pe`.

These examples do not directly manipulate **type_info** objects. Using the **typeid** operator with built-in types requires interaction with **type_info** objects:

```
int i;
// ...
    typeid(i) == typeid(int)    // True
    typeid(8) == typeid(int)   // True
// ...
```

The class **type_info** is described in “The `type_info` Class.”

The `type_info` Class

To use the **typeid** operator in Run-time Type Identification (RTTI) you must include the C++ standard header `<typeinfo.h>`. This header defines these classes:

type_info Describes the RTTI available to the implementation. It is a polymorphic type that supplies comparison and collation operators, and provides a member function that returns the name of the type represented.

The copy constructor and the assignment operator for the class **type_info** are private; objects of this type cannot be copied.

bad_cast Defines the type of objects thrown as exceptions to report dynamic cast expressions that have failed.

bad_typeid

Defines the type of objects thrown as exceptions to report a null pointer in a **typeid** expression.

Using RTTI in Constructors and Destructors

The **typeid** and **dynamic_cast** operators can be used in constructors or destructors, or in functions called from a constructor or a destructor.

If the operand of **dynamic_cast** used refers to an object under construction or destruction, **typeid** returns the **type_info** representing the class of the constructor or destructor.

If the operand of **dynamic_cast** refers to an object under construction or destruction, the object is considered to be a complete object that has the type of the constructor's or destructor's class.

The result of the **typeid** and **dynamic_cast** operations is undefined if the operand refers to an object under construction or destruction, and if the static type of the operand is not an object of the constructor's or destructor's class or one of its bases.

Understanding VisualAge for C++ for AS/400 Extensions to RTTI

The VisualAge for C++ for AS/400 `extended_type_info` class was designed to provide support for implementing a persistent object store. The basic operations that must be supported are:

- Allocation of memory of an object
- Allocation of memory for an array of objects
- Initial construction of an object
- Initial construction of an array of objects
- Copy construction of an object
- Copy construction of an array of objects

Additional operations for destroy and deallocation of memory are also provided to detect an exception that occurs during construction. These operations are:

- Destruction of an object
- Destruction of an array of objects
- Deallocation of memory for an object
- Deallocation of memory for an array of objects

The `extended_type_info` Class

The class definitions are:

```

class extended_type_info : public type_info {
public:
    ~extended_type_info();

    virtual size_t size() const=0;

    virtual void* create(void* at) const=0; //object
    virtual void* create(void* at, size_t count) const=0; // array

    virtual void* copy (void* to, const void* from) const=0; //object
    virtual void* copy (void* to, const void* from, size_t count) const=0;
    //array

    virtual void* destroy(void* at) const=0; //object
    virtual void* destroy(void* at, size_t count) const=0; //array

    virtual void* allocObject() const=0; //object
    virtual void* allocArray(size_t count) const=0; //array

    virtual void* deallocObject(void* at) const=0; //object
    virtual void* deallocArray(void* at, size_t count) const=0; //array
};

```

Explanation of terms:

size() Size of the type represented by the `extended_type_info` object.

create(void*)

This function is called to create an object of the type represented by the `extended_type_info` object at the storage location pointed to by `at`.

create(void*, size_t)

This function is called to create an array of objects of the type represented by the `extended_type_info` object at the storage location pointed to by `at`. If any exceptions are thrown during construction, `create()` destroys the array elements that were already constructed before rethrowing the exception.

copy(void* to, const void* from)

This function is called to copy an object of the type represented by the `extended_type_info` object into the storage location pointed to by `to`, using the value of the object referred to by `from`.

copy(void* to, const void* from, size_t)

This function is called to copy an array of objects of the type represented by the `extended_type_info` object into the storage location pointed to by `to`, using the value of the object referred to by `from`. If any exceptions are thrown during construction, `copy()` destroys the array elements that were already constructed before rethrowing the exception.

destroy(void*)

This function is called to destroy an object of the type represented by the `extended_type_info` object at the storage location pointed to by `at`.

destroy(void*, size_t)

This function is called to destroy an array of objects of the type represented by the `extended_type_info` object at the storage location pointed to by `at`. If any exceptions are thrown during destruction, `destroy()` destroys the remaining array elements that were already constructed before rethrowing the exception. If another exception is encountered during the destruction of the remaining elements, `destroy()` calls `terminate()`.

allocObject()

This function is called to allocate memory for an object of the type represented by the `extended_type_info` object. No initialization is performed. The user is expected to use the `create(void*)` or `copy(void*, const void*)` function to initialize the new memory.

allocArray(size_t)

This function is called to allocate memory for an array of objects of the type represented by the `extended_type_info` object. No initialization is performed. The user is expected to use the `create(void*, size_t)` or `copy(void*, const void*, size_t)` function to initialize the new memory.

deallocObject(void*)

This function is called to deallocate an object of the type represented by the `extended_type_info` object. No destruction is performed beforehand. The user is expected to use the `destroy(void*)` to terminate the object before deallocating the memory.

deallocArray(void*, size_t)

This function is called to deallocate an array of objects of the type represented by the `extended_type_info` object. No destruction is performed beforehand. The user is expected to use the `destroy(void*, size_t)` to terminate an array before deallocating the memory.

linkageInfo()

This function returns the mangled name of the class type.

Part 5. Appendix

This part contains information on implementation-defined behavior inherited from C, specific implementation defined behavior for C++, considerations for porting code from Windows and ILE C, and the macros supported by VisualAge for C++ for AS/400.

Appendix A. ANSI Notes on Implementation-Defined Behavior

There is no ANSI/ISO standard for the C++ language. An ISO committee is working on a definition. The *Working Paper for Draft Proposed American National Standard for Information Systems - Programming Language C++* ANSI X3J16/95-0001, is the base document for ongoing standardization of the language. The VisualAge for C++ for AS/400 compiler continues to implement extensions and clarifications of the language as they become available.

This section describes how VisualAge for C++ for AS/400 behaves where the American National Standard for Information Systems / International Standards Organization - Programming Language C standard 9899-1990[1992], describes behavior as implementation-defined. These behaviors can affect your writing of portable code.

Implementation-Defined Behavior Inherited from C

These sections describe how the VisualAge for C++ for AS/400 product defines the behavior classified as implementation-defined in the ANSI C Standard.

Identifiers

The number of significant characters in an identifier with no external linkage is 255

The number of significant characters in an identifier with external linkage is 255

The compiler truncates all external names to 255 characters

Case sensitivity: The case of identifiers is significant in an identifier with external linkage

Characters

The source code page on Windows is the active code page set in `SYSTEM.INI`. The compiler translates to an executable for AS/400 which is EBCDIC, and the default is CCSID 037. The compiler can translate to other EBCDIC code pages with the `/ASCP` compiler option.

When an integer character constant contains a character or escape sequence that is not represented in the basic execution character set, the `char` is assigned the character after the backslash and a warning is issued. For example, `\q` is interpreted as `q`.

When a wide character constant contains a character or escape sequence that is not represented in the extended execution character set, the `wchar_t` is assigned the character after the backslash, and a warning is issued.

When an integer character constant contains more than one character, the last 4 bytes represent the character constant.

When a wide character constant contains more than one multibyte character, the last `wchar_t` value represents the character constant.

The default behavior for `char` is `unsigned`.

Any sequential spaces in your source program are interpreted as one space.

All spaces are retained for the listing file.

The escape sequence values for listed sequences are:

Sequence	Value
<code>\a</code>	0x2F
<code>\b</code>	0x16
<code>\f</code>	0x12
<code>\n</code>	0x25 (Compiler option <code>ASi-</code> results in 0x15)
<code>\r</code>	0x13
<code>\t</code>	0x05
<code>\v</code>	0x11

Multibyte characters on AS/400 are enclosed in `SO` (shift out) hexadecimal value 0x0E, and a `SI` (shift in) hexadecimal value 0x0F characters.

A character is represented by 8 bits, as defined by the `CHAR_BIT` macro in `<limits.h>`.

Mapping is one-to-one between the members of the source character sets (in character and string literals) to members of the execution character set.

Strings

When you use DBCS (with the `/Sn` compiler option), a hexadecimal character that is a valid first byte of a double-byte character is treated as a double-byte character inside a string. A `0` is appended to the character that ends the string. Double-byte characters in strings must appear in pairs.

Integers

Type	Size (bits)	Range (in <limits.h>)
char	8	0 to 255
signed char	8	-128 to 127
unsigned char	8	0 to 255
short	16	-32768 to 32767
signed short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	32	-2147483647 to 2147483647
signed int	32	-2147483648 to 2147483647
unsigned int	32	0 to 4294967295
long	32	-2147483648 to 2147483647
signed long	32	-2147483648 to 2147483647
unsigned long	32	0 to 4294967295
Note: Do not use the values in this table as numbers in a source program. Use the macros defined in <limits.h> to represent these values.		

When you convert an integer to a `signed char`, the least-significant byte of the integer represents the `char`

When you convert an integer to a `short` signed integer, the least-significant 2 bytes of the integer represents the `short int`

When you convert an unsigned integer to a signed integer of equal length, if the value cannot be represented, the magnitude is preserved and the sign is not

When bitwise operations (OR, AND, XOR) are performed on a `signed int`, the representation is treated as a bit pattern

The remainder of integer division is negative if exactly one operand is negative

When either operand of the divide operator is negative, the result is truncated to the integer value and the sign will be negative

The result of a bitwise right shift of a negative signed integral type is sign extended

The result of a bitwise right shift of a non-negative signed integral type or an unsigned integral type is the same as the type of the left operand

Floating-Point Values

Table 35. Floating Point

Type	Size (bits)	Range of Exponents (base 10) (in <float.h>)
float (32-bit)	32	1.17549435e-38 to 3.40282347e+38
double (64-bit)	64	2.2250738585072014e-308 to 1.7976931348623157e+308
long double (64-bit)	64	2.2250738585072014e-308 to 1.7976931348623157e+308

When an integral number is converted to a floating-point number that cannot exactly represent the original value, it is truncated to the nearest representable value

When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest representable value

Arrays and Pointers

The type of the integer required to hold the maximum size of an array (the type of the `sizeof` operator, `size_t`) is `unsigned int`

The type of the integer required to hold the difference between two pointers to elements of the same array (`ptrdiff_t`) is `unsigned int`

When you cast a pointer to an integer, the integer is assigned the offset value (in bytes) of the object from the beginning of the storage block.

When you cast an integer to a pointer, the pointer is set to `NULL`

A 32 bit integer is required for a pointer to be converted to an integral type

Structures, Unions, Enumerations, Bit-Fields

If a member of a union object is accessed using a member of a different type, the result is undefined.

The alignment of the most strictly aligned members are:

Type	Alignment
<code>char</code>	1 byte
<code>short</code>	2 byte
<code>int</code>	4 byte
<code>long</code>	4 byte
<code>float</code>	4 byte
<code>double</code>	8 byte
<code>long double</code>	8 byte
<code>pointer</code>	16 byte

The default type of an integer bit field is `unsigned int`.

Bit fields are allocated from low memory to high memory.

Bit fields can cross storage unit boundaries.

The maximum bit field length is 32 bits. If a series of bit fields does not add up to the size of an `int`, padding may take place.

A bit field cannot have type `long double`.

The expression that defines the value of an enumeration constant cannot have type `long double`.

An enumeration can have the type `unsigned char`, or `signed short`, or `signed int`. In C++, enumerations are a distinct type, and although they may be the same size as a data type such as `char`, they are not considered to be of that type.

Qualifiers

All access to an object that has a type that is qualified as `volatile` is retained.

Declarators

There is no limit for the maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type. The only constraint is your system resources.

Statements

Because the case values must be integers and cannot be duplicated, the maximum number of case values in a `switch` statement is 4294967296.

Preprocessor Directives

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the character constant in the execution character set.

Such a constant can have a negative value.

For the method of searching system include source files (specified within angle brackets) see the *C++ User's Guide*

User include files can be specified in double quotation marks, for example `"myheader.h"`. For the method of searching user include files, see the *C++ User's Guide*

For the mapping between the name specified in the include directive and the external source file name, see the *C++ User's Guide*

For the behavior of each pragma directive, see the *C++ Language Reference*

The `__DATE__` and `__TIME__` macros are always defined as the system date and time.

Library Functions

In extended mode (compiler option `/Se`) and for all C++ programs, the `NULL` macro is defined to be `0`.

When `assert` is executed, if the expression is false, the diagnostic message written by the `assert` macro has the format:

```
Assertion failed: expression, file file_name, line line_number
```

The characters tested by the `isalnum()`, `isalpha()`, `isctrl()`, `islower()`, `isprint()` and `isupper()` functions, are:

Characters Tested	Function
A-Z, a-z, 0-9	<code>isalnum()</code>
A-Z, a-z	<code>isalpha()</code>
anything with <code>0x00</code> to <code>0x40</code> , <code>0x42</code> to <code>0x47</code> , <code>0x50</code> to <code>0x61</code> , or <code>0x63</code> .	<code>isctrl()</code>
a-z	<code>islower()</code>
alphanumeric, punctuation, space [, and].	<code>isprint()</code>
A-Z	<code>isupper()</code>

The value returned by all math functions after a domain error (EDOM) is zero.

The value `errno` is set the value of the macro `ERANGE` on underflow range errors.

If you call the `fmod()` function with `0` as the second argument, `fmod()` returns `0` and a domain error.

Error Handling

See Chapter 14, "Handling Exceptions" on page 255 for a list of the run-time messages generated for `perror()` and `strerror()`. Note that the value of `errno` is not generated with the message.

Use the `/Wn` compile-time option to control the level of messages generated. There is also a `/Wgrp` compiler option that provides programming-style diagnostics to aid you in determining possible programming errors. See the *C++ User's Guide* for a description of the compiler options.

Signals

The set of signals for the `signal()` function and the parameters and usage of each signal are described in Chapter 14, "Handling Exceptions" on page 255 and in the *C Library Reference* under `signal`.

`SIG_DFL` is the default signal, and the default action taken is termination.

If the equivalent of `signal(sig, SIG_DFL);` is not executed at the beginning of signal handler, no signal blocking is performed.

Whenever you leave a signal handler, it is reset to `SIG_DFL`.

At program startup the default handling for SIGIO is set to `SIG_IGN`, and the default handling for SIGABRT is set to `SIG_DFL`.

Translation

Each non-empty sequence of white-space characters other than new-line is replaced by one space character.

A diagnostic message is identified as `msg_id severity text` where `msg_id` consists of 7 characters. The first three characters are upper case letters and the last 4 characters are decimal digits.

The message classes are:

Message Class	Description	Return Status Code
Informational	Advises of conditions found during compilation. Compilation continues.	0
Warning	Warns of a possible error. Compilation continues.	4
Error	Conditions that the compiler can correct. Compilation continues.	8
Severe	Conditions that the compiler cannot correct.	12
Terminal	Internal error.	16

The level of the diagnostic is controlled through the message limit, message severity limit and message flagging options in the `iccas` command.

Streams and Files

The last line of a text stream requires a terminating new-line character, but if the last character is not a new-line character, the compiler inserts one implicitly.

Files are line buffered

Space characters that are written out to a text stream immediately before a new-line character do not appear when the stream is read back in

The number of null characters that can be appended to the end of a binary stream is zero or more for the last record in the file and for all other records, zero

The file position indicator of an append mode stream is positioned at the start of the file when reading and at the end of the file when writing

When a file is opened in write mode, the file is truncated. If the last character of the write is a new-line character, the associated file will be truncated beyond that point

A file of zero length does exist

For the rules for composing a valid file name, refer to “Introducing the AS/400 Data Management File System” on page 49

The same file can be opened multiple times

When the `remove()` function is used on an open file, the `remove` fails

When you use the `rename()` function to rename a file to a name that exists prior to the function call, the `rename()` fails

An open temporary file is automatically deleted if the program terminates abnormally

The output of `%p` conversion in the `fprintf()` function is as follows:

Pointer	Return Value
NULL or invalid pointer	NULL
Space pointer	SP:TTSSc-name:offset:idx:AM
System pointer	SY:TTSSc-name:TTSSo_name:AA:AM
Invocation pointer	IV:idx:AM
Procedure pointer	PR:idx:AM
Open pointer	Dependent on the type of pointer contained in the open pointer.
Suspend pointer	SU:idx:AM
Data pointer	SU:idx:AM
Instruction pointer	IN:idx:AM
Label pointer	LB:idx:AM

Note: TT is the type of the context/object; SS is the subtype; offset is the offset within the space; idx is the index to the pointer; AM the mark of the current activation; AA is the authorization bits;

The input of `%p` conversion in the `fscanf()` function is the same as is generated for `fprintf()` except that `fscanf()` returns NULL for label pointers, data pointers, instruction pointers, suspend pointers and invocation pointers

A '-' character that is neither the first nor the last character in the `fscanf` scan list (`%[characters]`) is considered to be part of the scan list

The possible values of `errno` on failure of `fgetpos` are `EERRSET`, `ENOSEEK`, and `EBADPOS`

The possible values of `errno` on failure of `fgetpos` or `ftell` are `ENODEV`, `ESTDERR`, `ERECIO`, `ENUMMBRS`, `ENUMRECS`

The format of the messages generated by the `perror()` function or `strerror()` function is text format

Memory Management

If the size requested is 0, the `calloc()`, `malloc()`, and `realloc()` functions all return a NULL pointer.

Files are closed and temporary files are erased when the `abort()` function is called.

Environment

You can pass arguments to `main()` through `argv` and `argc`.

int argc Argument count

char *argv[]

Arguments with the program name are in `argv[0]`; `argv[argc]` is always `NULL`.

When the `abort()` function is called, all open files are closed and temporary files are erased.

When a program ends successfully and calls the `exit()` function with the argument other than 0, the process terminates and returns the lower byte value as the return code.

For the format and mode of execution of a string on a call to the `system()` function, see the *C Library Reference* under `system()`.

Localization

A call to `setlocale(LC_ALL, "")` sets the environment to the C default locale.

The supported locales are listed in Chapter 11, "International Locale Support" on page 185.

Time

The local time zone and daylight saving time zone are set by the current locale. There are two entries in the C locale: the time zone difference `TZDIFF`; the time zone name `TNAME`. See Chapter 11, "International Locale Support" on page 185 for more information on specifying the time zone.

The era for the `clock()` function starts when the program is started by either a call from the operating system or a call to `system()`.

C++ Specific Implementation-Defined Behavior

These sections describe how the VisualAge for C++ for AS/400 product defines the behavior classified as implementation-defined in the ANSI C++ Working Paper.

Classes, Structures, Unions, Enumerations, Bit Fields

Class members are allocated in the order declared; access specifiers have no effect on the order of allocation

Padding is added to align class members on their natural boundaries and to end the class on its natural boundary

An `int` bit field behaves as an `unsigned int` for function overloading

Linkage Specifications

The valid values for the string literal in a linkage specification are:

"C++"	Default
"C"	C language linkage

Member Access Control

Class members are allocated in the order declared; access specifiers have no effect on the order of allocation.

Special Member Functions

Temporary objects are generated under the following circumstances:

- During reference initialization
- During evaluation of expressions
- In type conversions
- Argument passing
- Function returns
- In `throw` expressions

Temporary objects exist until there is a break in the flow of control of the program. They are destroyed on exiting the scope in which the temporary object was created with the exception of thrown objects.

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 guide. 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:

LPS: VisualAge for C++ for AS/400, GC09-2414
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 C++ Programming Guide, SC09-2417
ILE C/C++ MI Library Reference, SC09-2418
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++ 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

Other IBM Publications

ADTS/400: Advanced Printer Function, SC41-1766
National Language Support Guide, SC41-4101
ICF Programming, SC41-3442
IDDU Use, SC41-3704

DDS Reference, SC41-4712
Printer Device Programming, SC41-4713
System API Programming, SC41-4800
Backup and Recovery – Basic, SC41-4304
Backup and Recovery – Advanced, SC41-4305
Work Management, SC41-4306
Distributed Data Management, SC41-4307
Optical Support, SC41-4310
International Application Development, SC41-4603
SQL Database Programming, SC41-4701
Data Management, SC41-4710
Integrated File System Introduction, SC41-4711
Printer Device Programming, SC41-4713
Application Display Programming, SC41-4715
Tape and Diskette Device Programming, SC41-4716
CL Reference, SC41-4722
System API Reference, SC41-4801

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 contains a list of books that you might find helpful.

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 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 Margret 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

- _DecimalT class template
 - macros 328
- _DecimalT template class, using 117
- _RIOFB_T structure 324

A

- acquiring a default program device 155
- actions taken when a run-time error occurs 257
- ANSI xvi
 - implementation-defined behavior 357
 - RTTI implementation 347
 - standards supported xvi
- argument passing
 - by reference 203
 - by value directly 203
 - by value indirectly 203
 - match data type requirements 204
 - operational descriptors 204, 242
- arrays, size of 360
- AS/400 back-end compile-time 85
- AS/400 file descriptions
 - compiler generated output 94
 - disconnected mode 115
 - compiler options 116
 - header description 94
 - level checking 96
 - record format layout 92
 - referencing an AS/400 connection 92
 - retrieving 91
 - type definition structure 95
- AS/400 system exceptions
 - C record files 324
 - C stream files 320
- automatic template generation 249

B

- binary coded decimal data
 - conversion functions 117
- binary stream database files
 - I/O considerations 140
- binary stream display files
 - program devices 154

- binary stream files
 - I/O considerations
 - diskette files 174
 - save files 174
 - tape files 174
 - opening (character at a time) 59
 - opening (record at a time) 63
 - opening, reading, writing 61
 - overview 53
 - reading (character at a time) 60
 - reading (record at a time) 64
 - updating 62
 - updating (character at a time) 61
 - writing (character at a time) 60
 - writing (record at a time) 64
- binary stream ICF files
 - I/O considerations 167
 - program devices 167
- binary stream subfiles
 - I/O considerations 160
- binding
 - overview 4

C

- C streams
 - file types supported 50
- C++ language
 - description 3
 - parameter passing styles 198
- calling
 - EPM C 237
 - ILE C++ 236
 - ILE procedures 239
 - ILE programs 229
 - ILE-bindable APIs 239
 - OPM programs 221
- calling programs/procedures 22
 - call stack 194
 - calling procedures 194
 - calling programs 194
 - changing names 215
 - creating C++ classes 216
 - data-type compatibility 204
 - library qualified calls 221
 - linkage specification 196

- calling programs/procedures (*continued*)
 - overview 193
 - passing parameters 197
 - within ILE 22
- cancel handlers 304
 - reason codes 305
- case values, limit of 361
- CEEHDLR 283
- CEEHDLU 283
- CEEMRCR 295
- changing a default program device 157
- characters 357
 - code page 357
 - ctype functions 362
 - default type 358
 - escape sequences 357
 - implementation-defined behavior 357
 - multibyte 357
- checking the AS/400 system exceptions for C record files 324
- checking the AS/400 system exceptions for C stream files 320
- checking the errno value 315
- checking the major/minor return code 320
- checking the return value of a function 315
- classes, exception 291
- clock function, era for 365
- command
 - Add ICF Device (ADDICFDEVE) 168
 - Change ICF File (CHGICFF)
 - ACQPGMDEV parameter 168
 - Create a Display File (CRTDSPF) 153
 - Create DDM file (CRTDDMF) 140
 - Create ICF File (CRTICFF) 168
 - Override Diskette File (OVRDKTF) 171
 - Override ICF Device (OVRICFDEVE) 168
 - Override ICF File (OVRICFF) 168
- common mechanism to return function results 202
- compiler
 - differences between C and C++ 3
 - differences between ILE C and ILE C++ 327
 - differences between Windows C++ and ILE C++ 343
 - standards
 - C++ 3
- compiler options xv
 - /F options 254
 - generation of files for template resolution 254
 - redirecting template-include files 251
 - syntax for xv

- compiling
 - a program 4
 - overview 4
- condition token 295
- constructors 351
 - using RTTI operators 351
- control boundary 261, 309
- conventions, file naming 50
- creating programs 29
 - strategy to avoid 29
- ctype functions, characters in 362

D

- data abstraction 5
- data description specification 132
- data management file system
 - binary stream files, opening (character at a time) 59
 - binary stream files, opening (record at a time) 63
 - binary stream files, opening, reading, writing 61
 - binary stream files, reading (character at a time) 60
 - binary stream files, reading (record at a time) 64
 - binary stream files, updating 62
 - binary stream files, updating (character at a time) 61
 - binary stream files, writing (character at a time) 60
 - binary stream files, writing (record at a time) 64
- binary streams 53
- C streams and file types 50
- file naming conventions 50
- file objects 49
- fopen function 54
- open modes 54
- overview 49
- record files 51
- session I/O 55
- stream buffering 56
- stream files versus database files 53
- text stream files, opening 57
- text stream files, opening, reading, writing 58
- text stream files, reading 58
- text stream files, updating 58
- text stream files, writing 57
- text streams 53
- data-type compatibility 225
 - CL command line call 212
 - ILE CL 207
 - ILE COBOL 206
 - ILE RPG 204
 - length of variables 204

- data-type compatibility (*continued*)
 - OPM CL 211
 - OPM COBOL 209
 - OPM RPG 208
- data-type mapping 113
- database and DDM files
 - record functions 141
- database files 131
 - access path 133
 - arrival sequence access path 133
 - binary stream functions
 - record-at-a-time processing 140
 - commitment control 143, 144
 - comparisons with stream file 40
 - data file 133
 - keyed sequence access path 135
 - arranging key fields 135
 - duplicate key values 137
 - multiple record formats 102
 - null capable fields 139
 - open as binary files 140
 - open as record files 140
 - record-level description 132
 - sharing 138
- database record
 - delete 137
 - lock conditions 138
- database record file
 - arrival sequence 133
 - keyed sequence 135
 - record I/O functions 141
- DDM files 131
- declarators, limit of 361
- declaring pointer variables 179
- decreasing program size 86
- default parameter passing styles 203
- default program device, acquiring 155
- default program device, changing 157
- destructors 351
 - using RTTI operators 351
- device files
 - device attributes feedback area 175
 - multiple record formats 110
 - separate indicator area
 - INDARA keyword 108
 - part of the file buffer 109
- different passing methods 203
- direct monitor handlers 288
- disconnected mode 115
- diskette files 170, 172
 - blocking 174
 - using 172
- display files 149
 - change the default program device 155
 - I/O considerations 154
- display files, ICF files, printer files
 - I/O considerations 149
 - indicators in the file buffer 151
 - major/minor return codes 153
 - open as record files 153
 - option indicators 149
 - response indicators 149
 - separate indicator areas 150
- display files, subfiles, ICF files, printer files
 - binary stream functions 154
 - open as binary stream files 153
 - record functions 154
- document library services file system 36
- dynamic binding 5
- dynamic_cast operator 348
 - casting pointers 348
 - casting references 349
 - using to downcast 348

E

- enabling integrated file system stream I/O 45
- encapsulation 5
- entry module 23
- EPM C, calling 237
- era for clock function 365
- errno macros
 - list of 316
- errno values
 - checking 315
 - for the integrated file system 318
- error macros
 - EBADMODE 54
 - EIOERROR 320
 - EIORECERR 320
 - EPAD 64
 - ERECIO 64
 - ETRUNC 58, 64
 - mapping stream I/O exceptions 323
- escape sequences 357
- example 160, 162, 171, 172, 247
 - acquiring a default program device 155
 - AS/400 pointers 180
 - AS/400 system exception handling 320

example (continued)

- C++ 5
- C++ objects in a C program 217
- calling EPM C 237
- calling ILE-bindable APIs 239
- cancel handlers 307
- changing a default program device 157
- commitment control 144
- condition handler intercepts a thrown exception 272
- control boundary 309
- database file INPUT fields 97
- database file KEY fields 98
- database file NULLFLDS fields 99
- database record file in arrival sequence 133
- database record file in keyed sequence 135
- device file BOTH fields 106
- device file INPUT fields 105
- device file OUTPUT fields 105
- direct monitor handler and signal function 299
- direct monitor handlers using labels 297
- direct monitor intercepts a thrown exception 270
- dynamic_cast operator 348
- exception percolation 313
- externally described files 118
- generating template definitions 247
- I/O feedback area 65
- ILE condition handler 285
- multiple formats in a database file 102
- multiple formats in a device file 110
- ofstream and ifstream classes 45
- opening, reading, writing to a binary stream file 61
- opening, reading, writing to a text stream file 58
- operational descriptors 242
- other ways to handle errors 314
- pointer declarations 179
- printer files 168
- promote an exception 295
- record I/O functions 141
- separate exception-handling program 278
- setting up a signal handler 274
- template-implementation file 251
- template-include file 252
- try-catch-throw 265
- try-catch-throw and direct monitor handlers 266
- try-catch-throw and ILE condition handlers 268
- two ILE condition handlers 286
- typeid operator 350
- updating a binary stream file 62

examples

- calling ILE C++ 236

examples (continued)

- calling ILE programs 229
- calling OPM programs 221
- database file BOTH fields 97
- device file OUTPUT fields and option indicators 109
- dynamic screen manager APIs 56
- indicators as part of the file buffer 151
- indicators in a separate indicator area 150
- performing level check on a database file 100

exception 351

- bad_typeid 351

exception classes 282, 291

exception handling

- actions taken during run-time 257
- AS/400 system exceptions 320
- cancel handlers 304
- checking the AS/400 system exceptions for C record files 324
- checking the errno value 315
- checking the return value of a function 315
- direct monitor handlers 288
- errno macros 316
- handling exceptions in your programs 262
- ILE condition handlers 283
- ILE message handling 256
- nested exceptions 260
- overview 255
- record I/O error macro to exception mapping 323
- signal function 273
- try-catch-throw 264
- unhandled exceptions default actions 261

exception message

- types 257

exceptions, nested 260

exceptions, unhandled 261

externally described files

- database files 96
- device files 104
- using 96, 104

F

- file description 49
- file naming conventions 50
- file objects 49
- file server file system 38
- function
 - _GetExcData 274
 - _Racquire 155
 - _Rdevatr 175

function (*continued*)

- _Rfeod 174
- _Rfeov 175
- _Rformat 160
- _Rindara 150
- _Ropen 154
- _Rpgmdev 155
- _Rreadindv 155
- _Rreadnc 160
- checking the return value 315
- fgetpos 364
- fopen 54, 153
- fopen() 42
 - lrecl parameter 44
- ftell 364
- perror 315
- raise 274
- signal 273, 274
 - signals raised 280
- strerror 315

functions 249

- clock 365
- implementation-defined behavior 362
- templates 249
 - structuring manually 253
 - template-include files 249

G

global variable `_EXCP_MSGID` 320

H

handle system errors 320

handle the signal 281

handling exceptions in your programs 262

header file

- <errno.h> 315
- <except.h> 292
- <leawi.h> 242
- <leod.h> 242
- <recio.h> 139
- <signal.h> 281
- <stdio.h> 54
- <stdlib.h> 80

help xvi

- contextual xvi
- from the command line xvii
- How Do I xvii
- inside VisualAge for C++ for AS/400 xvii

help (*continued*)

online documents xvi

I

ICF files 149, 162

- I/O considerations 162
- using 162

ILE 19

- effects of 24
- family of ILE compilers 19
- overview 19
- program call 22
- program creation 20
- program creation strategies 23
- program management 21
- program scenarios 26

ILE C 19

- as ILE language 19
- in mixed-language programs 27

ILE C++, calling 236

ILE CL 19

- as ILE language 19
- calling ILE C++ program 26
- in mixed-language program 27

ILE COBOL 19

- as ILE language 19

ILE condition handlers 283

- move the resume cursor 295
- percolate an exception 286
- promote an exception 295
- register 283
- resume cursor, move 295

ILE message handling 256

ILE programs, calling 229

ILE-bindable APIs 22

- calling 239
- CEEHDLR 283
- CEEHDLU 283
- CEEMRCR 295
- overview 22
- procedure calls 239

implementation-defined behavior 357

include name 92

indicators as part of the file buffer 151

indicators in a separate indicator area 150

inheritance 5

integrated file system

- binary streams 41
- document library services file system 36

- integrated file system (*continued*)
 - enabling 45
 - errno values 318
 - file server file system 38
 - fopen() function 42
 - LAN Server/400 file system 37
 - library file system 35
 - open modes 44
 - open systems file system 35
 - optical support file system 38
 - other file systems 46
 - overview 33
 - root file system 34
 - session I/O 44
 - storing data as a text or binary stream 42
 - stream buffering 45
 - stream files 39
 - stream files versus database files 40
 - text and binary stream files 42
 - text streams 41
- international locale support 185

L

- LAN Server/400 file system 37
- language description 3
- language standard xvi
 - industry (ANSI, ISO) xvi
- LC_ALL locale variable 186
- LC_COLLATE locale variable 186
- LC_CTYPE locale variable 186
- LC_MONETARY locale variable 186
- LC_NUMERIC locale variable 186
- LC_TIME locale variable 186
- LC_TOD locale variable 186
- library file system 35
- library function 362
 - implementation-defined behavior 362
- locale 186
 - customizing 186
 - environment variables 188
 - LC_TOD locale variable 186
 - locale library function 186
 - setting an active locale 187

M

- macros 362
 - assert 362
 - defined by `_DecimalT` template class 328

- macros (*continued*)
 - NULL 362
 - major/minor return code, checking 320
 - managing programs 21
 - mapping a C++ class to a C structure 216
 - mapping fields from externally described files 113
 - match data type requirements 203, 225
 - migrating
 - C to C++ on AS/400 327
 - Windows C++ to C++ on AS/400 343
 - multiple record formats
 - database file 102
 - device file 110

N

- nested exceptions 260
- null capable fields 139
- NULL macro, definition 362

O

- object-oriented programming 5
- open data path 138
 - I/O feedback area 65
 - open feedback area 65
- open modes for integrated file system files 44
- open pointer rules 178
- open systems file system 35
- opening binary stream files (character at a time) 59
- opening binary stream files (record at a time) 63
- opening text stream files 57
- operational descriptors 204
- operator 348
 - `dynamic_cast` operator 348
 - `typeid` operator 350
- operator overloading 3
- OPM programs, calling 221
- optical support file system 38
- optimizing 86
 - for size 86
 - for speed 86
 - for speed and size 86
- options, compiler xv
 - See also* compiler options
- other pointer rules 178
- other than open pointers rules 178
- overloaded function 16

P

- parameters
 - default passing styles 203
 - passing in C++ 198
 - passing in ILE 197
- passing arguments 203
- passing constants 212
- passing styles 203
- passing variables 213
- performance tips 85
 - bit fields 70
 - block records 75
 - compiler performance option
 - /Asp 71
 - data types 81
 - direct monitor handler 72
 - dynamic heap 80
 - dynamic memory allocation calls 81
 - exception handling 71
 - exception messages
 - C2M 72
 - feedback information
 - _RIOFB_T 74
 - function calls operators 72
 - ILE bindable API
 - CEECSST 81
 - CEEFRST 81
 - indirect access 79
 - library names 76
 - open pointers 77
 - operators 85
 - overview 69
 - percolating exceptions 72
 - physical files 76
 - pointer comparisons 77
 - record I/O 74
 - register storage class 70
 - run-time limits 86
 - shared files 76
 - space considerations 80
 - space padding 82
 - static and global variables 70
 - stream functions 76
 - system buffer 75
 - tape files 76
 - volatile qualifier 70
 - Windows front-end compile-time 85
- pointer 348
 - dynamic_cast operator 348
- pointers 178
 - casting rules 183
 - declaring pointer variables 179
 - open rules 178
 - overview 177
 - pass AS/400 pointers as arguments 180
 - size of 360
 - types 177
- portability xvi
 - considerations xvi, 357
- porting ILE C to ILE C++
 - binary coded decimal class library 328
 - character array initialization 341
 - file inclusion 340
 - function prototypes, declarations and pointers 340
 - header files 337
 - integrated file system 342
 - inter-language calls 327
 - name mangling 340
 - overview 327
 - set_terminate function 342
 - string literals 342
 - type checking 340
- porting Windows C to ILE C++
 - abort function 344
 - exception handler 345
 - externally described files 345
 - inter-language calls 344
 - operational descriptors 345
 - pointers 345
 - record I/O 344
 - user interface and GUI applications 344
 - Windows terms 344
- porting Windows C++ to ILE C++
 - overview 343
- pragma
 - define 253
 - disable_handler 288
 - exception_handler 288, 299
 - implementation 251
 - undeclared 252
- preprocessor directives 16
- printer files
 - FCFC 149
 - I/O considerations 168
 - using 168
- procedure 194
 - procedure pointer call 194
 - static procedure call 194

- program 194
 - calling 194
 - compiling 4
 - development process 4
 - OPM-compatible 29
 - preparing 4
- program creation 29
 - strategy to avoid 29
- program creation in ILE 20
- program devices
 - I/O feedback area 65
- program entry procedure (PEP) 194
 - and the call stack 194
- program management 21
- program-described files 132
- program/procedure call 22
 - call stack 194
 - calling procedures 194
 - calling programs 194
 - changing names 215
 - creating C++ classes 216
 - data-type compatibility 204
 - library qualified calls 221
 - linkage specification 196
 - overview 193
 - passing parameters 197
 - within ILE 22
- programming languages
 - C++ features 3
 - supported 19
- record files (*continued*)
 - I/O considerations (*continued*)
 - save 174
 - tape 174
- record format name 93
- record I/O error macro to exception mapping 323
- record ICF files
 - I/O considerations 167
 - program devices 167
- record subfiles
 - I/O considerations 160
- record tape files
 - blocking 175
 - I/O considerations
 - using _Rfeod 174
 - using _Rfeov 175
- redefining basic language operators 3
- reducing program size 86
- rename function 364
- retrieve a return value from main 236
- retrieving AS/400 file descriptions 91
- return function results 202, 236
- return value of a function, checking 315
- root file system 34
- run time type information 347
 - C++ implementation 347
 - constructors and destructors 351
 - dynamic_cast operator 348
 - type_info class 351
 - typeid operator 350
- run-time limits 86

Q

- QINLINE 56

R

- reading binary stream files (character at a time) 60
- reading binary stream files (record at a time) 64
- reading syntax diagrams xiii
- reading text stream files 58
- reason codes for cancel handlers 305
- record diskette files
 - blocking 175
 - read and write 175
- record display files
 - I/O considerations 154
- record field names 94
- record files 51
 - I/O considerations
 - diskette 174

S

- save files 170
- session I/O, integrated file system 44
- signal
 - SIG_DFL 280
 - SIG_IGN 280
 - SIGABRT 280
 - SIGFPE 280
 - SIGILL 280
 - SIGINT 280
 - SIGIO 280
 - SIGOTHER 280
 - SIGSEGV 280
 - SIGTERM 280
 - SIGUSR1 280
 - SIGUSR2 280

- signal handler
 - change the state of nested 263
- signal handling 281
- signal handling-action definitions 281
- source debug 22
 - overview 22
- standards, language xvi
- storing data as a text or binary stream 42
- strategies for creating ILE programs 23
- stream buffering
 - fully buffered 45, 56
 - line buffered 45, 56
 - unbuffered 45, 56
- stream files
 - comparison to record files 40, 53
 - open modes 54
 - processing 52
- subfiles 160
 - I/O considerations 160
 - using 160
- supported programming languages 19
- syntax xiii
 - for commands, preprocessor directives, statements xiii
 - for compiler options xv
 - how to read xiii
- syntax diagrams xiii

T

- tape files 170, 171
 - blocking 174
 - using 171
- tape files, diskette files, save files
 - binary stream files
 - I/O considerations 174
 - binary stream functions 175
 - I/O considerations 171
 - open as binary stream files 170
 - open as record files 170
 - record functions 175
- template-implementation files 249
 - creating 249
 - naming 251
- template-include files 251, 252
- templates 245
 - See also* Language Reference
 - #pragma define 253
 - automatic instantiation 249
 - example 247

- templates (*continued*)
 - generating function definitions 246
 - how the compiler expands 246
 - implementation files 249
 - including everywhere 248
 - manual instantiation 253
 - TEMPINC directory 251
 - template-include files 249
 - terms 245
- text stream files
 - opening 57
 - opening, reading, writing 58
 - reading 58
 - updating 58
 - writing 57
- text streams 53
 - stderr 44, 55
 - stdin 44, 55
 - stdout 44, 55
- time 186
 - zone, specifying 186
- translation 363
- try-catch-throw 264
- type information 347
 - extended class 352
 - for RTTI 347
 - usage 351

U

- updating binary stream files (character at a time) 61
- updating text stream files 58
- user entry procedure (UEP) 194
 - and the call stack 194

W

- writing binary stream files (character at a time) 60
- writing binary stream files (record at a time) 64
- writing text stream files 57

Z

- zoned decimal data
 - conversion functions 117
- zoned decimal data, using 117