AS/400e™

# ILE C for AS/400® Language Reference

*Version 4*

SC09-2711-01

AS/400e™

# ILE C for AS/400® Language Reference

*Version 4*

SC09-2711-01

**Note**

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

**Second Edition (June 1999)**

This edition applies to Version 4, Release 4, Modification Level 0, of IBM Application System/400 ILE C (Program 5769-CX2) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the proper edition for the level of the product.

This edition replaces SC09–2711–00.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

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

IBM welcomes your comments. You can send your comments to:

IBM Canada Ltd. Laboratory,
2G/KB7/1150/TOR
1150 Eglinton Avenue East
Toronto, 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 “How to Send Your Comments” on page xiv for a description of the methods.

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.

# Contents

# Tables

# Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM product, program, or service 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. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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:

| Director of Licensing,
| Intellectual Property & Licensing
| International Business Machines Corporation,
| North Castle Drive, MD - NC119
| Armonk, New York 10504-1785,
| U.S.A.

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Ltd.
Department 071
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

## Programming Interface Information

This book is intended to help you write Integrated Language Environment C for AS/400 programs. It contains information necessary to use the Integrated Language Environment C for AS/400 compiler. The *ILE C for AS/400 Language Reference* and the *ILE C for AS/400 Run-Time Library Reference* primarily document general-use programming interfaces and associated guidance information provided by the Integrated Language Environment C for AS/400 compiler.

## Trademarks and Service Marks

## Industry Standards

The Integrated Language Environment C compiler and library are designed according to the American National Standard Institute (ANSI) for Programming Languages — C, ANSI/ISO 9899-1990 standard.

## About This Book

This book contains reference information on:

- Elements of C
- Declarations, definitions, functions, expressions, and operators
- C language statements
- Preprocessor directives
- Input/output considerations

Use this book as a reference when you write Integrated Language Environment® (ILE) C applications.

This guide does not describe how to program in the ILE C programming language nor does it explain the concepts of ILE. The following are companion publications to this book:

- *ILE C for AS/400 Programmer's Guide*
- *ILE C for AS/400 Run-Time Library Reference*
- *ILE Concepts*

For information about other ILE C publications, see either of the following:

- The *Publications Reference*.
- The *AS/400e series Softcopy Library CD-ROM*.

For a list of related publications, see "Bibliography" on page 183.

Use the AS/400 Information Center as your starting point for looking up AS/400technical information. You can access the Information Center from the AS/400e Information Center CD-ROM (English version: *SK3T-2027-01*) or from one of these Web sites:

```
http://www.as400.ibm.com/infocenter
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

The AS/400 Information Center contains important topics such as logical partitioning, clustering, Java, TCP/IP, Web serving, and secured networks. It also contains Internet links to Web sites such as the AS/400 Online Library and the AS/400 Technical Studio. Included in the Information Center is a link that describes at a high level the differences in information between the Information Center and the Online Library.

For a softcopy version of AS/400 publications refer to the *AS/400e series Softcopy Library CD-ROM*, SK3T-0118-03.

## Who Should Use This Book

This book is intended for programmers who are familiar with the C programming language and who want to write or maintain ILE C applications. This book is a reference rather than a tutorial. You must have experience in using applicable AS/400® menus and displays, or control language (CL) commands, and knowledge of ILE as explained in *ILE Concepts*.

## A Note About Examples

The examples in this book that illustrate the use of the ILE C compiler are written in a simple style. They are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all possible uses of C language constructs. Some examples are only code fragments and do not compile without additional code.

Other examples, which are shipped with the product, appear in the *ILE C for AS/400 Run-Time Library Reference* and the *ILE C/C++ MI Library Reference*.

## How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other AS/400 documentation.

- If you prefer to send comments by mail, use the following address:

  IBM Canada Ltd. Laboratory
  Information Development
  2G/KB7/1150/TOR
  1150 Eglinton Avenue East
  Toronto, Ontario, Canada M3C 1H7

  If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use the following number:
  - 1-416-448-6161
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

    torrcf@ca.ibm.com

    IBMLink: to toribm(torrcf)
  - Comments on the AS/400 Information Center:

    RCHINFOC@us.ibm.com

Be sure to include the following:
- The name of the book
- The publication number of the book
- The page number or topic to which your comment applies.

# Chapter 1. Introduction to C

This section introduces you to the C programming language and shows you how to structure C language source programs.

**Implementation-defined behavior** is any action that is defined by the compiler and library and not by standards.

**Undefined behavior** is any action by the compiler and library on an erroneous program that does not result in any expected manner. You should not write programs that rely on such behavior.

**Unspecified behavior** is any other action by the compiler and library that is not defined.

## Overview of the C Language

The C programming language is designed for a wide variety of programming tasks. It is used for system-level code, text processing, graphics, and in many other application areas. The language contains a concise set of statements and numerous data types such as characters, packed decimal, integers, floating-point numbers, and pointers — each in a variety of forms. In addition, C also supports arrays, structures (records), unions, and enumerations. Additional functionality is provided by the C library. This feature enables C to be flexible and efficient, as well as highly consistent across different systems.

The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as well as date and time manipulation. Use of this library helps to maintain program portability, because the underlying implementation details for the various operations need not concern the programmer. See the *ILE C for AS/400 Run-Time Library Reference* for more details.

## C Source Programs

A **C source program** is a collection of one or more directives, declarations, and statements that are contained in one or more source files.

| | |
|---|---|
| **Statements** | Specify the action to be performed. |
| **Directives** | Instruct the C preprocessor to act on the text of the program. |
| **Declarations** | Establish names and define characteristics such as scope, data type, and linkage. |
| **Definitions** | Declarations that allocate storage for data objects or define a body for functions. An object definition allocates storage and may optionally initialize the object. |

A function definition includes a function body. The **function body** is a compound statement that may contain declarations and statements that define what the function does. The function definition declares the function name, its parameters, and the data type of the value it returns.

The order and scope of declarations affect how you can use variables and functions in statements. In particular, an identifier can be used only after it is declared.

A program must contain at least one function definition. If the program contains only one function definition, the function must be called `main()`. If the program contains more than one function definition, only one of the functions can be called `main()`. The `main()` function is the first function that is called when a program is run.

This is the source code of a simple C program:

```
#include <stdio.h>              /* standard library header which
                                   contains I/O function declarations
                                   such as printf used below          */
#include <math.h>               /* standard library header which
                                   contains math function declarations
                                   such as cos used below             */

#define NUM 46.0                /* Preprocessor directive             */

double x = 45.0;                /* External variable
                                           definitions                */

double y = NUM;


int main(void)                  /* Function definition
                                   for main function                  */
{
   double z;                    /* Automatic variable                 */
   double w;                    /*   definitions                      */

   z = cos(x);                  /* cos is declared in math.h as
                                         double cos(double arg)       */
   w = cos(y);
   printf ("cosine of x is %lf\n", z);  /* Print cosine of x          */
   printf ("cosine of y is %lf\n", w);  /* Print cosine of y          */
}
```

This source program defines `main()` and calls the function `cos()`, declared in the <math.h> library header file. It also calls the `printf()` function declared in the <stdio.h> library header file. The program defines the global variables `x` and `y`, initializes them, and declares two local variables `z` and `w`.

## C Source Files

The text of a C source program is kept in one or more source files. To create an executable program object, you can compile the source files individually and then bind them as one program. To include other source files in a single compilation, use the #include directive in the source file. The collection of source files that is compiled during a single compilation is called a **compilation unit**.

A source file contains any combination of preprocessor directives, declarations, and definitions. You can split items such as function definitions and large data structures between source files, but you cannot split them between compilation units. Before the source file is compiled, preprocessor directives are run and macro calls expanded.

It can be useful to place variable definitions in one source file and declare references to those variables in any source files that use them. Doing so makes definitions easy to find and change, if necessary. You can also organize constants and macros into separate files and include them into source files as required.

Directives in a source file apply to that source file and its included files only. Each directive applies only to the part of the file following the directive.

The following example is a C program in two source files. The `main()` and `max()` functions are in separate files. The processing of the program begins with the `main()` function.

**Source file 1**

```
/************************************************************************
*  Source file 1 - main function                                        *
************************************************************************/
#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int, int);             /* Function declaration */

int main(int argc, char * argv[])      /* Function definition  */
{
   int u, w, x, y, z;

   u = 5;
   z = 2;
   w = max(u, ONE);
   x = max(w,TWO);
   y = max(x,THREE);
   z = max(y,z);
}
```

**Source file 2**

```
/************************************************************************
*  Source file 2 - max function                                         *
************************************************************************/
int max (int a,int b)                /* Function  definition          */
{
   if ( a > b )
       return (a);
   else
       return (b);
}
```

The first source file declares the function `max()`, but does not define it. This is a reference to the function in source file 2. Four statements in `main()` are **function calls** to `max()`.

The lines beginning with a number sign (#) are preprocessor directives. These directives direct the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the digits `1`, `2`, and `3`. The directives do not apply to the second source file.

The second source file contains the function definition for the `max()` function, which is called four times in the `main()` function. After you compile the source files, you can bind and run them as a single program.

## Program Processing

Every program must have a User Entry Procedure (UEP). In ILE C, the UEP is the function that is named `main()`. See the *ILE C for AS/400 Programmer's Guide* for more information about the UEP.

The `main()` function is the starting point for running a program. The statements within the `main()` function are run sequentially. There may be calls to other functions. A program usually stops running at the end of the `main()` function, although it can stop at other points in the program.

You can make your program more modular by creating separate functions to perform a specific task or set of tasks. The `main()` function calls these functions to perform the tasks. Whenever a function call is made, the statements are run sequentially starting with the first statement in the function. Control will return back to the calling function when a `return` statement or the end of the function is encountered.

You can declare any function to have parameters. When functions are called, they receive values for their parameters from the arguments that are passed by the calling functions. You can declare parameters for the `main()` function so you can pass values to `main()` from the command line. The command line that starts the program can pass such values as described in "main" on page 69.

## Scope

An identifier becomes **visible** with its declaration.

The region where an identifier is visible is referred to as the identifier's **scope**. The four kinds of scope are:

- Block
- Function
- File
- Function prototype

The scope of an identifier is determined by where the identifier is declared. See "Identifiers" on page 12 for more information on identifiers.

| | |
|---|---|
| **Block scope** | The identifier's declaration is located inside a block. A block starts with an opening brace ({) and ends with a closing brace (}). An identifier with block scope is visible from the point where it is declared to the closing brace that ends the block.<br><br>You can nest block visibility. A block that is nested inside a block can contain declarations that redeclare variables that are declared in the outer block. The new declaration of the variable applies to the inner block. The original declaration is restored when program control returns to the outer block. A variable from the outer block is visible inside inner blocks that do not redefine the variable. |
| **Function scope** | The only identifier with function scope is a label name. A label is implicitly declared by its appearance in the program text. A `goto` statement transfers control to the label that is specified on the `goto` statement. The label is visible to any `goto` statement that appears in the same function as the label. |
| **File scope** | The identifier's declaration appears outside of any block. It is visible from the point where it is declared |

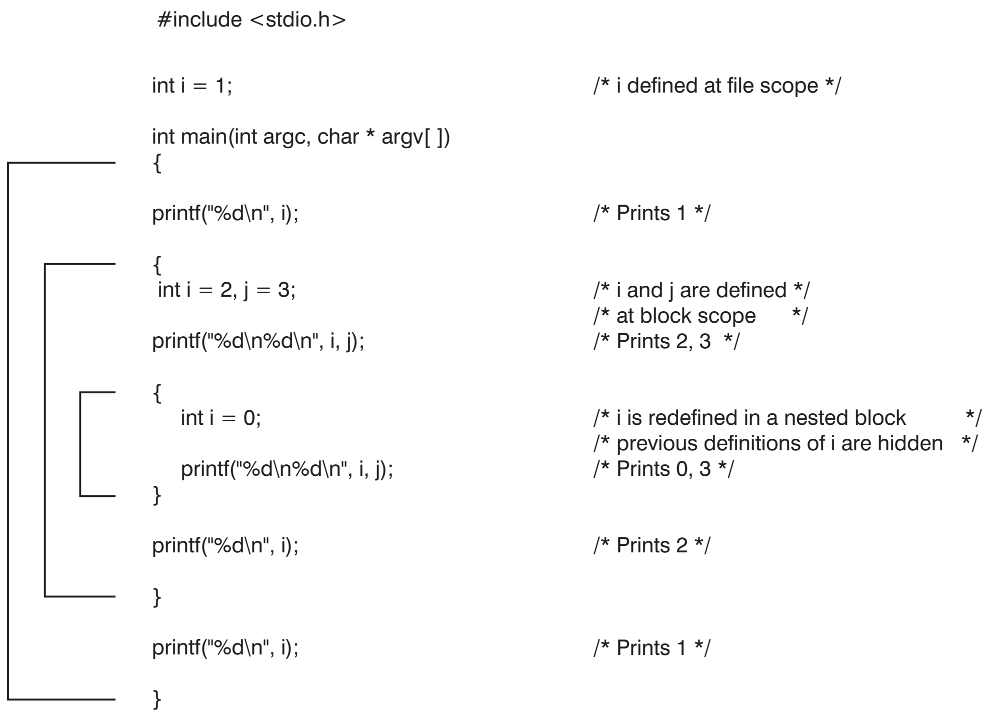
to the end of the source file. If source files are included by `#include` preprocessor directives, those files are considered to be part of the source. The identifier will be visible to all included files that appear after the declaration of the identifier. The identifier can be declared again as a block scope variable. The new declaration replaces the file-scope declaration until the end of the block.

**Function prototype scope**
The identifier's declaration appears within the list of parameters in a function prototype. It is visible from the point where it is declared to the closing parenthesis of the prototype declaration.

In the following example, the variable `x`, which is declared on `line 1`, is different from the `x` declared on `line 2`. The variable that is declared on `line 2` has function prototype scope and is visible only up to the closing parenthesis of the prototype declaration. Visibility of the variable `x` declared on `line 1` resumes after the end of the prototype declaration.

```
1   int x = 4;             /* variable x defined with file scope */
2   long myfunc(int x, long y); /* variable x has function       */
3                               /* prototype scope               */
4   int main(void)
5   {
6      /* . . . */
7   }
```

The following program illustrates blocks, nesting, and scope. The example shows two kinds of scope: file and block. The `main()` function prints the values `1, 2, 3, 0, 3, 2, 1` on separate lines. Each instance of `i` represents a different variable.



```
#include <stdio.h>

int i = 1;                          /* i defined at file scope */

int main(int argc, char * argv[ ])
{

printf("%d\n", i);                  /* Prints 1 */

{
 int i = 2, j = 3;                  /* i and j are defined */
                                    /* at block scope      */
printf("%d\n%d\n", i, j);           /* Prints 2, 3  */

{
    int i = 0;                      /* i is redefined in a nested block         */
                                    /* previous definitions of i are hidden   */
    printf("%d\n%d\n", i, j);       /* Prints 0, 3 */
}

printf("%d\n", i);                  /* Prints 2 */

}

printf("%d\n", i);                  /* Prints 1 */

}
```

*Figure 1. Example Illustrating Blocks, Nesting, and Scope*

## Linkage

The association or lack of association between two identical identifiers is known as **linkage**. A C identifier can have one of the following kinds of linkage:

| | |
|---|---|
| **Internal linkage** | Identical identifiers within a single source file refer to the same data object or function |
| **External linkage** | Identical identifiers in separately compiled files refer to the same data object or function |
| **No linkage** | Each identifier refers to a unique object. |

In Figure 2, the variable `b` is declared in both Source File 1 and Source File 2 as `extern` and refers to the same data object. It has external linkage.

If the declaration of an identifier with file scope contains the keyword `static`, it has internal linkage. In Figure 2, all references to the variable `a` in Source File 1 refer to the same data object. The variable `a` in Source File 2 refers to a different data object than `a` in Source File 1.

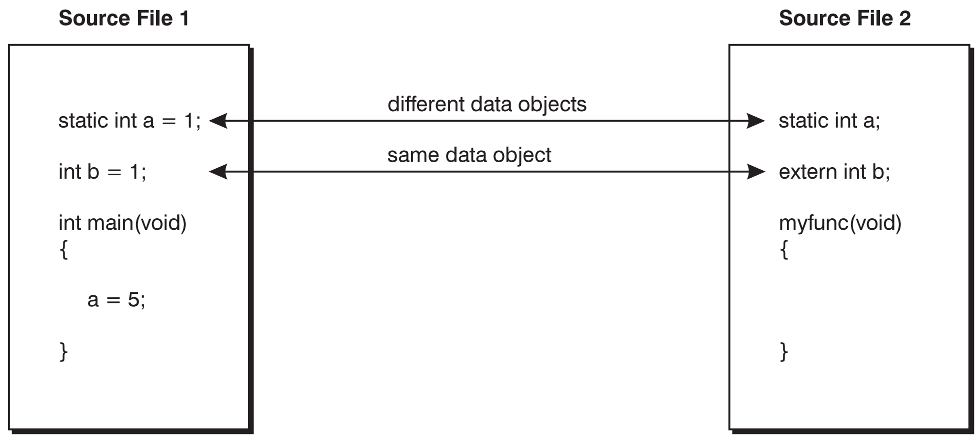


*Figure 2. Example of External and Internal Linkage*

If the declaration of an identifier has the keyword `extern` and if there is a previous declaration of the identifier at file scope, the identifier has the same linkage as the first declaration. If a declaration of the identifier is not visible within the file scope, the identifier has external linkage.

In Figure 3 on page 7, the variable `x` has internal linkage because the first declaration of `x` occurs in file `try.h` and the storage class `static` is specified. The variable `y` in Figure 3 on page 7 has external linkage because a previous declaration of the identifier `y` is not visible within the same file scope.

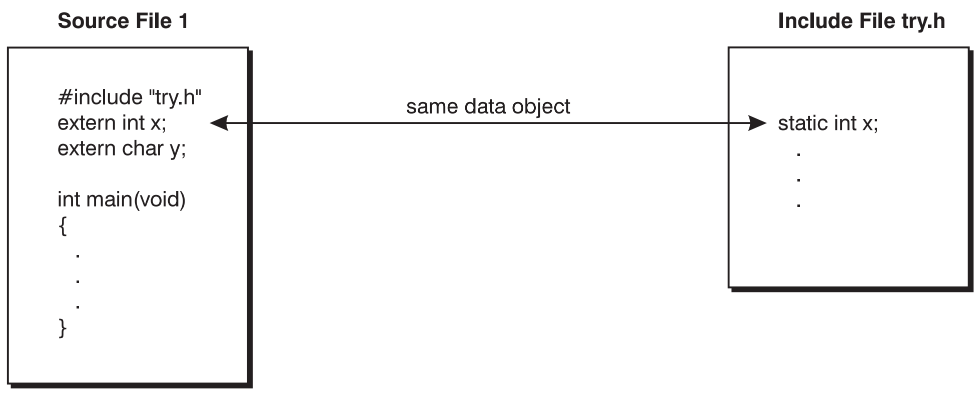


*Figure 3. Example of Linkage Using the Keyword extern*

If an object or function is declared without a storage class specifier at file scope, it has external linkage.

An identifier that falls into one of the following categories has no linkage:

- An identifier that does not represent an object or a function. For example, a C label is neither an object nor a function.
- An identifier that represents a function parameter.
- An identifier declared inside a block without the keyword `extern`.

You can make identifiers refer to the same object or function in other source files with appropriate `extern` declarations, as described in "Chapter 3. Declarations and Definitions" on page 23.

## Storage Duration

**Storage duration** determines how long storage for an object exists. An object has either **static** storage duration or **automatic** storage class that depends on its declaration.

An object with static storage duration has storage allocated for it prior to program startup which remains available until the end of the program. All objects with file scope have static storage duration. An object has static storage duration if it has internal or external linkage, or if it contains the keyword `static`. All other objects have automatic storage.

Storage for an object with automatic storage class is allocated and removed according to the scope of the identifier. For example, storage for an object declared at block scope is allocated when the block is entered and removed when the closing brace of the block is reached. An object has automatic storage duration if it is declared with no linkage and does not have the `static` storage class specifier.

## Name Spaces

In any C program, identifiers refer to functions, data objects, labels, tags, parameters, macros, and typedefs. C allows the same identifier to be used for more than one class of identifier, as long as you follow the rules that are outlined in this section.

**Name spaces** are categories that are used to group similar types of identifiers.

You must assign unique names within each name space to avoid conflict. The same identifier can be used to declare different objects as long as each identifier is unique within its name space. The context of an identifier within a program lets the compiler resolve its name space without ambiguity.

Identifiers in the same name space can be redefined within enclosed blocks as described in "Scope" on page 4.

Within each of the following four name spaces, the identifiers must be unique.

- These identifiers must be unique within a single scope:
  - Function names
  - Variable names
  - Names of function parameters
  - Enumeration constants
  - `typedef` names
- Tags of these types must be unique within a single scope:
  - Enumerations
  - Structures
  - Unions
- Members of structures and unions must be unique within a single structure or union.
- Statement labels have function scope and must be unique within a function.

Structure tags, structure members, and variable names are in three different name spaces. No conflict occurs among the three items named `student` in the following example:

```
struct student       /*  structure tag       */
{
   char student[20]; /*  structure member    */
   int class;
   int id;
} student;           /*  structure variable  */
```

Each occurrence of `student` is interpreted by its context in the program. For example, when `student` appears after the keyword `struct`, it is a structure tag. When `student` appears after either of the member selection operators **.** or ->, the name refers to the structure member. (See "Chapter 5. Expressions and Operators" on page 79 to find out how to refer to members of union or structure variables.) In other contexts, the identifier `student` refers to the structure variable.

# Chapter 2. Lexical Elements of C

This chapter describes the basic elements of the C programming language.

## Character Set

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

- The uppercase and lowercase letters of the English alphabet

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

- The decimal digits `0` through `9`

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

- The following graphic characters:

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

- The caret (ˆ) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (¬) character in EBCDIC
- The split vertical bar (¦) character in ASCII, which may be represented by the vertical bar (|) character on EBCDIC systems
- The space character
- The control characters that represent horizontal tab, vertical tab, form feed, and end of string.

Uppercase and lowercase letters are treated as distinct characters. If a lowercase `a` is specified as part of an identifier name, you cannot substitute an uppercase `A` in its place.

For the keyboards that do not support the entire character set, you can use trigraphs as alternative symbols to represent some characters.

## Trigraphs

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

*Table 1. Trigraph Sequences*

| Trigraph | Character |
|---|---|
| ??= | # |
| ??( | [ |
| ??) | ] |
| ??< | { |
| ??> | } |
| ??/ | \ |
| ??' | ˆ<br>**Note:** AS/400 systems translate and store this trigraph as a NOT symbol. |
| ??! | \| |

Table 1. Trigraph Sequences (continued)

| Trigraph | Character |
|---|---|
| ??- | |

## Escape Sequences

An escape sequence contains a backslash (\) symbol followed by one of the escape sequence characters: a, b, f, n, r, t, v, ', ", ?, or \ or followed by an octal or hexadecimal number. A hexadecimal escape sequence contains an x followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence contains one or more octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

You can represent any member of the character set that is used at run time by an **escape sequence**. For example, you can use escape sequences to place such characters as tab, carriage return, and backspace into an output stream. An escape sequence has the form:

```
►►──\──┬─escape_sequence_character──┬──────────────────────────────────►◄
       ├─octal_constant─────────────┤
       └─x──hexadecimal_constant────┘
```

The C language escape sequences and the characters they represent are:

Table 2. Escape Sequences

| Escape Sequence | Character Represented |
|---|---|
| \a | Alert (bell) |
| \b | Backspace |
| \f | Form feed (new page) |
| \n | New-line (IFSIO changes the value to 0x25) |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \? | Question mark |
| \\ | Backslash |
| **Note:** \ The line continuation sequence (\ followed by a new-line character) which is used in C language character strings to indicate that the current line continues on the next line, is not an escape sequence. See page 137 for more information on the line continuation character. | |

The value of an escape sequence represents the member of the character set that is used at run time. For example, on a system that uses the ASCII character codes, the escape sequence \x56 represents the letter V. On a system that uses EBCDIC character codes, the letter V is represented by \xE5.

**Notes on Usage**

You can place an escape sequence in a character constant or in a string constant. An error message is issued if an escape sequence is not recognized.

When you want to represent the backslash in string and character sequences, use the `\\` escape sequence.

**Related Information**

- "Character Constants" on page 18
- "Strings" on page 20

## Comments

You can use **comments** to document code. Comments are notes in the source code that are replaced by one space character when the code is compiled and are otherwise ignored.

Comments begin with the `/*` characters, end with the `*/` characters, and may span more than one line. You can place comments anywhere the C language allows white space. White space includes space, tab, form feed, and new-line characters.

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

In the following program, `line 6` is a comment:

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

Because the comment on line 6 is equivalent to a space, the output is as follows:

```
This program has a comment.
```

`/*...*/` found in a string literal is not interpreted as a comment. For example, line 6 in the following program is not a comment.

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

The output is as follows:

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

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

In the following example, the comments are shaded:

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

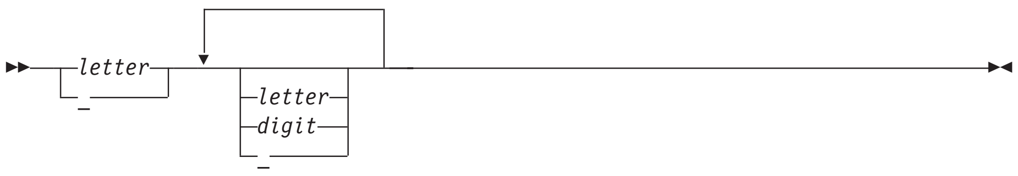
In `test_function()`, the compiler reads the `/*` in line 14 through the `*/` in line 17 as a comment and line 18 as C language code, causing errors at line 18. You can use conditional compilation preprocessor directives to cause the compiler to bypass sections of you program to avoid commenting over comments already in the source code. For example, instead of commenting out the above statements, change line 2 and lines 14 to 18 in the following way:

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

Multibyte characters can also be included within a comment.

## Identifiers

**Identifiers** provide names for functions, data objects, labels, tags, parameters, macros, and typedefs. An identifier has the form:



There is no limit for the number of characters in an identifier. However, only the first several characters of an identifier may be significant. The following table shows the minimal character lengths of identifiers that are recognized. Other compilers may have implemented a minimum number of significant characters that is less than or greater than 255.

| Identifier | Minimum Number of Significant Characters |
|---|---|
| Static data objects | 255 |
| Static function names | 255 |
| External data objects | 255 |
| External function names | 255 |

For identifiers, uppercase and lowercase letters are viewed as different symbols. Thus, `PROFIT` and `profit` represent different identifiers.

**Note:** For complete portability, never use different case representations to refer to the same object.

Avoid creating identifiers that begin with an underscore (_) for function names and variable names. Identifiers that begin with an underscore that is followed by an uppercase letter, or beginning with two underscores are always reserved. Other than member names, identifiers with file scope that begin with an underscore are reserved.

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

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

## Keywords

The C language reserves some words for special usage, known as **keywords**. You cannot use these words as identifiers. Although you can use them for macro names, it is not recommended that you do so. Only the exact spellings of the words as specified below are reserved. For example, `auto` is reserved, but `AUTO` is not.

The following table lists the C language keywords:

*Table 3. ILE C Language Keywords*

| | | | |
|---|---|---|---|
| `_Packed` | `digitsof`[1] | `if` | `static` |
| `auto` | `do` | `int` | `struct` |
| `break` | `double` | `long` | `switch` |
| `case` | `else` | `precisionof`[1] | `typedef` |
| `char` | `enum` | `register` | `union` |
| `const` | `extern` | `return` | `unsigned` |
| `continue` | `float` | `short` | `void` |
| `default` | `for` | `signed` | `volatile` |
| `decimal`[1] | `goto` | `sizeof` | `while` |
| **Note:** [1] The ILE C compiler recognizes _Decimal, __digitsof and __precisionof as keywords. However, decimal, digitsof and precisionof are defined as macros in the <decimal.h> header file for compatibility with other C compilers. | | | |

## Constants

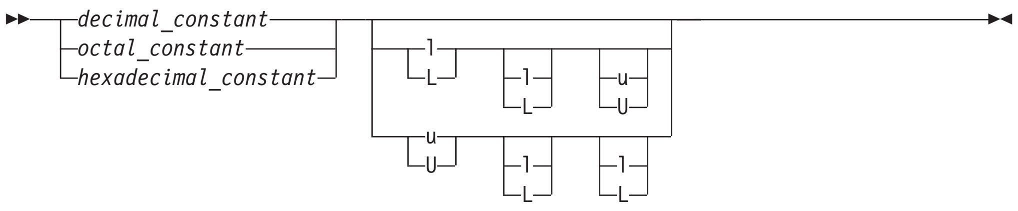
The ILE C language contains the following types of constants:
- Integer
- Floating-Point
- Character
- Decimal
- Packed Decimal
- String
- Enumeration.

A constant is data with a value that does not change during the processing of a program. The value of any constant must be in the range of representable values for its type.

For more information on data types, see “Types” on page 38.

## Integer Constants

**Integer constants** can be either decimal, octal, or hexadecimal values. The following diagram lists these forms:



### Data Types for Integer Constants

The data type of an integer constant is determined by the constant’s value. The following table describes the integer constant and a list of possible data types for that constant. The smallest data type in the list that can contain the constant value will be associated with the constant.

*Table 4. Data Types for Integer Constants*

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

Table 4. Data Types for Integer Constants (continued)

| Constant | Data Type |
|---|---|
| suffixed by all (u or U, l or L, and l or L) or (l or L, lor L, and u or U) | unsigned long long int |

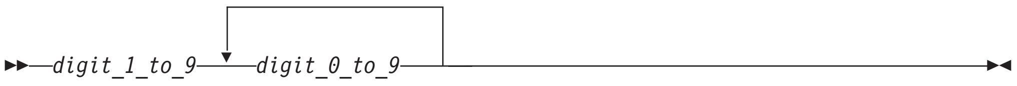
A plus (+) or minus (-) symbol can precede the constant. It is treated as a unary operator rather than as part of the constant value.

**Related Information**

- "Decimal Constants"
- "Octal Constants" on page 16
- "Hexadecimal Constants"
- "Integers" on page 40
- See the *ILE C for AS/400 Run-Time Library Reference* for more information on the <limits.h> header file.

## Decimal Constants

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



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

**Data Type**

See Table 4 on page 14 for a complete description of the data types of decimal constants.

**Examples**

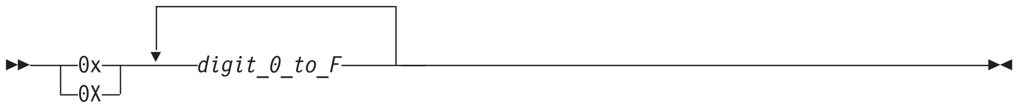
```
485976
433132211
20
5
```

**Related Information**

- "Integer Constants" on page 14
- "Octal Constants" on page 16
- "Hexadecimal Constants"
- "Integers" on page 40

## Hexadecimal Constants

A **hexadecimal constant** begins with the 0 digit that is followed by either an x or X. After the 0x, you can place any combination of the digits 0 through 9 and the letters a through f or A through F. When used to represent a hexadecimal constant, the lowercase letters are equivalent to their corresponding uppercase letters. A hexadecimal constant has the form:



**Data Type**

See Table 4 on page 14 for a complete description of the data types of hexadecimal constants.

**Examples**

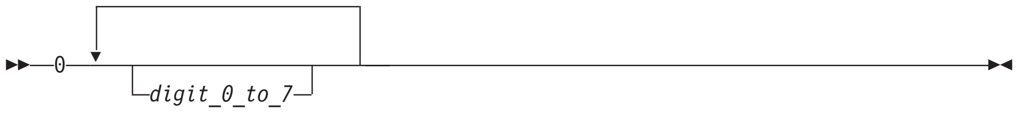
```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

**Related Information**

- “Integer Constants” on page 14
- “Decimal Constants” on page 15
- “Octal Constants”
- “Integers” on page 40

## Octal Constants

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



**Data Type**

See Table 4 on page 14 for a complete description of the data types of octal constants.

**Examples**

The following are examples of octal constants:

```
0
0125
034673
03245
```

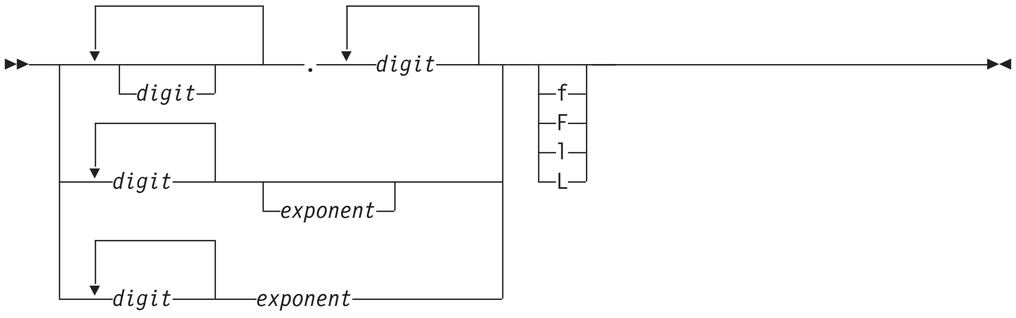
**Related Information**

- “Integer Constants” on page 14

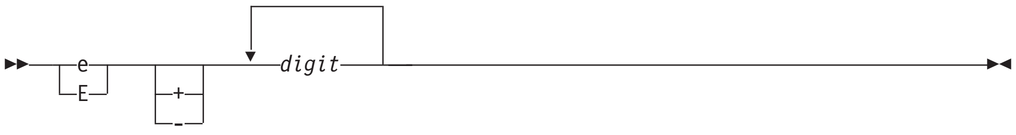
- “Decimal Constants” on page 15
- “Hexadecimal Constants” on page 15
- “Integers” on page 40

## Floating-Point Constants

A **floating-point constant** consists of an integral part, a decimal point, a fractional part, an exponent part, and an optional suffix. Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part (but not both). A floating-point constant has the form:



The exponent part consists of `e` or `E`, followed optionally by a sign and a decimal number. An exponent has the form:



### Value

The floating-point constant `8.45e+3` evaluates as follows:

$8.45 * 10^3 = 8450.0$

If a floating-point constant is too large or too small, the result is undefined.

### Data Type

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

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

### Examples

| Floating-Point Constant | Value |
|---|---|
| `5.3876e4` | `53,876` |

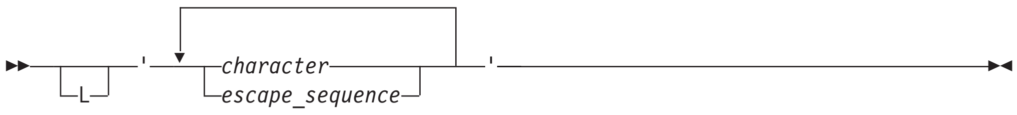
| Floating-Point Constant | Value |
|---|---|
| `4e-11` | `0.00000000004` |
| `1e+5` | `100,000` |
| `7.321E-3` | `0.007321` |
| `3.2E+4` | `32,000` |
| `0.5e-6` | `0.0000005` |
| `0.45` | `0.45` |
| `6.e10` | `60,000,000,000` |

**Related Information**

- "Floating-Point Variables" on page 39

## Character Constants

A **character constant** contains a sequence of characters or escape sequences that are enclosed in single quotation mark symbols. A character constant has the form:



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

### Value

The value of a character constant that contains a single character is the numeric representation of the character in the character set that is used at run time. The value of a wide character constant containing a single multibyte character is the code for that character, as defined by the `mbtowc()` function (see the *ILE C for AS/400 Run-Time Library Reference*). If the character constant contains more than one character, the last 4 bytes represent the character constant.

### Data Type

A character constant has type `int`. A wide character constant is represented by a double-byte character of type `wchar_t`, as defined in the `<stddef.h>` include file. Multibyte characters represent character sets that go beyond the single byte character set. Each multibyte character can contain up to 4 bytes.

### Restrictions

You can represent the double quotation mark symbol by itself. However, you must use the backslash symbol followed by a single quotation mark symbol (`\'` escape sequence) to represent the single quotation mark symbol. You can represent the new-line character by the `\n` new-line escape sequence. You can represent the backslash character by the `\\` backslash escape sequence.

### Examples

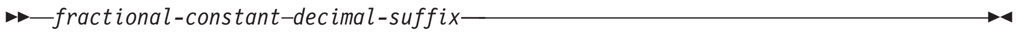
| | |
|---|---|
| **'a'** | `'\'` |
| **'0'** | `'('` |
| **'x'** | `'\n'` |
| **'7'** | `'\117'` |
| **'b'** | `L'b'` (character `b` stored a wide character) |

**Related Information**

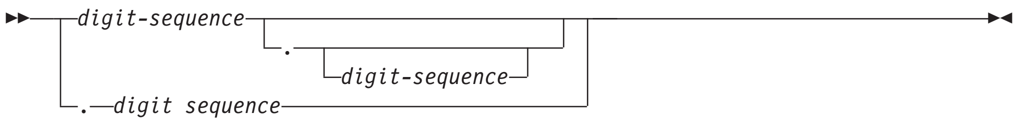
- “Strings” on page 20
- “Escape Sequences” on page 10
- “Integers” on page 40

## Packed Decimal Constants

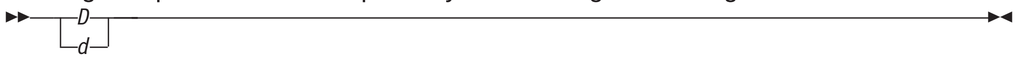
A **packed decimal constant** has a numeric part and a suffix that specifies its type. Each packed decimal constant has the attributes **number of digits** (size) and **number of decimal places** (precision). No leading or trailing zeros are stripped off when determining the size and the precision. A packed decimal constant has the form:



The fractional constant has the form:
The decimal-suffix has the form:



The digit sequence is made up of any decimal digits 0 through 9.



**Value**

The fractional-constant is a component of the numeric part. It may include a digit sequence that represents the integral part, followed by a decimal point(.), followed by a digit sequence that represents the fractional part. Either the integral part or the fractional part shall be present, or both. The decimal-suffix d or D identifies the constant as a packed decimal constant.

**Data Type**

A packed decimal constant has the type `decimal(n,p)`, where `n` represents the number of digits of the packed decimal constant, and `p` is the number of digits in the fractional part.

**Examples**

The following example shows you some packed decimal constants and their corresponding attributes:

| **Packed Decimal Constant** | **(Size,Precision)** |
|---|---|
| 1234567890123456D | (16,0) |
| 12345678.12345678D | (16,8) |
| 12345678.d | (8,0) |
| .1234567890d | (10,10) |
| 12345.99d | (7,2) |
| 000123.990d | (9,3) |
| 0.00D | (3,2) |

You can use packed decimal constants to define macros such as the following:

```
#define x ( 12345.67d )
```

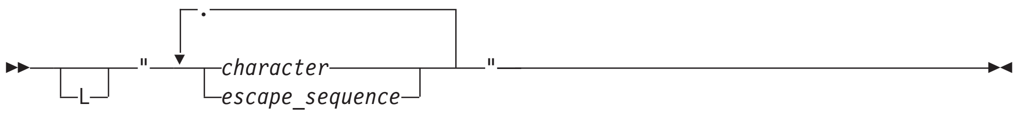
You can initialize a packed decimal variable with a packed decimal constant. For example:

```
#include <decimal.h>
decimal(5,2) x = 123.45d;
decimal(DEC_DIG, DEC_PRECISION) a =.000000000000000000000000000000005d;
decimal(6,2) b[] = {1.2d, 2d, 1234.56d, -33d};
decimal(28,20) a = -12.123456789d;
```

## Strings

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

A string constant has the form:



### Value

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

If you want to continue a string on the next line, use the line continuation sequence (\ symbol immediately followed by a new-line character).

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals are concatenated to produce a single string. (The null character of the first string will no longer exist after the concatenation.) You cannot concatenate a wide string constant with a character string constant.

Multiple spaces that are contained within a string constant are held.

### Data Type

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

## Restrictions

You can use the escape sequence `\n` to represent a new-line character as part of the string.

You can use the escape sequence `\\` to represent a backslash character as part of the string.

You can represent the single quotation mark symbol by itself `'`, but you use the escape sequence `\"` to represent the double quotation mark symbol.

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

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

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

```
#pragma strings( readonly )
#include <stdio.h>
/* Since "readonly" strings are being used, the string literal   */
/* "ABC" will only be allocated in storage once.  The address of */
/* that location will be stored in both 'p1' and 'p2'.           */
 char *p1 = "ABC";
 char *p2 = "ABC";
main() {
                                  /* change the first character   */
  *p1 = 'a';                      /* pointed to by 'p1'           */
                                  /* output the string pointed to */
  printf("p2:  %3.3s \n", p2 );   /* by 'p2' to see that it has   */
                                  /* also changed.                */
}
```

The output from this example would be:

```
    p2:  aBC
```

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

```
    p2:  ABC
```

## Example

```
char titles[ ] = "Bach's \"Jesu, Joy of Man's Desiring\"";
char *mail_addr = "Last Name    First Name    MI   Street Address   \
   City     Province   Postal code ";
char *temp_string = "abc" "def" "ghi";  /* *temp_string = "abcdefghi\0" */
```

**Related Information**

- “Character Constants” on page 18
- “Escape Sequences” on page 10
- “Characters” on page 38
- “Arrays” on page 47

## Enumeration Constants

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

### Value

Each enumeration constant has an integer value. You can use an enumeration constant anywhere an integer constant is allowed. The value of the constant is determined in the following way:

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

### Data Type

An enumeration constant has type `int`.

### Examples

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

```
enum grain { oats, wheat, barley, corn, rice };
   /*         0      1      2      3     4         */
enum grain { oats=1, wheat, barley, corn, rice };
   /*         1        2      3      4     5       */
enum grain { oats, wheat=10, barley, corn=20, rice };
   /*          0     10        11     20       21  */
```

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

```
enum status { run, delete=5, suspend, resume, hold=6 };
```

**Related Information**

- “Enumerations” on page 44
- “Integers” on page 40

# Chapter 3. Declarations and Definitions

This chapter describes the C language definitions and declarations for data objects and functions. A **declaration** establishes the names and characteristics of data objects and functions used in a program. A **definition** is a declaration that allocates storage for data objects or specifies the body for a function.

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

*Table 5. Examples of Declarations and Definitions*

| **Declarations** | **Declarations and Definitions** |
|---|---|
| `extern double pi;` | `double pi = 3.14159265;` |
| `float square(float x);` | `float square(float x) { return x*x; }` |
| `struct payroll;` | `struct payroll { char *name; float salary; } employee;` |

The declaration for a data object can include the following components:

- Storage class, described in “Storage Class Specifiers” on page 25
- Type, described in “Types” on page 38
- Qualifier and Declarator, described in “Declarators” on page 33
- Initializer, described in “Initializers” on page 37.

The “Chapter 4. Functions” on page 69 describes the function declarations.

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

- Scope, which describes the visibility of an identifier in a block or source file. For a complete description of scope, see “Scope” on page 4.
- Storage duration, which describes when the system allocates and frees storage for a data object.
- Linkage, which describes the association between two identical identifiers. See “Linkage” on page 6 for more information.
- Type, which describes the kind of data the object is to represent.

## Block Scope Data Declarations

A **block scope data declaration** can only be placed at the beginning of a block. It describes a variable and makes that variable accessible to the current block. All block scope declarations that do not have the `extern` storage class specifier are definitions and allocate storage for that object.

You can define a data object with block scope with any one of the following storage class specifiers:

- `auto`
- `register`
- `static`
- `extern`

If you do not specify a storage class specifier in a block-scope data declaration, the default storage class specifier `auto` is used. If you specify a storage class specifier, you can omit the type specifier. If you omit the type specifier, all variables that are declared in that declaration will have the type `int`.

**Initialization**

You cannot initialize a variable that is declared in a block scope data declaration and has the `extern` storage class specifier.

The types of variables you can initialize and the values that uninitialized variables receive vary for each storage class specifier.

**Storage**

Declarations with the `auto` or `register` storage class specifier result in automatic storage duration. Declarations with the `extern` or `static` storage class specifier result in static storage duration.

**Related Information**
- “auto Storage Class Specifier” on page 25
- “register Storage Class Specifier” on page 30
- “extern Storage Class Specifier” on page 28
- “static Storage Class Specifier” on page 31
- “Declarators” on page 33
- “Initializers” on page 37
- “Types” on page 38

## File Scope Data Declarations

A **file scope data declaration** appears outside any block. It describes a variable and makes that variable accessible to all functions that are in the same file and whose definitions appear after the declaration.

A **file scope data definition** is a data declaration at file scope that also causes the system to allocate storage for that variable. All objects whose identifiers are declared at file scope have static storage duration.

You can use a file scope data declaration to declare variables that you want several functions to access.

The only storage class specifiers you can place in a file scope data declaration are `static` and `extern`.

If you specify `static`, all variables that are defined in it have internal linkage. If you do not specify `static`, all variables that are defined in it have external linkage.

If you specify the storage class `static` or `extern`, you can omit the type specifier. If you omit the type specifier, all variables that are defined in that declaration receive the type `int`.

**Initialization**

You can initialize any object with file scope. If you do not initialize a file scope variable, its initial value is zero of the appropriate type. If you do initialize it, the initializer must be described by a constant expression. Or it must reduce to the address of a previously declared variable at file scope, possibly modified by a constant expression. Initialization of all variables at file scope takes place before the `main()` function begins processing.

**Storage**

All objects with file scope data declarations have static storage duration. The system allocates memory for all file scope variables when the program begins processing and frees it when the program is finished processing.

**Related Information**

- “extern Storage Class Specifier” on page 28
- “static Storage Class Specifier” on page 31
- “Declarators” on page 33
- “Initializers” on page 37
- “Types” on page 38

## Storage Class Specifiers

This section describes C language object declarations and the storage durations that are associated with the objects and the linkage of their identifiers. The storage class specifier that is used within the declaration determines the following:

- Whether the object has internal, external, or no linkage.
- Whether the object has static class (storage for the object is maintained throughout program execution) or automatic class (storage for the object is maintained only during the execution of the block in which the identifier of the object is defined) storage duration.
- Whether the object is stored in memory or in a register, if available.
- Whether the object receives the default initial value `0` or an indeterminate default initial value.

For a function, the storage class specifier determines the function’s linkage.

## auto Storage Class Specifier

The `auto` storage class specifier enables you to define a variable with automatic storage; its use and storage are restricted to the current block. The storage class keyword `auto` is optional in a data declaration and is forbidden in a parameter declaration. A variable that has the `auto` storage class specifier must be declared within a block. It cannot be used for file scope declarations.

The following example lines declare variables that have the `auto` storage class specifier:

```
auto int counter;
auto char letter = 'k';
```

### Initialization

You can initialize any auto variable except parameters. If you do not initialize an automatic object, its value is undefined. If you provide an initial value, the expression that represents the initial value can be any valid C expression. For aggregates or unions, the initial value must be a valid constant expression. The object is then set to that initial value each time the program block that contains the object's definition is entered.

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

### Storage

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

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

### Usage

Declaring variables with the auto storage class specifier can decrease the amount of memory that is required for program processing. This occurs because auto variables require storage only while they actually are needed. Generally, it is not a good idea to use automatic storage for large objects. The operating system will take time to allocate or deallocate large amounts of storage.

### Examples

The following program shows the scope and initialization of auto variables. The function main() defines two variables, each named auto_var. The first definition occurs on line 8. The second definition occurs in a nested block on line 11. While the nested block runs, only the auto_var created by the second definition is available. During the rest of the program, only the auto_var created by the first definition is available.

```
 1                           /* program to illustrate auto variables */
 2
 3   #include <stdio.h>
 4
 5   int main(void)
 6   {
 7      void call_func(int passed_var);
 8      auto int auto_var = 1;  /* first definition of auto_var  */
 9
10      {
11         int auto_var = 2;    /* second definition of auto_var */
12         printf("inner auto_var = %d\n", auto_var);
13      }
14      call_func(auto_var);
15      printf("outer auto_var = %d\n", auto_var);
16   }
17
18   void call_func(int passed_var)
19   {
   printf("passed_var = %d\n", passed_var);
   passed_var = 3;
   printf("passed_var = %d\n", passed_var);
}
```

This program produces the following output:

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

The following example uses an array that has the storage class `auto` to pass a character string to the function `sort`. The C language views an array name that appears without subscripts (for example, `string` instead of `string[0]`) as a pointer. Thus, the `sort()` function receives the address of the character string, rather than the contents of the array. The address enables the `sort` function to change the values of the elements in the array.

```
/* Sorted string program */
#include <stdio.h>
int main(void)
{
   void sort(char *array, int n);
   char string[75];
   int length;
   printf("Enter letters:\n");
   scanf("%74s", string);
   length = strlen(string);
   sort(string,length);
   printf("The sorted string is: %s\n", string);
}
void sort(char *array, int n)
{
   int gap, i, j, temp;
   for (gap = n / 2; gap > 0; gap /= 2)
      for (i = gap; i < n; i++)
         for (j = i - gap; j >= 0 && array[j] > array[j + gap];
            j -= gap)
         {
            temp = array[j];
            array[j] = array[j + gap];
            array[j + gap] = temp;
         }
}
```

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

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

**Related Information**

- “register Storage Class Specifier” on page 30
- “Block Scope Data Declarations” on page 23
- “Address &” on page 88

## extern Storage Class Specifier

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

You can distinguish an `extern` declaration from an `extern` definition by the presence of the keyword `extern` and the absence of an initial value. If the keyword `extern` is absent or if there is an initial value, the declaration is also a definition; otherwise, it is just a declaration. An `extern` definition can appear only outside a function definition. Only one declaration of the variable without the keyword `extern` can be used. That declaration is the definition of the storage for the variable.

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

## Declaration

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

If you choose not to specify a storage class specifier, the function will have external linkage. So if you include a declaration for the same function with the storage class specifier `static` before the declaration with no storage class specifier, an error will be noted because of the incompatible declarations. If you had included the `extern` storage class specifier on the original declaration, there would be no error, and the function would have internal linkage.

### Initialization of Variables

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

- Appear as part of the definition and the initial value must be described by a constant expression.
- Reduce to the address of a previously declared object with static storage duration. This object may be changed by a constant expression.

If you do not initialize an `extern` variable, its initial value is zero of the appropriate type. By the start of program processing, initialization of an `extern` object is completed.

### Storage for Objects

`extern` objects have static storage duration. Memory is allocated for `extern` objects before the `main()` function begins processing. When the program finishes processing, the storage is freed.

### Examples

The following program shows the linkage of `extern` objects and functions. The `extern` object `total` is declared on line `12` of `File 1` and on line `11` of `File 2`. The definition of the external object `total` appears in `File 3`. The extern function `tally()` is defined in `File 2`. The function `tally()` can be placed in the same file as `main()` or in a different file. Because `main()` precedes these definitions and main uses both `total` and `tally()`, `main()` declares `tally()` on line `11` and `total` on line `12`. Each file needs to be compiled using CRTCMOD and then bound using CRTPGM.

**File 1**

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

**File 2**

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

**File 3**

```
 1   float total;
```

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

```
/* Sorted string program */
#include <stdio.h>
char string[75];
int length;

int main(void)
{
   void sort(void);
   printf("Enter letters:\n");
   scanf("%s", string);
   length = strlen(string);
   sort();
   printf("The sorted string is: %s\n", string);
}

void sort(void)
{
   int gap, i, j, temp;
   for (gap = length / 2; gap > 0; gap /= 2)
      for (i = gap; i < length; i++)
         for (j = i - gap;
              j >= 0 && string[j] > string[j + gap];
              j -= gap)
         {

            temp = string[j];
            string[j] = string[j + gap];
            string[j + gap] = temp;
         }
}
```

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

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

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

```
static int var1;
   ⋮
extern int var1;
```

**Related Information**

- “File Scope Data Declarations” on page 24
- “Function Definition” on page 70
- “Function Declarator” on page 72
- “Constant Expression” on page 82

## register Storage Class Specifier

The `register` storage class specifier indicates to the compiler within a file scope data definition or a parameter declaration that the object being described will be heavily used (such as a loop control variable). If possible, the compiler will place the object into a machine register for fast access storage. The storage class keyword `register` is required in a data definition and in a parameter declaration that describes an object that has the `register` storage class. An object that has the `register` storage class specifier must be defined within a block or declared as a parameter to a function.

The following example lines define automatic storage duration objects that use the `register` storage class specifier:

```
register int score1 = 0, score2 = 0;
register unsigned char code = 'A';
register int *element = &order[0];
```

### Initialization

You can initialize any `register` object except parameters. If you do not initialize an automatic object, its value is undefined. If you provide an initial value, the expression that represents the initial value can be any valid C expression. For aggregates or unions, the initial value must be a valid constant expression. The variable is then set to that initial value each time the program block that contains the object's definition is entered.

### Storage

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

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

The `register` storage class specifier indicates that the object is heavily used. It also indicates to the compiler that the value of the object should reside in a machine register. Not all `register` variables are actually placed in registers.

If the compiler does not allocate a machine register for a `register` object, the object is treated as having the storage class specifier `auto`. Because of the limited size and number of registers available on most systems, few variables can be stored in registers at the same time. In C programs, even if a `register` variable is treated as a variable with storage class `auto`, the address of the variable cannot be taken.

### Restrictions

You cannot use the `register` storage class specifier on file scope data declarations.

### Related Information

- "auto Storage Class Specifier" on page 25
- "Block Scope Data Declarations" on page 23
- "Parameter Declaration List" on page 73
- "Address &" on page 88

## static Storage Class Specifier

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

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

### Initialization of Variables

You can initialize any `static` object. If you do not provide an initial value, the object receives the value of zeros of the appropriate type. If you initialize a `static` object, a constant expression must describe the initializer. Otherwise, the initializer must reduce to the address of a previously declared `extern` or `static` object, possibly changed by a constant expression.

### Storage for Object

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

### Restrictions

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

### Usage

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

### Examples

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

#### File 1

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

#### File 2

```
#include <stdio.h>
static int stat_var = 2;
void var_print(void)
{
    printf("file2 stat_var = %d\n", stat_var);
}
```

The preceding program produces the following output:

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

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

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

The preceding program produces the following output:

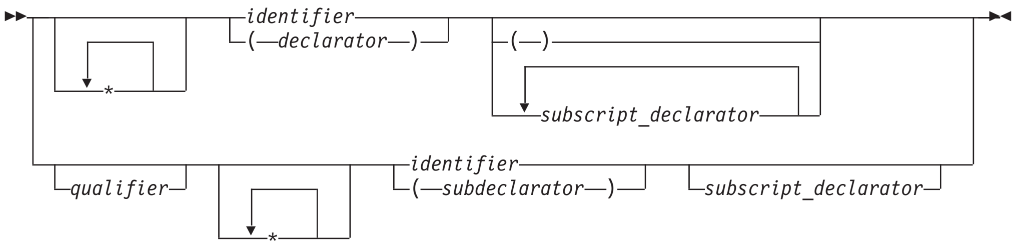
```
stat_var = 1 auto_var = 1
stat_var = 2 auto_var = 1
stat_var = 3 auto_var = 1
stat_var = 4 auto_var = 1
```

**Related Information**

- "Block Scope Data Declarations" on page 23
- "File Scope Data Declarations" on page 24
- "Function Definition" on page 70
- "Function Declarator" on page 72

## Declarators

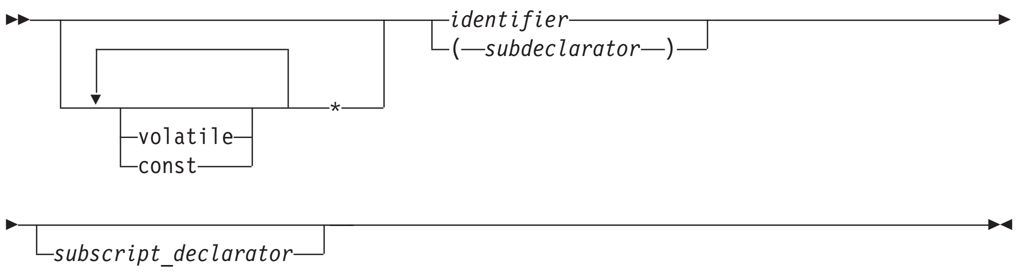
A **declarator** designates a data object or function. Declarators appear in all data definitions and declarations and in some type definitions. A declarator has the form:



A qualifier is one of: const, volatile, or _Packed.

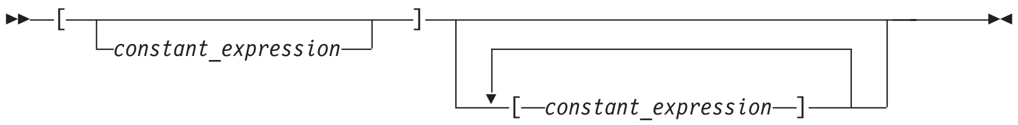
You cannot declare or define a volatile or const function.

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



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

A subscript declarator has the form:



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

```
auto char initial;
```

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

You can define or declare an aggregate by using a declarator containing an identifier to name the data object, and some combination of symbols and identifiers to describe the data types represented by the object. An aggregate is a structure, union, or array. The following declaration uses `compute[5]` as the declarator:

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

### Examples

The following table describes some declarators:

*Table 6. Example Declarators*

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

*Table 6. Example Declarators (continued)*

| Example | Description |
|---|---|
| `_Packed struct struct_type s` | `s` is a packed structure of type `struct_type`. |

**Related Information**

- "volatile and const Qualifiers"
- "_Packed Qualifier" on page 36
- "Chapter 3. Declarations and Definitions" on page 23
- "Arrays" on page 47
- "Enumerations" on page 44
- "Pointers" on page 52
- "Structures" on page 57
- "Unions" on page 62

## volatile and const Qualifiers

The `volatile` qualifier maintains the intent of the original expression with respect to stores and fetches of `volatile` objects. The `volatile` qualifier is useful for data objects that have values that may be changed in ways unknown to your program (such as the system clock). Portions of an expression that refer to `volatile` objects are not to be changed or removed.

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

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

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

For a pointer to a `volatile` or `const` data object, you must place the keyword before the type specifier. For example:

```
volatile int *x;         /* x is a pointer to a volatile int  */
const int *y;            /* y is a pointer to a const int  */
```

You can assign a value to the `int * const y` but not to `const int * y`.

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

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

This definition provides the same storage as:

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

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

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

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

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

You cannot declare or define a volatile or const function. However, you can define or declare a function that returns a pointer to a volatile or const object.

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

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

## _Packed Qualifier

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

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

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

```
_Packed struct struct_type s     /* s is a packed structure of type struct_type  */
```

If you specify the `_Packed` qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the contained structure or union.

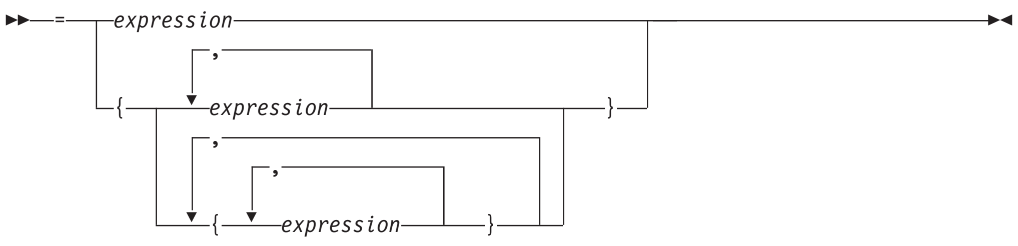
**Related Information**

- “Arrays” on page 47
- “Enumerations” on page 44
- “Pointers” on page 52
- “Structures” on page 57
- “Unions” on page 62

## Initializers

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

An initializer has the form:



The initializer consists of the `=` symbol that is followed by an initial *expression* or a braced list of initial expressions that are separated by commas. The number of initializers should not be more than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

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

```
int group = 3;
```

For unions and structures, the set of initial expressions must be enclosed in braces (`{ }`). If the initializer of a character string is a string literal, the `{ }` are optional. Commas must separate the individual expressions. Groups of expressions can be enclosed in braces and separated by commas. The number of initializers must be less than or equal to the number of initialized objects. In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if you explicitly initialized them to zero.

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

The initial values of `grid` are:

| Element | Value | Element | Value |
|---|---|---|---|
| `grid[0] [0]` | `0` | `grid[1] [2]` | `1` |

| Element | Value | Element | Value |
|---|---|---|---|
| `grid[0] [1]` | `0` | `grid[1] [3]` | `1` |
| `grid[0] [2]` | `0` | `grid[2] [0]` | `0` |
| `grid[0] [3]` | `1` | `grid[2] [1]` | `0` |
| `grid[1] [0]` | `0` | `grid[2] [2]` | `0` |
| `grid[1] [1]` | `0` | `grid[2] [3]` | `0` |

Initialization considerations for each data type are described in the section for that data type.

## Types

The ILE C data types are:
- Characters
- Floating-Point Numbers
- Integers
- Packed Decimal

From these types, you can derive the following:
- Arrays
- Pointers
- Enumerations
- Structures
- Unions

## Characters

The C language contains three basic character data types: `char`, `signed char`, and `unsigned char`. These data types provide enough storage to hold any member of the character set that is used at run time.

An `unsigned char` represents a `char` . For information on changing this default, see “chars” on page 158. If it does not matter whether a `char` data object is `signed` or `unsigned`, you can declare the object as having the data type `char`. Otherwise, explicitly declare `signed char` or `unsigned char`. When a `char` (that is `signed` or `unsigned`) is widened to an `int`, its value is preserved.

To declare a data object that have a character data type, place a *char specifier* in the type specifier position of the declaration. The `char` specifier has the form:

```
►►──┬────────────┬──char──────────────────────────────────────────►◄
    ├─unsigned───┤
    └─signed─────┘
```

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

**Examples**

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

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

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

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

You can use the `char` specifier in variable definitions to define such variables as: arrays of characters, pointers to characters, and arrays of pointers to characters.

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

```
char *string_pointer;
```

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

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

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

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

**Related Information**

- “Arrays” on page 47
- “Pointers” on page 52
- “Character Constants” on page 18
- “Assignment Expression” on page 102

## Floating-Point Variables

The C language defines three types of floating-point variables: `float`, `double`, and `long double`.

The storage size of a `float` variable is less than or equal to the storage size of a `double` variable. The storage size of a `double` is equal to the storage size of a `long double` variable. Thus, the following expression always evaluates to `1` (true):

```
sizeof(float) <= sizeof(double) && sizeof(double) == sizeof(long double)
```

The size of a `float` is 4 bytes. The size of `double` and `long double` is 8 bytes. The size of `float`, `double`, and `long double` may vary for different compilers.

To declare a data object that has a floating-point type, use the **float specifier**. The float specifier has the form:

```
►►──┬─float───────┬──────────────────────────────────────────►◄
    ├─double──────┤
    └─long double─┘
```

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

### Examples

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

```
double pi;
```

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

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

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

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

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

```
extern long double maximum;
```

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

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

### Related Information

- “Floating-Point Constants” on page 17
- “Assignment Expression” on page 102
- “Integers”

## Integers

The C language supports eight types of integer variables:

- `short int`, `short`, `signed short`, or `signed short int`
- `signed`, `signed int`, or `int` (In some cases, no type specifier is needed; see “Block Scope Data Declarations” on page 23 and “File Scope Data Declarations” on page 24.)
- `long int`, `long`, `signed long`, or `signed long int`
- `unsigned short int` or `unsigned short`
- `unsigned` or `unsigned int`
- `unsigned long int` or `unsigned long`
- `long long int`, `long long`, `signed long long`, or `signed long long int`
- `unsigned long long int` or `unsigned long long`

The storage size of a `short` type is less than or equal to the storage size of an `int` variable. The storage size of an `int` variable is equal to the storage size of a `long` variable. The storage size of a `long` variable is less than or equal to the storage size of a `long long` variable. Thus, the following expression always evaluates to `1` (true):

```
    sizeof(short) <= sizeof(int)
&& sizeof(int) == sizeof(long)
&& sizeof(long) <= sizeof(long long)
```

ILE C provides three sizes for integer data types. Objects that have type `short` are two bytes of storage in length. Objects that have type `long` are four bytes of storage in length. Objects that have type `long long` are eight bytes of storage in length. An `int` requires four bytes of storage.

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

**Note:** A program that uses `long long` integer data type will only compile with one of the following:

1. The keyword LANGLVL(*EXTENDED) on the CRTCMOD command
2. No #pragma langlvl directive in the program source and the keyword LANGLVL(*SOURCE) on the CRTCMOD command (the default setting)
3. #pragma langlvl(EXTENDED) in the program source and the keyword LANGLVL(*SOURCE) on the CRTCMOD command

The `long long` integer data type is not currently a universal standard. Its use may restrict portability to some other platforms.

To declare a data object that has an integer data type, place an **int specifier** in the type specifier position of the declaration. The `int` specifier has the form:

```
►►──┬─┬────────────┬─────┬─int──────────────────────┬─┬──────────────────────────►◄
    │ ├─unsigned─┤     ├─short──┬───────┬────────────┤ │
    │ └─signed───┘     │        └─int─┘             │ │
    │                  └─long───┬──────┬───┬─────┬──┘ │
    │                           └─long─┘   └─int─┘    │
    └─┬─signed───┬────────────────────────────────────┘
      └─unsigned─┘
```

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

**Examples**

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

```
short int flag;
```

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

```
int result;
```

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

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

The following example defines the `unsigned long long int` variable `temp` as having the initial value 4388888344294967296:

```
unsigned long long temp = 4388888344294967296ULL;
```

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

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

**Related Information**

- "Integer Constants" on page 14
- "Decimal Constants" on page 15
- "Octal Constants" on page 16
- "Hexadecimal Constants" on page 15

## Packed Decimal

The size of a packed decimal type can vary from 1 byte to 16 bytes. Each packed decimal digit occupies half a byte. In addition, half a byte is used for the sign. The number of bytes used by decimal(*n*,*p*) is the smallest whole number greater than or equal to (n+1)/2. For example, sizeof(decimal(*n*,*p*)) > = ceil((*n*+ 1)/2).

To declare a data object that has a packed decimal data type, include the <decimal.h> header file in the C source. Use a packed decimal specifier in the type specifier position of the declaration. The packed decimal specifier has form:

```
►►──decimal──(──constant_expression──┬──────────────────────┬──)──────────────►◄
                                     └─,constant_expression─┘
```

The *constant_expression* is evaluated as a positive integral constant expression. The second *constant_expression* is optional; the default value is 0.

The type specifier decimal(*n*,*p*) is used to declare variables of packed decimal data type.

decimal(*n*,*p*) designates a packed decimal number with *n* digits and *p* decimal places. *n* is the number of digits for the integral part and the fractional part. *p* is the number of digits for the decimal (fractional) part. The maximum number of digits *n* and the maximum precision *p* is 31 digits. For example, decimal(5,2) represents a number such as 123.45 where n=5 and p=2.

*n* and *p* have a range of allowed values according to the following rules:

$1 \leq n \leq 31$
$0 \leq p \leq n$

The range applies to packed decimal constants and variables. *n* and *p* can be integral constant expressions, integral constants, hexadecimal constants, or octal constants.

The type specifiers that are `long`, `short`, `signed`, and `unsigned` cannot be used with decimal(*n*,*p*).

**Examples**

The following example shows how to declare a variable of packed decimal data type:

```
#include <decimal.h>    decimal(10,2)  x;
    decimal(5,0)   y;  /* p is optional. p is specified as 0.                 */
    decimal(5)     z;  /* p is optional. p is not specified. The default is 0. */
    decimal(18,10) *ptr;
    decimal(8,2)   arr[100];
```

In this example, x can have values between -99999999.99D to +99999999.99D; *y*and *z*can have values between -99999D to +99999D. *ptr* is a pointer to type decimal(18,10), and *arr* is an array of 100 elements, where each element is of type decimal(8,2).

**Related Information**

- “Packed Decimal Constants” on page 19

## void Type

The `void` data type always represents an empty set of values. The keyword for this type is `void`. When a function does not return a value, you should use `void` as the type specifier in the function definition and declaration. Only a pointer can be declared with the type specifier `void`.

**Examples**

On line `4` of the following example, the function `find_max()` is declared as having type `void`. Lines `12` through `23` contain the complete definition of `find_max()`.

**Note:** The use of the `sizeof` operator in line `10` is a standard method of determining the number of elements in an array.

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

**Related Information**

- “Cast” on page 89

## Enumerations

An **enumeration** data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement. As well, you can separate the declaration of the enumeration data type from all variable definitions. The identifier that is associated with the data type (not an object) is a tag.

## Declaring an Enumeration Data Type

An enumeration type declaration contains the `enum` keyword that is followed by an identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator. An enumeration type declaration has the form:

```
                                 ┌─,────────────────┐
                                 ▼                  │
►►──enum──┬──────────────┬──{────enumerator─────────┴──}──────────────────►◄
          └─identifier───┘
```

The keyword `enum` that is followed by the identifier, names the data type (like the tag on a `struct` data type). The list of enumerators provides the data type with a set of values. Each enumerator represents an integer value. To conserve space, enumerations may be stored in spaces smaller than that of an `int`. By default the size of an enumeration is the minimum that is required to represent all the values of the enumeration. For example, if all the values of the enumeration can be represented with one byte, the size of that enumeration will be one byte. An enumeration type can have a size of 1, 2, or 4 bytes, depending on the values in the enumeration.

**Note:** The #pragma enumsize directive can be used to change the size that is used by the ILE C compiler to represent enumerations.

An enumerator has the form:

```
►►──identifier──┬──────────────────────────────────┬──────────────────►◄
                └─=──integral_constant_expression──┘
```

An **enumeration constant** is the identifier in an enumerator. You can use an enumeration constant anywhere an integer constant is allowed. The following rules determine the value of an enumeration constant:

1. If an `=` (equal sign) and a constant expression follow the identifier, the identifier represents the value of the constant expression.
2. If the enumerator is the leftmost value in the list, the identifier represents the value `0`.
3. Otherwise, the identifier represents the integer value that is one greater than the value that is represented by the preceding enumerator.

The following example declares the enumeration tag `status`:

```
enum status { run, create, delete=5, suspend };
```

The data type `status` represents the following values:

| Enumeration Constant | Integer Representation |
|---|---|
| run | 0 |
| create | 1 |
| delete | 5 |
| suspend | 6 |

Each enumeration identifier must be unique within the block or the file where the enumeration data type is declared. In the following example, the declarations of `average` on line `4` and of `poor` on line `5` cause compiler error messages:

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

## Defining a Variable That Has an Enumeration Type

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

The first line of the following example declares the enumeration tag `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (`2`). The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`:

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

The initializer for an enumeration variable contains the `=` symbol that is followed by an expression. The expression must evaluate to an `int` value.

## Example of Defining an Enumeration Type and Enumeration Objects

You can place a type definition and a variable definition in one statement by placing a declarator and an optional initializer after the type definition. If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the declaration. For example:

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

This example is equivalent to the following two declarations:

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

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier `register`, the data type `enum score`, and the initial value `3` (or `good`).

If you combine a data type definition with the definitions of all variables that have that data type, you can leave the data type unnamed. For example:

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

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

**Example**

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

```
#include <stdio.h>
enum days {
            Monday=1, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday
          } weekday;
void french(enum days);
int main(void)
{
   int num;
   printf("Enter an integer for the day of the week.  "
          "Mon=1,...,Sun=7\n");
   scanf("%d", &num);
   weekday=num;
   french(weekday);
}
void french(enum days weekday)
{
   switch (weekday)
   {
      case Monday:
         printf("Le jour de la semaine est lundi.\n");
         break;
      case Tuesday:
         printf("Le jour de la semaine est mardi.\n");
         break;
      case Wednesday:
         printf("Le jour de la semaine est mercredi.\n");
         break;
      case Thursday:
         printf("Le jour de la semaine est jeudi.\n");
         break;
      case Friday:
         printf("Le jour de la semaine est vendredi.\n");
         break;
      case Saturday:
         printf("Le jour de la semaine est samedi.\n");
         break;
      case Sunday:
         printf("Le jour de la semaine est dimanche.\n");
         break;
      default:
         printf("C'est le mauvais jour.\n");
   }
}
```

**Related Information**

- “Enumeration Constants” on page 22
- “Constant Expression” on page 82
- “Identifiers” on page 12

- “enumsize” on page 162

## Arrays

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

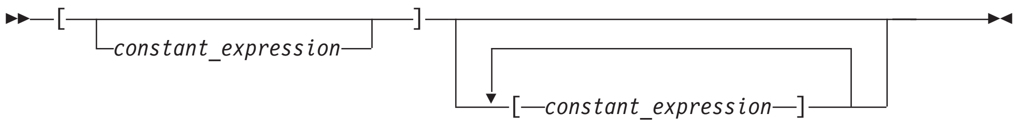
### Type Specifiers of Arrays

You can use any type specifier in an array definition or declaration. Thus, array elements can be of any data type, except function. You can, however, declare an array of pointers to functions.

### Declarators of Arrays

The declarator contains an identifier followed by a **subscript declarator**. An identifier can be preceded by an asterisk (`*`) to declare an array of pointers.

The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. A subscript declarator has the form:



Each bracketed expression describes a different dimension. The constant expression must have an integral value. The value of the constant expression determines the number of elements in that dimension. The following example defines a one-dimensional array that contains four elements that have type `char`:

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

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

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

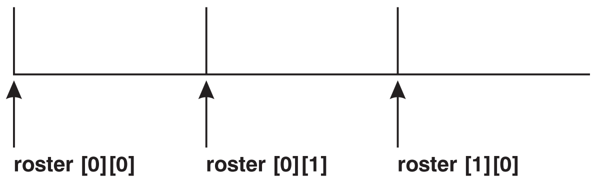
The following example defines a two-dimensional array that contains six elements of type `int`:

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

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

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

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



You can leave the first (and only the first) set of subscript brackets empty in array definitions that contain initializations, `extern` declarations, and parameter declarations.

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

## Initializers of Arrays

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

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

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

The array `number` contains the following values:

| Element | Value |
|---|---|
| **number[0]** | 5 |
| **number[1]** | 7 |
| **number[2]** | 2 |

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

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

The values of `number1` are:

| Element | Value |
|---|---|
| **number1[0]** | 5 |
| **number1[1]** | 7 |
| **number1[2]** | 0 |

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

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

The compiler gives `item` the five initialized elements:

| Element | Value |
|---|---|
| **item[0]** | 1 |
| **item[1]** | 2 |
| **item[2]** | 3 |
| **item[3]** | 4 |
| **item[4]** | 5 |

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

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

If you specify a string constant, the null character (`\0`) is placed at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

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

These definitions create the following elements:

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

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

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

You can initialize a multidimensional array by:

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

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

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

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

You can use nested braces to initialize dimensions and elements in a dimension selectively.

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

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

The initial values of `matrix` are:

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

**Note:** You should place braces around each dimension (**fully braced**). Otherwise, only use one set of braces to enclose the entire set of initializers (**unbraced**). Avoid putting braces around some elements and not others.

An unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the array's first element. For more information, see "Primary Expression" on page 83.

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

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

### Examples

The following example shows a parameter declaration for a one-dimensional array:

```
   test(int y[ ])
   {
      .
      .
      .
   }
```

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

```
/* Example of one-dimensional arrays. */
#include <stdio.h>
#define  ARR_SIZE  5
int main(void)
{
  static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
  auto float total;
  int i;
  for (i = 0; i < ARR_SIZE; i++)
  {
    printf("price = $%.2f\n", prices[i]);
  }
  printf("\n");
  for (i = 0; i < ARR_SIZE; i++)
  {
    total = prices[i] * 1.05;
    printf("total = $%.2f\n", total);
  }
}
```

The first `for` statement prints the values `prices`'s elements. The second `for` statement adds five percent to the value of each element of `prices`, and assigns the result to `total`, and prints the value of `total`. The example produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86
total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

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

```
/* example of a multidimensional array */
#include <stdio.h>
#define  NUM_ROW     3
#define  NUM_COLUMN  5
int main(void)
{
  static int salary_tbl[NUM_ROW][NUM_COLUMN] =
  {
    {  500,  550,  600,  650,  700   },
    {  600,  670,  740,  810,  880   },
    {  740,  840,  940, 1040, 1140   }
  };
  int grade , step;
  for (grade = 0; grade < NUM_ROW; ++grade)
   for (step = 0; step < NUM_COLUMN; ++step)
   {
     printf("salary_tbl[%d] [%d] = %d\n", grade, step,
                                     salary_tbl[grade] [step]);
   }
}
```

The preceding program produces the following output:

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

**Related Information**

- “Pointers”
- “Array Element Specification (Array Subscript) [ ]” on page 85
- “Strings” on page 20
- “Declarators” on page 33
- “Initializers” on page 37
- “Chapter 6. Conversions” on page 105

## Pointers

A **pointer** type variable holds the address of a data object or a function. [1] A pointer can refer to an object of any one data type but cannot point to an object having the `register` storage class specifier or to a bit field. Some common uses for pointers are:

- To pass the address of a variable to a function. By refering to a variable through its address, a function can change the contents of that variable. See “Calling Functions and Passing Arguments” on page 76.
- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure.
- To access an array of characters as a string.

## Declarators of Pointers

The following example declares `pcoat` as a pointer to an object that has type `long`:

```
extern long *pcoat;
```

If the keyword `volatile` appears before the `*`, the declarator describes a pointer to a `volatile` object. If the keyword `volatile` comes between the `*` and the identifier, the declarator describes a `volatile` pointer.

The keyword `const` operates in the same manner as the `volatile` keyword that is described in the preceding paragraph. In the following example, `pvolt` is a constant pointer to an object that has type `short`:

```
short * const pvolt;
```

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

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

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

```
float * volatile psoup;
```

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

```
enum bird *pfowl;
```

1. AS/400 pointers can hold addresses of more than just data objects or functions. See the *ILE C for AS/400 Programmer’s Guide* for more information about using AS/400 pointers.

The next example declares x as a pointer to a function that returns a char object:

```
char (*x)(void);
```

## Initializers of Pointers

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

The following example shows compatible declarations for the assignment operation:

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

The next example shows incompatible declarations for the assignment operation:

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

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

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

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

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

This definition is equivalent to:

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

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer. The following example defines the pointer variable string and the string constant "abcd". The pointer string is set to point to the character a in the string "abcd".

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

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

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

A pointer can also be initialized to the integer constant `0`. Such a pointer is a **NULL pointer** that does not point to any object.

## Restrictions

You cannot use pointers to refer to bit fields or objects that have the `register` storage class specifier.

A pointer to a packed structure or union is incompatible with a pointer to a corresponding nonpacked structure or union. This is because packed and nonpacked objects have different memory layouts. As a result, comparisons and assignments between pointers to packed and nonpacked objects are not valid.

You can, however, perform these assignments and comparisons with type casts. Consider the following example:

```
int main(void)
{
   _Packed struct ss *ps1;
   struct ss         *ps2;
      .
      .
      .
   ps1 = (_Packed struct ss *)ps2;
      .
      .
      .
}
```

In the preceding example, the cast operation allows you to compare the two pointers. However, you must be aware that `ps1` still points to a nonpacked object.

## Using Pointers

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

```
int x, *p_to_x;

p_to_x = &x;
```

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

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

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

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

## Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. [2]These operations are:

- Increment and decrement
- Addition and subtraction
- Comparison
- Assignment.

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

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

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

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

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

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`. See "Chapter 5. Expressions and Operators" on page 79 for more information on these operators.

Pointer comparisons are defined only when the pointers point to elements of the same array. You can assign to a pointer the address of a data object, the value of another compatible pointer, or the `NULL` pointer.

Consider the following example of pointer arithmetic:

```
ptr = ptr + i - j;
```

```
ptr + i
```

must fall within the bounds of allocated memory. If the value of `i` causes the intermediate value `ptr` to be outside the valid range, parenthesis must be added as follows:

```
ptr = ptr + (i - j);
```

### Examples

The following program contains pointer arrays:

```
/********************************************************************
**   Program to search for the first occurrence of a specified     **
**   character string in an array of character strings.           **
********************************************************************/
#include <stdio.h>
```

2. There are restrictions on pointer arithmetic for AS/400 pointers. See the *ILE C for AS/400 Programmer's Guide* for more information about using AS/400 pointers.

```
#include <stdlib.h>
#include <string.h>
#define  SIZE  20
int main(void)
{
   static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
   char * find_name(char **, char *);
   char new_name[SIZE], *name_pointer;
   printf("Enter name to be searched.\n");
   scanf("%s", new_name);
   name_pointer = find_name(names, new_name);
   printf("name %s%sfound\n", new_name,
          (name_pointer == NULL) ? " not " : " ");
   exit(EXIT_FAILURE);
} /* End of main */
/*******************************************************************
**     Function find_name.  This function searches an array of     **
**     names to see if a given name already exists in the array.   **
**     It returns a pointer to the name or NULL if the name is     **
**     not found.                                                  **
*******************************************************************/
/* char **arry is a pointer to arrays of pointers (existing names) */
/* char *strng is a pointer to character array entered (new name)  */
char * find_name(char **arry, char *strng)
{
   for (; *arry != NULL; arry++)         /* for each name          */
   {
      if (strcmp(*arry, strng) == 0)     /* if strings match       */
         return(*arry);                  /* found it!              */
   }
   return(*arry);                        /* return the pointer     */
} /* End of find_name */
```

Interaction with the preceding program could produce the following sessions:

| | |
|---|---|
| **Output** | `Enter name to be searched.` |
| **Input** | `Mark` |
| **Output** | `name Mark found` |

or:

| | |
|---|---|
| **Output** | `Enter name to be searched.` |
| **Input** | `Bob` |
| **Output** | `name Bob not found` |

**Related Information**

- “Address &” on page 88
- “Indirection *” on page 88
- “Declarators” on page 33
- “volatile and const Qualifiers” on page 35
- “Initializers” on page 37

## Structures

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

You can use structures to group logically related objects. For example, if you want to allocate storage for the components of one address, you can define the following variables:

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

To allocate storage for more than one address, you can group the components of each address. This can be done by defining a structure data type and defining several variables having the structure data type:

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

Lines 1 through 7 declare the structure tag `address`. Line 8 defines the variable `perm_address`, and line 9 defines the variable `temp_address`. Both of which are instances of the structure `address`. Both `perm_address` and `temp_address` contain the members described in lines 2 through 6. Line 10 defines a pointer `p_perm_address`, which points to a structure of `address`. `p_perm_address` is set to point to `perm_address`.

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

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

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

All references to structures must be fully qualified. Therefore, in the preceding example, you cannot refer to the fourth field by `prov` alone. You must refer to this field by `perm_address.prov`.

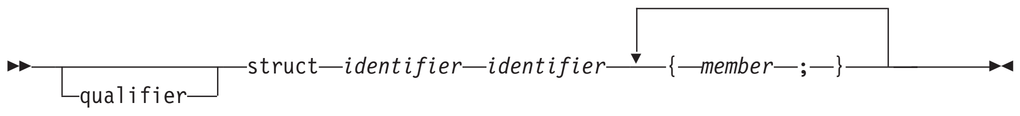
You cannot declare a structure with members of incomplete types. See “Incomplete Types” on page 67 for more information.

## Declaring a Structure Data Type

A structure type declaration does not allocate storage. It describes the members that are part of the structure.

A structure type declaration contains the `struct` keyword that is followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

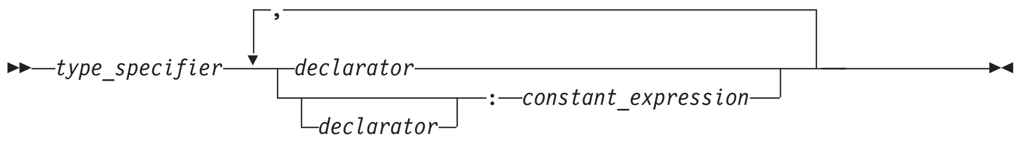
A structure declaration has the form:



The keyword `struct` followed by the identifier (tag) names the data type. If you do not provide a tag, you must place all variable definitions that refer to that data type within the statement that defines the data type.

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

A member has the form:



A member that does not represent a bit field can be of any data type and can have the `volatile` or `const` qualifier. If a **:** (colon) and a constant expression follow the declarator, the member represents a **bit field**.

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

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

## Defining a Variable That Has a Structure Data Type

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

## Storage Classes of Structures

The ILE C compiler ignores register storage class for structures, unions, and their members. Other C compilers may treat unions that are declared with the register storage class differently.

## Initializers of Structures

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

The following definition shows a completely initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
static struct address perm_address =
               { 9876, "Goto St.", "Cville", "Ontario", "X9X 1A1"};
```

The values of `perm_address` are:

| Member | Value |
|---|---|
| perm_address.street_no | 9876 |
| perm_address.street_name | address of string "`Goto St.`" |
| perm_address.city | address of string "`Cville`" |
| perm_address.prov | address of string "`Ontario`" |
| perm_address.postal_code | address of string "`X9X 1A1`" |

The following definition shows a partially initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
struct address temp_address =
               { 321, "Aggregate Ave.", "Structown", "Ontario" };
```

The values of `temp_address` are:

| Member | Value |
|---|---|
| temp_address.street_no | 321 |
| temp_address.street_name | address of string "Aggregate Ave.″ |
| temp_address.city | address of string "`Structown`" |
| temp_address.prov | address of string "`Ontario`" |
| temp_address.postal_code | value depends on the storage class. |

**Note:** The initial value of `temp_address.postal_code` depends on the storage class associated with the member. See “Storage Class Specifiers” on page 25 for details on the initialization of different storage classes.

## Example of Declaring a Structure Type and Structure Variables

You can place a type definition and a variable declaration in one statement by placing a declarator and an initializer (optional) after the type definition. If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the statement. For example:

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

The preceding example does not name the structure data type. Thus, `perm_address` and `temp_address` are the only structure variables that will have this data type. If an identifier is placed after `struct`, additional variable definitions of this data type can be made later in the program.

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

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

This example gives the `volatile` qualifier to the structures `file1` and `file2`, and to the structure member `subfile.code`.

## Declaring and Using Bit Fields

A structure can contain **bit fields** that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier that is followed by an optional declarator, a colon, a constant expression, and a semicolon.

The constant expression specifies how many bits the field reserves. A bit field that is declared as having a length of `0` causes the next field to be aligned on the next integer boundary. For a `_Packed` structure, a bit field of length `0` causes the next field to be aligned on the next byte boundary. Bit fields with a length of `0` must be unnamed.

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

You cannot define an array of bit fields, or take the address of a bit field.

You can declare a bit field as type `int`, `signed int`, or `unsigned int`. Bit fields of the type `int` are equivalent to those of type `unsigned int`.

If a series of bit fields does not add up to the size of an `int`, padding can take place. The amount of padding is determined by the alignment characteristics of the structure's members. In some instances, bit fields can cross word boundaries.

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

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

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

| Member Name | Storage Occupied |
|---|---|
| **light** | 1 bit |
| **toaster** | 1 bit, and padding to next `int` boundary |
| **count** | The size of an `int` |
| **ac** | 4 bits |
| | 4 bits (unnamed field) |
| **clock** | 1 bit and padding to next `int` boundary (unnamed field with size 0) |
| **flag** | 1 bit |

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

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

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

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

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

## Declaring a Packed Structure

Data elements of a structure are stored in memory on an address boundary specific for that data type. For example, a `double` value is stored in memory on a double-word (8-byte) boundary. Gaps may be left in memory between elements of a structure to align elements on their natural boundaries. You can reduce the padding of bytes within a structure by using the `_Packed` qualifier on the structure declaration.

### Examples

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

```
/* program to illustrate linked lists */
#include <stdio.h>
struct record {
                int number;
                struct record *next_num;
              };
int main(void)
{
   struct  record name1, name2, name3;
   struct  record *recd_pointer = &name1;
   int sum = 0;
   name1.number = 144;
   name2.number = 203;
   name3.number = 488;
   name1.next_num = &name2;
   name2.next_num = &name3;
   name3.next_num = NULL;
   while (recd_pointer != NULL)
   {
      sum += recd_pointer->number;
      recd_pointer = recd_pointer->next_num;
   }
   printf("Sum = %d\n", sum);
}
```

The structure type `record` contains two members: `number` (an integer) and `next_num` (a pointer to a structure variable of type `record`).

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

| Member Name | Value |
|---|---|
| name1.number | 144 |
| name1.next_num | The address of `name2` |
| name2.number | 203 |
| name2.next_num | The address of `name3` |
| name3.number | 488 |
| name3.next_num | `NULL` (Indicating the end of the linked list.) |

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

The `while` loop causes the linked list to be scanned until `recd_pointer` equals `NULL`. The following statement advances the pointer to the next object in the list:

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

**Related Information**

- “Structure and Union Member Specification . −>” on page 85
- “Declarators” on page 33
- “Initializers” on page 37

## Unions

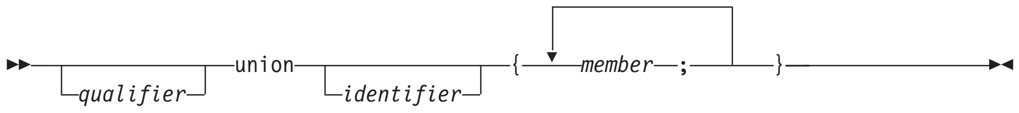
A **union** is an object that can hold any one of a set of named members. The members of the named set can be of any data type. Members are overlaid in storage.

## Declaring a Union

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

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

The following diagram shows the form of a union type declaration:

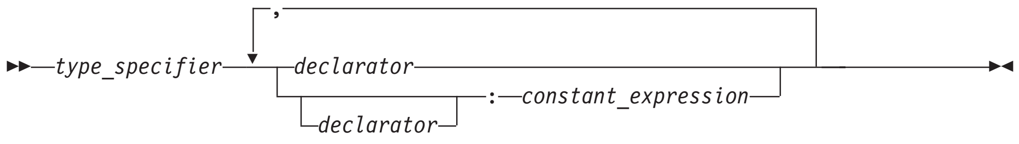


The ILE C compiler ignores register storage class for structures, unions, and their members. Other C compilers may treat unions that are declared with the register storage class differently.

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

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

A member has the form:



You can refer to one of the union’s possible members as you refer to a member of a structure. For example:

```
union {
      char birthday[9];
      int sex:1;  /*  0 = male; 1 = female  */
      float weight;
      } people;
people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`. At any given time, a union can represent only one of its members. In the preceding example, the union `people` will contain either `sex`, `birthday`, or `weight` but never more than one of these.

The following code fragment shows some common errors when using unions. The first line stores a date into the union through the people.birthday member. However, the assignment to people.sex on line 2 will change the leftmost bit of the union to 1. It will probably change the date stored in people.birthday. On line 3, since a date (an array of char) is stored in the union, do not display the value as a float by using people.weight.

```
1  people.birthday = "25/10/67";
2  people.sex = 1;
3  printf("%f\n", people.weight);
```

## Example of Defining a Variable that Has Union Data Type

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

**Type Specifier**

The type specifier contains the keyword `union` that is followed by the name of the union type. You must declare the union data type before you can define a union that has that type.

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

**Declarator**

The declarator is an identifier, possibly with the `volatile` or `const` qualifier.

**Initializer**

You can only initialize the first member of a union.

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

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = {"04/06/57"};
```

## Defining a Union Type and a Union Variable

You can place a type definition and a variable definition in one statement by placing a declarator after the type definition. If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the statement.

## Defining a Packed Union

You can use `_Packed` to qualify a union. However, the memory layout of the union members is not affected. Each member starts at offset zero. The `_Packed` qualifier does affect the total alignment restriction of the whole union. Consider the following example:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};
union uu           n_array[2];
_Packed union uu   p_array[2];
```

Each of the elements in the nonpacked `n_array` is of type `union uu`. Because it is nonpacked, each element has an alignment restriction of 2 bytes. The largest alignment requirement among the union members is that of `short a`. There is 1 byte of padding at the end of each element to enforce this requirement.

Now consider the packed array p_array. The alignment restriction of every element is the byte boundary, because each of its elements is of type _Packed union uu. Therefore, each element has a length of only 3 bytes, instead of the 4 bytes in the previous example.

### Examples

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

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

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

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

The following statement assigns a character to input:

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

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

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

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

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

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

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

### Related Information

- "Structure and Union Member Specification . −>" on page 85
- "Declarators" on page 33
- "Initializers" on page 37

## typedef

C language `typedef` declarations allow you to define your own identifiers that can be used in place of C type specifiers such as `int`, `float`, and `double`. The data types you define using `typedef` are not new data types. The identifiers defined are synonyms for the primary data types that are used by the C language or data types that are derived by combining the primary data types.

The syntax of a `typedef` declaration is:

```
►►──typedef──type_specifier──identifier──;──────────────────────────────────►◄
```

A `typedef` declaration does not reserve storage.

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

**Examples**

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

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

The preceding lines are equivalent to the following:

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

Similarly, `typedef` defines a `structure` type. For example:

```
typedef struct {
                int kilos;
                int grams;
               } WEIGHT;
```

The following declarations can use the structure `WEIGHT`:

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

**Related Information**

- “Characters” on page 38
- “Floating-Point Variables” on page 39
- “Integers” on page 40
- “void Type” on page 43
- “Arrays” on page 47
- “Enumerations” on page 44
- “Pointers” on page 52
- “Structures” on page 57
- “Unions” on page 62

## Incomplete Types

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

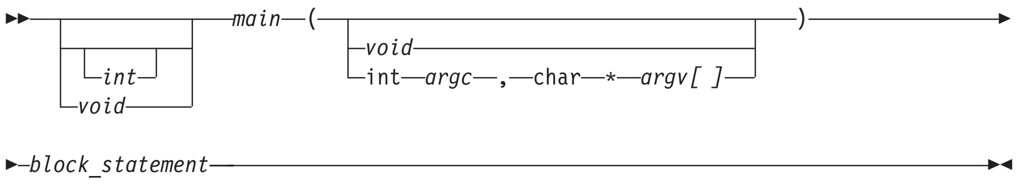
```
void var;
struct struct_type st;  /*  no previous definition of struct_type  */
```

The `void` type is incomplete and cannot be completed. Structure, union, or enumeration tags that are incomplete when first declared must be completed before their first use in a statement.

# Chapter 4. Functions

## main

When you begin the processing of a program, the system automatically calls the function `main`. Every program must have one function named `main`, with the name `main` that is written in lowercase letters. A `main` function has the form:



The function `main()` can declare either none or two parameters. Although any name can be given to these parameters, they are usually referred to as *argc* and *argv*. The first parameter, *argc* (argument count), has type `int` and indicates how many arguments were entered on the command line. The second parameter, *argv* (argument vector), has type array of pointers to `char` array objects. `char` array objects are null-ended strings.

The value of *argc* indicates the number of pointers in the array *argv*. If a program name is available, the first element in *argv* points to a character array that contains the program name or the invocation name of the program that is being executed. If the name cannot be determined, the first element in *argv* points to a null character. This name is counted as one of the arguments to the function `main()`. For example, if only the program name is entered on the command line, *argc* has a value of `1`, and *argv[0]* points to the program name.

Regardless of the number of arguments that are entered on the command line, *argv[argc]* always contains `NULL`.

The following program `backward` prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
}
```

If you start this program from a command line with the following:

```
CALL PGM(MYLIB/BACKWARD) PARM('string1' 'string2')
```

the output that is generated is:

```
string2 string1
```

The arguments *argc* and *argv* would contain the following values:

| **Object** | **Value** |
|---|---|
| `argc` | `3` |
| `argv[0]` | pointer to string `"MYLIB/BACKWARD"` |

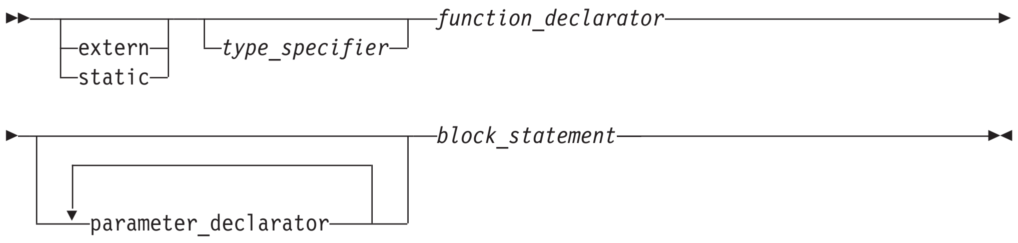
| Object | Value |
|---|---|
| argv[1] | pointer to string "string1" |
| argv[2] | pointer to string "string2" |
| argv[3] | NULL |

**Related Information**

- “Calling Functions and Passing Arguments” on page 76
- “Parameter Declaration List” on page 73
- “Types” on page 38
- “Identifiers” on page 12
- “Block” on page 120

## Function Definition

A **function definition** specifies the name, formal parameters, and body of a function. You can also specify the function’s return type and storage class. A function definition has the form:



There are two ways to define a function: prototype and nonprototype. You should use the prototype method because of the parameter type checking that can be performed.

A function definition (either prototype or nonprototype) contains the following:

- The optional storage class specifier `extern` or `static`, which determines the scope of the function. If a storage class specifier is not given, the function has external connection.
- An optional type specifier, which determines the type of value that the function returns. If a type specifier is not given, the function has type `int`.
- A function declarator, which provides the function with a name, can further describe the type of the value that the function returns. It can list any parameters (and their types) that the function expects. Parentheses enclose the parameters that the function is expecting.
- A block statement, which contains data definitions and code.

In addition, the non-prototype function definition may also have parameter declarations, which describe the types of parameters that the function receives. In non-prototype functions, parameters that are not declared have type `int`.

A function can be called by itself or by other functions. Unless a function definition has the storage class specifier `static`, the function also can be called by functions that appear in other modules. If the function has a storage class specifier of static, it can only be directly started from within the same source file. If a function has the storage class specifier static or a return type other than int, the function definition or a declaration for the function must appear before, and in the same file as, a call to the function.

If a function definition has external linkage and a return type of int, calls to the function can be made before it is visible. An implicit declaration of extern int func(); is assumed. All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

The default type for the return value and parameters of a function is int, and the default storage class specifier is extern. If the function does not return a value or it is not passed any parameters, use the keyword void as the type specifier.

You can include ellipses (...) at the end of your parameter list to indicate that a variable number of arguments will be passed to the function. When a function with a variable number of arguments is called, argument type promotion is performed, and no type checking is done on the arguments.

You cannot declare a function as a struct, or union member.

A function cannot have a return type of function, array, or any type having the volatile or const qualifier. However, it can return a pointer to an object with a volatile or const type.

You cannot define an array of functions. You can, however, define an array of pointers to functions. For example, in the following, ary is an array of two function pointers. Type casting is performed to the values that are assigned to ary for compatibility:

```
#include <stdio.h>

int func1(void);
void func2(double a);

int main(void)
{
   double num;
   void (*ary[2]) ();
   ary[0] = ((void(*)())func1);
   ary[1] = ((void(*)())func2);

   ((int (*)())ary[0])();            /*  calls func1  */
   ((void (*)(double))ary[1])(num);  /*  calls func2  */
}
```

The following example is a complete definition of the function sum:

```
int sum(int x,int y)
{
   return(x + y);
}
```

The function sum() has external connection, returns an object that has type int, and has two parameters of type int declared as x and y. The function body contains a single statement that returns the sum of x and y.

To use the function definition for library functions, you must undefine the macro with the #undef directive before calling the function. The function definition for library functions are provided as both macros and functions. A macro definition can be suppressed one of the following ways:

- Enclose the name of the function in parentheses. For example, `(atan)(x)` uses the `atan()` function to calculate the arc tangent of `x`.
- Take the address of the library function.
- Use the #undef directive to remove the macro definition of the function. For example, `#undef atan`.

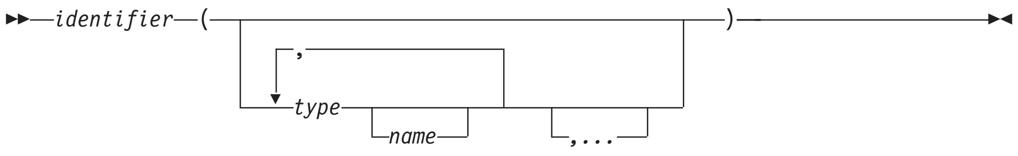
For library functions that are provided as both macros and functions, the macro is defined in the header file that declares the function.

## Function Declarator

The **function declarator** names the function and lists the function parameters. A function declarator contains an identifier that names the function and a list of the function parameters. There are two types of function declarators: **prototype** and **nonprototype**.

## Prototype Function Declarator

A **prototype function declarator** has the form:



Each parameter should be declared within the function declarator for prototype function declarations. Any calls to the function must pass the same number of arguments as there are parameters in the declaration.

To indicate that a function does not receive any values, use the keyword `void` in place of the parameter. For example:

```
int stop(void)
{
}
```

The example below contains a function declarator `sort` with `table` that is declared as a pointer to `int` and `length` declared as type `int`. Note that arrays as parameters are implicitly converted to a pointer to the type.

```
inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1void sort(int table[ ], int length)
{
  int i, j, temp;

  for (i = 0; i < length -1; i++)
    for (j = i + 1; j < length; j++)
      if (table[i] > table[j])
      {
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
}
```

The following examples contain prototype function declarators:

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

The example below illustrates how a function declarator uses a `typedef` identifier:

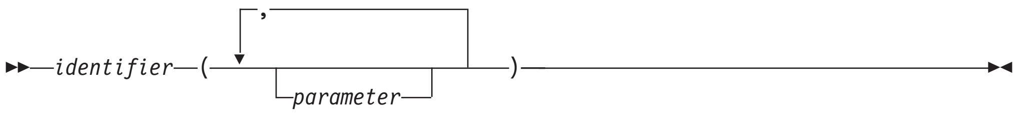
```
typedef struct tm_fmt { int minutes;
                      int hours;
                      char am_pm;
                    } struct_t;
long time_seconds(struct_t arrival)
```

The following function `set_date()` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class specifier `register`.

```
set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

## Nonprototype Function Declarator

A **non-prototype function declarator** has the form:



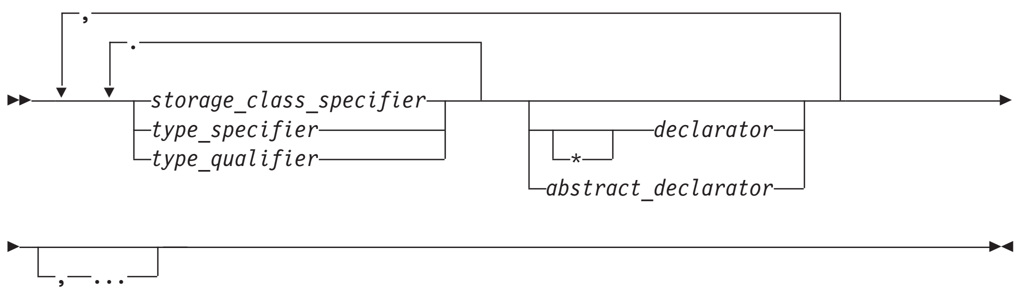
Each parameter should be declared in a parameter declaration list following the declarator. If a parameter is not declared, it has type `int`.

The `char` and `short` parameters are widened to `int`, and `float` to `double`. No type checking between the argument type and the parameter type is done for non-prototyped functions. As well, there are no checks to ensure that the number of arguments matches the number of parameters.

## Parameter Declaration List

Each value that a function receives should be declared in a parameter declaration list for non-prototype function definitions that follows the declarator.

A parameter declaration determines the storage class specifier and the data type of the value. It has the form:



The only storage class specifier that is allowed is the `register` storage class specifier. Any type specifier for a parameter is allowed. If you do not specify the `register` storage class specifier, the parameter will have the `auto` storage class specifier. If you omit the type specifier and you are not using the prototype form to define the function, the parameter will have type `int`.

```
    int func(i,j)
    {
       /*  i and j have type int  */
    }
```

You cannot declare a parameter in the parameter declaration list if it is not listed within the declarator.

## Function Body

The body of a function is a block statement. (For more information on block statements, see “Block” on page 120.) The following function has an empty body:

```
void stub1(void)
{
}
```

The following function body contains a definition for the integer variable `big_num`, an `if`-`else` control statement, and a call to the function `printf()`:

```
void largest(int num1, int num2)
{
   int big_num;

   if (num1 >= num2)
      big_num = num1;
   else
      big_num = num2;

   printf("big_num = %d\n", big_num);
}
```

## Function Declarations

A function declaration establishes the name and the parameters of the function. A function is declared implicitly by its appearance in an expression if it has not been defined or declared previously. The implicit declaration is equivalent to a declaration of `extern int` *func_name*`()`.

If the called function returns a value that has a type other than `int`, you must declare the function before the function call. Even if a called function returns a type `int`, explicitly declaring the function prior to its call is good programming practice.

Some declarations do not have parameter lists; the declarations simply specify the types of parameters and the return values, such as in the following example:

```
   int func(int,long);
```

**Examples**

The following example defines the function `absolute()` with the return type `double`. Because this is a noninteger return type, `absolute()` is declared prior to the function call.

```
#include <stdio.h>

double absolute(double);

int main(void)
{
   double f = -3.0;

   printf("absolute number = %lf\n", absolute(f));
}

double absolute(double number)
{
   if (number < 0.0)
      number = -number;

   return (number);
}
```

Specifying a return type of `void` on a function declaration indicates that the function does not return a value. The following example defines the function `absolute()` with the return type `void`. Within the function `main()`, `absolute()` is declared with the return type `void`.

```
#include <stdio.h>

int main(void)
{
  void absolute(float);
  float f = -8.7;

  absolute(f);
}

void absolute(float number)
{
  if (number < 0.0)
    number = -number;

  printf("absolute number = %f\n", number);
}
```

**Related Information**

- “extern Storage Class Specifier” on page 28
- “Declaration” on page 28.

## Calling Functions and Passing Arguments

A function call specifies a function name and a list of arguments. The calling function passes the value of each argument to the specified function. Parentheses surround the argument list, and a comma separates each argument. The argument list can be empty.

The arguments to a function are evaluated before the function call. When an argument is passed in a function call, the function receives a copy of the argument value. If the value of the argument is an address, the called function can use indirection to change the contents that are pointed to by the address. If a function or array is passed as an argument, the argument is converted to a pointer that points to the function or array.

Arguments that are passed to parameters in prototype declarations will be converted to the declared parameter type. For non-prototype function declarations, `char`, and `short` parameters are promoted to `int`, and `float` to `double`.

You can pass a packed structure argument to a function expecting a non-packed structure of the same type and vice versa. (The same applies to packed and non-packed unions.)

**Note:** If you do not use a function prototype and you send a packed structure when a non-packed structure is expected, a run-time error may occur.

In ILE C, arguments are evaluated and passed to the function from right to left. This may differ for other C compilers. For example, the following sequence of statements calls the function `tester()`:

```
int x;
x = 1;
tester(x++, x);
```

The call to the `tester()` function in the preceding example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first, or `x` may be evaluated first. To avoid ambiguity, if you want `x++` to be evaluated first, you can replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

**Examples**

The following statement calls the function `startup()` and passes no parameters:

```
startup();
```

The following function call causes copies of `a` and `b` to be stored in a local area for the function `sum()`. The function `sum()` is started using the copies of `a` and `b`.

```
sum(a, b);
```

The following function call passes the value `2` and the value of the expression `a + b` to the `sum()`function:

```
sum(2, a + b);
```

The following statement calls the functions `printf()` and `sum()`. The `sum()` receives the values of `a` and `b`. The `printf()` function receives a character string and the return value of the function `sum()`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of `count` to the function `increment()`. The `increment()` function increases the value of the parameter `x` by `1`.

```
#include <stdio.h>

void increment(int);

int main(void)
{
  int count = 5;

  /* value of count is passed to the function */
  increment(count);
  printf("count = %d\n", count);
}

void increment(int x)
{
  ++x;
  printf("x = %d\n", x);
}
```

The output illustrates that the value of `count` in the `main()` function remains unchanged:

```
x = 6
count = 5
```

In the following program, the `main()` function passes the address of `count` to the `increment()` function. The function `increment()` was changed to handle the pointer. The parameter `x` is declared as a pointer. The contents to which `x` points are then incremented.

```
#include <stdio.h>

int main(void)
{
  void increment(int *x);
  int count = 5;

  /* address of count is passed to the function */
  increment(&count);
  printf("count = %d\n", count);
}

void increment(int *x)
{
  ++*x;
  printf("*x = %d\n", *x);
}
```

The output shows that the variable `count` is increased:

```
*x = 6
count = 6
```

See "Using Packed Decimal Data in Your ILE C Programs" in the *ILE C for AS/400 Programmer's Guide* for more information about using packed decimal data types in function calls.

# Chapter 5. Expressions and Operators

This chapter describes C language expressions. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used:

Most expressions can contain several different, but related, types of operands. The following **type classes** describe related types of operands:

| | |
|---|---|
| **Integral** | Character objects and constants, objects that have an enumeration type, and objects that have the type `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `long long`, or `unsigned long long`. |
| **Arithmetic** | Integral objects and objects having the type `float`, `double`, packed decimal, and `long double`. |
| **Scalar** | Arithmetic objects and pointers to objects of any type. |
| **Aggregate** | Arrays, structures, and unions. |

Many operators cause conversions from one data type to another. See "Chapter 6. Conversions" on page 105. Note that logical operations always return 32-bit Boolean values.

## Grouping and Evaluating Expressions

Two operator characteristics determine how operands group with operators: **precedence** and **associativity**. Precedence provides a priority system for grouping different types of operators with their operands. Associativity provides a left-to-right or right-to-left order for grouping operands to operators that have the same precedence. For example, in the following statements, the value of `5` is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

You can explicitly force the grouping of operands with operators by using parentheses because the order of expression evaluation is not specified.

In the expression `a + b * c / d`, the `*`, and `/` operations are performed before `+` because of precedence. `b` is multiplied by `c` before it is divided by `d` because of associativity.

The following table lists the C language operators in order of precedence and shows the direction of associativity for each operator. The primary operators have the highest precedence. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

Table 7. Operator Precedence and Associativity

| Operator Name | Associativity | Operators |
|---|---|---|
| Primary | left to right | `() [ ] . ->` |
| Unary | right to left | `++ -- - + ! & *` (*typename*) `sizeof digitsof, precisionof` |
| Multiplicative | left to right | `* / %` |
| Additive | left to right | `+ -` |
| Bitwise Shift | left to right | `<< >>` |
| Relational | left to right | `< > <= >=` |
| Equality | left to right | `== !=` |
| Bitwise Logical AND | left to right | `&` |
| Bitwise Exclusive OR | left to right | `ˆ` or `¬` |
| Bitwise Inclusive OR | left to right | `¦` |
| Logical AND | left to right | `&&` |
| Logical OR | left to right | `¦¦` |
| Conditional | right to left | `? :` |
| Assignment | right to left | `= += -= *= /= <<= >>= %= &= ˆ= ¦=` |
| Comma | left to right | `,` |

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing such ambiguous expressions as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, `++y` and `func1(y)` may not be evaluated in the same order by all C language implementations. If `y` had the value of 1 before the first statement, it is not known whether or not the value of 1 or 2 is passed to `func1()`. In the second statement, if `i` had the value of 1, it is not known whether the first or second array element of `x[ ]` is passed as the second argument to `func2()`.

The order of grouping operands with operators in an expression, that contain more than one instance of an operator, with both associative and commutative properties, is not specified. The operators that have the same associative and commutative properties are: `*`, `+`, `&`, `¦`, and `ˆ` (or `¬`). Grouping the expression in parentheses can force the grouping of operands.

**Examples**

The parentheses in the following expressions explicitly show how the C language groups operands and operators. If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses.

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

The following expression contains operators that are both associative and commutative:

```
total = price + prov_tax + city_tax;
```

Because the C language does not specify the order of grouping operands with operators that are both associative and commutative, the operands and operators could be grouped in the following ways (as indicated by parentheses):

```
total = (price + (prov_tax + city_tax));
total = ((price + prov_tax) + city_tax);
total = ((price + city_tax) + prov_tax);
```

However, the grouping of operands and operators could affect the result. In the following expression, each function call may change the same global variables. These side effects may result in different values for the expression that depend on the order in which the functions are called:

```
a = b() + c() + d();
```

If the expression contains operators that are both associative and commutative, and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable `a`.

```
a = b();
a += c();
a += d();
```

### Related Information

- “Parenthesized Expression ( )” on page 83

## Lvalue

An **lvalue** is an expression that represents an object. A **modifiable lvalue** is an expression representing an object that can be changed. A modifiable lvalue is the left operand in an assignment expression. However, arrays and `const` objects are not modifiable lvalues.

### Usage

All assignment operators evaluate their right operand and assign that value to their left operand. The left operand must evaluate to a reference to an object.

The assignment operators are not the only operators that require an operand to be an lvalue. The address operator requires an lvalue as an operand while the increment and the decrement operators require a modifiable lvalue as an operand.

### Examples

| Expression | Lvalue |
|---|---|
| `x = 42;` | `x` |
| `*ptr = newvalue;` | `*ptr` |
| `a++` | `a` |

### Related Information

- “Assignment Expression” on page 102
- “Address &” on page 88
- “Structure and Union Member Specification . −>” on page 85

## Constant Expression

A **constant expression** is an expression with a value that is determined during compilation and cannot be changed during processing. It can only be evaluated. Constant expressions can be composed of integer constants, character constants, floating-point constants, and enumeration constants.

**Usage**

The ILE C language requires constant expressions in the following places:
- In the subscript declarator, as the description of an array bound
- After the keyword `case` in a `switch` statement
- In an enumerator, as the numeric value of an enum constant
- In a bit-field width specifier
- In the preprocessor `if` statement
- In the initializer of a file scope data definition
- In the size and precision attributes of a packed decimal variable

In all the cases above, except for an initializer of a file scope data definition, the constant expression can contain integer, character, and enumeration constants, casts to integral types, and `sizeof` expressions.

In a file scope data definition, the initializer must evaluate to a constant or to the address of a static storage ( `extern` or `static`) object (plus or minus an integer constant) that is defined or declared earlier in the file. Thus, the constant expression in the initializer can contain integer, character, enumeration, and float constants, casts to any type, `sizeof` expressions, and addresses (possibly modified by constants) of static objects.

**Examples**

The following examples show constants that are used in expressions.

| Expression | Constant |
|---|---|
| `x = 42;` | `42` |
| `extern int cost = 1000;` | `1000` |
| `y = 3 * 29;` | `3 * 29` |

**Related Information**
- “Arrays” on page 47
- “File Scope Data Declarations” on page 24
- “switch” on page 131
- “Enumerations” on page 44
- “Structures” on page 57

- "Conditional Compilation" on page 149

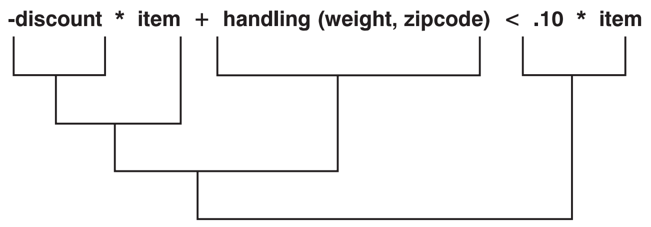
## Primary Expression

A **primary expression** can be:
- An identifier
- A string literal
- A parenthesized expression
- A constant expression
- A function call
- An array element specification
- A structure or union member specification.

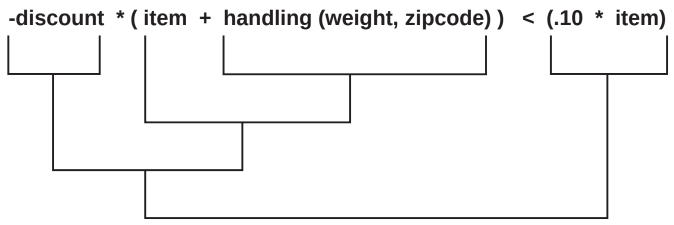
All primary operators have the same precedence and have left-to-right associativity.

## Parenthesized Expression ( )

You can use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses that are used for grouping operands and operators. The parentheses surrounding `weight, zipcode` are used to form a function call. Notice how the operands and operators are grouped in this expression according to the rules for operator precedence and associativity:



The following expression is similar to the previous expression. This expression, however, contains parentheses that change how the operands and operators group:


-discount * ( item + handling (weight, zipcode) ) < (.10 * item)

In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:

```
x = f + (g + h);
```

## Function Call ( )

A **function call** is a primary expression containing a parenthesized argument list. The argument list can contain any number of expressions that are separated by commas. For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

The arguments are evaluated, and each parameter is assigned the value of the corresponding argument. Assigning a value to a parameter within the function body changes the value of the parameter within the function. However, it has no effect on the argument.

In the following example, `main()` passes `func()` two values: `5` and `7`. The function `func()` receives copies of these values and accesses them by the identifiers: `a` and `b`. The function `func()` changes the value of `a`. When control passes back to `main()`, the actual values of `x` and `y` are not changed. The called function `func()` only receives copies of `x` and `y`, not the values themselves.

```
#include <stdio.h>
void func(int, int);
int main(void)
{
   int x = 5, y = 7;
   func(x, y);
   printf("In main, x = %d    y = %d\n", x, y);
}
void func (int a, int b)
{
   a += b;
   printf("In func, a = %d    b = %d\n", a, b);
}
```

This program produces the following output:

```
In func, a = 12   b = 7
In main, x = 5    y = 7
```

When you define a function as returning a certain type, for example type `int`, the result of the function call has that type.

If you want a function to change the value of a variable, pass a pointer to the variable you want changed. When a pointer is passed as a parameter, the pointer is copied; the object pointed to is not copied. (See “Pointers” on page 52.)

All arrays and functions are converted to pointers when passed as function arguments. Arguments that are passed to non-prototyped functions undergo conversions: type `short` or `char` parameters will be converted to `int`, and `float` parameters to `double`. Use a cast expression for other conversions. (See “Cast” on page 89.)

If the function has not been previously declared, an implicit declaration of `extern int func();` is assumed.

The compiler compares the data types that are provided by the calling function with the data types that the called function expects. The compiler may also perform type conversions if the declaration of the function is:

- in function prototype format and the parameters differ from the prototype
  OR
- visible at the point where the function is called.

The order in which parameters are evaluated is not specified. Avoid such calls as:

```
method(sample1, batch.process--, batch.process);
```

In the preceding example, `batch.process--` may be evaluated last, causing the last two arguments to be passed with the same value.

A function can call itself.

See "Chapter 4. Functions" on page 69 for detailed characteristics of functions.

## Array Element Specification (Array Subscript) [ ]

A primary expression followed by an expression in `[ ]` (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a subscript.

The primary expression must have a pointer type, and the subscript must have integral type. The result of an array subscript is an lvalue.

The first element of each array has the subscript `0`. Thus, the expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can refer to each element (in the order of increasing storage locations) by increasing the rightmost subscript most frequently. For example, the following statement gives the value `100` to each element in the array `code[4][3][6]`:

```
for (first = 0; first <= 3; ++first)
   for (second = 0; second <= 2; ++second)
      for (third = 0; third <= 5; ++third)
         code[first][second][third] = 100;
```

"Arrays" on page 47 explains how to define and use an array.

## Structure and Union Member Specification . −>

Two primary operators enable you to specify structure and union members: `.` (dot) and `->` (arrow).

The dot (a period) and arrow (formed by a minus and a greater than symbol) operators are always preceded by a primary expression. They are followed by an identifier.

When you use the dot operator, the primary expression must be an instance of a type of structure or union. The identifier must name a member of that structure or union. The result is the value that is associated with the named structure or union member. The result is an lvalue if the first expression is an lvalue.

Some sample dot expressions:

```
roster[num].name
roster[num].name[1]
```

When you use the arrow operator, the primary expression must be a pointer to a structure or a union. The identifier must name a member of the structure or union. The result is the value of the named structure or union member to which the pointer expression refers. In the following example, `name` is an `int`:

```
roster -> name
```

See also “Unions” on page 62 and “Structures” on page 57.

## Unary Expression

A **unary expression** contains one operand and a unary operator. All unary operators have the same precedence and have right-to-left associativity.

### Increment ++

The ++ (increment) operator adds `1` to the value of the scalar operand. Otherwise, if the operand is a pointer, it increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. Thus, the operand must be a modifiable lvalue.

You can place the ++ before or after the operand. If it appears before the operand, the operand is increased. Then the new value is used in the expression. If you place the ++ after the operand, the current value of the operand is used in the expression. Then the operand is increased. For example:

```
play = ++play1 + play2++;
```

is equivalent to the following three expressions:

```
play1 = play1 + 1;
play  = play1 + play2;
play2 = play2 + 1;
```

The type of the increment expression is the same type as that of the operand.

The usual arithmetic conversions on the operand are performed. See “Usual Arithmetic Conversions” on page 105.

### Decrement −−

The `--` (decrement) operator subtracts `1` from the value of the scalar operand. Otherwise, if the operand is a pointer, it decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. Thus, the operand must be a modifiable lvalue.

You can place the `--` before or after the operand. If it appears before the operand, the operand is decreased, and the decreased value is used in the expression. If the `--` appears after the operand, the current value of the operand is used in the expression and the operand is decreased.

For example:

```
play = --play1 + play2--;
```

is equivalent to the following three expressions:

```
play1 = play1 - 1;
play = play1 + play2;
play2 = play2 - 1;
```

The type of the decrement expression is the same type as that of the operand.

The usual arithmetic conversions are performed on the operand. See “Usual Arithmetic Conversions” on page 105.

## Unary Plus +

The `+` (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

The result of the unary plus expression has the same type as the operand after any integral promotions (for example, `char` to `int`).

**Note:** Any plus sign preceding a constant is not part of the constant.

The usual arithmetic conversions on the operand are performed. See “Usual Arithmetic Conversions” on page 105.

## Unary Minus −

The `-` (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value `100`, `-quality` has the value `-100`.

The result of the unary minus expression has the same type as the operand after any integral promotions (for example, `char` to `int`).

**Note:** Any minus sign preceding a constant is not part of the constant.

The usual arithmetic conversions on the operand are performed. See “Usual Arithmetic Conversions” on page 105.

## Logical Negation !

The `!` (logical negation) operator determines whether the operand evaluates to `0` (false) or nonzero (true). The expression yields the value `1` (true) if the operand evaluates to `0`, and yields the value `0` (false) if the operand evaluates to a nonzero value. The operand must have a scalar data type, but the result of the operation has always type `int` and is not an lvalue.

The following two expressions are equivalent:

```
!right;
right == 0;
```

The usual arithmetic conversions on the operand are performed. See “Usual Arithmetic Conversions” on page 105.

## Bitwise Negation

The (bitwise negation) operator supplies the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand, but is not an lvalue.

Suppose that `x` represents the decimal value `5`. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `x` supplies the following result (that is represented here as a 16-bit binary number):

```
1111111111111010
```

The 16-bit binary representation of `0` is:

```
1111111111111111
```

The usual arithmetic conversions on the operand are performed. See “Usual Arithmetic Conversions” on page 105.

## Address &

The `&` (address) operator supplies a pointer to its operand. The operand must be an lvalue or function designator. It cannot be a bit field, nor can it have the storage class specifier `register`. The result is a pointer to an object or function having the type of the operand. Thus, if the operand has type `int`, the result is a pointer to an object having type `int`. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

See also “Pointers” on page 52.

## Indirection *

The `*` (indirection) operator determines that the value referred to by the pointer-type operand. The operand cannot be a pointer to void, or one of the AS/400 pointer types label pointer, invocation pointer, suspend pointer, system pointer, or open pointer. The operation supplies an lvalue or a function designator if the operand points to a function. Arrays and functions are converted to pointers. The type of the operand determines the type of the result. Thus, if the operand is a pointer to an `int`, the result has type `int`.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`. The result of applying an indirection operator to such a pointer is not defined.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value `3`.

See also “Pointers” on page 52.

## Cast

A *cast* operator converts the type of the operand to a specified data type and performs the necessary conversions to the operand for the type. The cast operator is a parenthesized type specifier. This type and the operand must be scalar; the type may also be `void`. The result has the type of the specified data type but is not an lvalue.

The following expression contains a cast expression to convert an operand of type `int` to a value of type `double`:

```
int x;
printf("x=%lf\n", (double)x);
```

The function `printf()` receives the value of `x` as a `double`. The variable `x` remains unchanged by the cast.

The following example shows you how to explicitly cast a packed decimal type to a float type and to an integer type.
The expected output is:

```
#include <decimal.h>
main (void)
{
  decimal(10,2) op_1 = -1123.4d;
  decimal(12,5) op_2 = 12d;
  decimal(4) op_3 = 5d;
  printf("op_1 = %D(5, 1)\n", (decimal(5,1)) op_1);
  printf("op_2 = %f\n", (float) op_2);
  printf("op_3 = %d\n", (int) op_3);
}
```

```
op_1 = -1123.4
op_2 = 12.000000
op_3 = 5
```

“Chapter 6. Conversions” on page 105 describes how the C language performs conversions.

## Size of an Object

The `sizeof` operator yields the size in bytes of the operand. The `sizeof` operation cannot be performed on a bit field, a function, or an incomplete type such as `void`. The operand may be the parenthesized name of a type. The compiler must be able to evaluate the size at compile time. The expression is not evaluated; there are no side effects. For example, the value of `b` is 5 from initialization to the end of the program:

```
int main(void){
  int b = 5;
  sizeof(b++);
}
```

The size of a `char` object is the size of a byte. Given that the variable `x` has type `char`, the expression `sizeof(x)` always evaluates to `1`.

The result of a sizeof operation has type `size_t`. `size_t` is an unsigned integral type defined in the `<stddef.h>` header.

The compiler determines the size of an object on the basis of its definition. The `sizeof` operator does not perform any conversions. However, if the operand contains operators that perform conversions, the compiler takes these conversions into consideration. The following expression causes the usual arithmetic conversions to be performed. The result of the expression `x + 1` has type `int` (if `x` has type `char`, `short`, or `int` or any enumeration type) and is equivalent to `sizeof(int)`:

```
sizeof (x + 1)
```

When you perform the `sizeof` operation on an array, the result is the total number of bytes in the array. The compiler does not convert the array to a pointer before evaluating the expression.

You can use a `sizeof` expression wherever a constant or unsigned constant is required. One of the most common uses for the `sizeof` operator is to determine the size of objects that are being communicated to or from storage allocation, input, and output functions.

For portability of code, you should use the `sizeof` operator to determine the size that a data type represents. In this instance, the name of the data type must be placed in parentheses after the `sizeof` operator. For example:

```
sizeof(int)
```

When you use the sizeof operator with decimal(*n*,*p*), the result is the total number of bytes that are occupied by the packed decimal type.

## Digits of an Object

When you use the digitsof operator with a packed decimal type, the result is an integer constant. The `digitsof` operand can be a packed decimaldata type or a packed decimal constant expression. The `digitsof` operator returns the number of significant digits (*n*) in a packed decimal type.

The following example gives you the number of digits (*n*) in a packed decimaltype.

```
#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.  */
```

## Precision of an Object

When you use the `precisionof` operator with a packed decimal type, the result is an integer constant. The `precisionof` operand can be a packed decimaldata type or a packed decimal constant expression. The precisionof operator tells you the number of decimal digits (*p*) of the packed decimal type.

The following example gives you the number of decimal digits (*p*) of the packed decimal type:

```
#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.  */
```

## Binary Expression

A **binary expression** contains two operands that are separated by one operator.

Not all binary operators have the same precedence. Table 7 on page 80 shows the order of precedence among operators. All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. Therefore, to ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

For binary expressions involving packed decimal types, alignment to the decimal point is performed before the addition, subtraction, relational, equality and logical operands are done. If more than 31 digits are needed to align the two operands, the compilation will fail. If the two operands align within the 31-digit boundary without truncation, you should check the following tables for information about intermediate results.

To calculate the size of the packed decimal data type intermediate result when you do not expect the result to exceed 31 digits, use Table 8. To calculate the size of the result when you do expect the result to exceed 31 digits, use Table 9. Both tables summarize the intermediate expression results with the four basic arithmetic operators and conditional operators when applied to the packed decimal types. Both tables assume that *x* has type decimal (n sub 1, p sub 1), *y* has type decimal(n sub 2, p sub 2) and decimal(*n*,*p*) is the result type.

*Table 8. Intermediate Results of Packed Decimal Expressions (Without Overflow in n or p)*

| **Expression** | **(*n*, *p*)** |
|---|---|
| x * *y* | *n* = n sub 1 + n sub 2, *p* = p sub 1 + p sub 2 |
| x / *y* | *n* = DEC_DIG, *p* = DEC_DIG - ((n sub 1 - p sub 1) + p sub 2) |
| x + *y* | *p* = max(p sub 1, p sub 2) ,*n* = max(n sub 1- p sub 1, n sub 2- p sub 2) + *p* + 1 |
| x − *y* | same rule as addition |
| *z* ? x : *y* | *p* = max(p sub 1, p sub 2), *n* = max(n sub 1- p sub 1, n sub 2- p sub 2) + *p* |

Use Table 9 to calculate the size of the result when the integral part or fractional part or both parts of a packed decimal value are expected to exceed 31 digits. Overflow occurs when, in order to calculate the intermediate result, more than 31 digits is required for *n*, *p* or both *n* and *p*.

*Table 9. Intermediate Results with Packed Decimal Expressions (With Overflow in n or p)*

| **Expression** | **(*n*, *p*)** |
|---|---|
| x * *y* | *n* = min(n sub 1 + n sub 2, DEC_DIG), *p* = min(p sub 1+ p sub 2, DEC_DIG - min((n sub 1 - p sub 1) + (n sub 2 - p sub 2), DEC_DIG)) |
| x / *y* | *n* = DEC_DIG, *p* = max(DEC_DIG - ((n sub 1- p sub 1) + p sub 2), 0) |
| x + *y* | $i_r$ = min(max(n sub 1- p sub 1, n sub 2- p sub 2) + 1, DEC_DIG), *p* = min(max(p sub 1, p sub 2), DEC_DIG - $i_r$), *n* = $i_r$ + *p* |
| x − *y* | same rule as addition |
| *z* ? x : *y* | $i_r$ = max(n sub 1- p sub 1, n sub 2- p sub 2), *p*= min(max(p sub 1, p sub 2), DEC_DIG - $i_r$), *n* = $i_r$ + *p* |

There is no alignment to the decimal point before multiplication and division operands are applied.

## Multiplication *

The `*` (multiplication) operator supplies the product of its operands. The operands must have an arithmetic type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed. See “Usual Arithmetic Conversions” on page 105.

The compiler may rearrange the operands in an expression that contains more than one multiplication operator. This is because the multiplication operator has both associative and commutative properties. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

When both operands are packed decimal types, the resulting type is formed according to the rules in Table 8 on page 91 and Table 9 on page 91.

**Note:** No negative zero results from the operation. For example,

```
-1d * +0d ==>   +00d
 1d * -0d ==>   +00d
```

The following example shows you how to multiply packed decimal variables.

```
#include <decimal.h>            /* Packed Decimal Header File */
#include <stdio.h>
int main(void)
{
  decimal(10,2) op_1 = 12d;
  decimal(5,5) op_2 = -.12345d;
  decimal(15,7) res_mul;
  res_mul = op_2 * op_1;
  printf("res_mul =%D(*,*)\n",digitsof(res_mul),precisionof(res_mul),res_mul);

}
```

The expected output is:

```
res_mul =-1.4814000
```

See the *Run-Time Library Reference* for information about `printf()` and %D(*,*).

## Division /

The `/` (division) operator supplies the quotient of its operands. The operands must have an arithmetic type. The result is not an lvalue.

If both operands are positive integers and the operation produces a remainder, the remainder is ignored. Thus, the expression `7 / 4` supplies the value `1` (rather than `1.75` or `2`).

For the ILE C compiler, the result of `-7 / 4` is `-1` with a remainder of `-3`, assuming that both `-7` and `4` are signed. If `4` is unsigned, `-7` is converted to unsigned.

The result is undefined if the second operand evaluates to `0`.

The usual arithmetic conversions on the operands are performed. See “Usual Arithmetic Conversions” on page 105.

When both operands are packed decimal types, the resulting type is formed according to the rules in Table 8 on page 91 and Table 9 on page 91.

**Note:** No negative zero results from the operation. For example,

```
-0d /  1d  <  +0.000000000000000000000000000000d
```

The following example shows you how to divide packed decimal variables.
The expected output is:

```
#include <decimal.h>           /* Packed Decimal Header File */
#include <stdio.h>
int main(void)
{
  decimal(24,12) op_3 = 12.34d;
  decimal(20,5) op_4 = 11.01d;
  decimal(31,14) res_div;
  res_div = op_3 / op_4;
  printf("res_div =%D(*,*)\n",digitsof(res_div),precisionof(res_div),res_div);

}
```

```
res_div =1.2079927338782
```

See the *Run-Time Library Reference* for information about `printf()` and %D(*,*).

## Remainder %

The `%` (remainder) operator supplies the remainder from the division of the left operand by the right operand. For example, the expression `5 % 3` supplies `2`. The result is not an lvalue.

Both operands must have an integral type. If the right operand evaluates to `0`, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of `a` if `b` is not `0` and `a/b` is representable:

```
( a / b ) * b + a % b;
```

The usual arithmetic conversions on the operands are performed. See “Usual Arithmetic Conversions” on page 105.

## Addition +

The + (addition) operator supplies the sum of its operands. Both operands must have an arithmetic type. Otherwise, the first operand must be a pointer to an object type and the second operand must have an integral type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

**Note:** Function pointers and the AS/400 pointer types system pointer, call pointer, label pointer, and suspend pointer cannot be used as operands in an addition expression.

When both operands are packed decimal types, the resulting type is formed according to the rules in Table 8 on page 91 and Table 9 on page 91.

**Note:** No negative zero results from the operation. For example:

```
-0d + -0d  <  +00d
```

The following example shows you how to add packed decimal variables.
The expected output is:

```
#include <decimal.h>           /* Packed Decimal Header File */
#include <stdio.h>
int main(void)
{
  decimal(10,2) op_1 = 12d;
  decimal(5,5) op_2 = -.12345d;
  decimal(14,5) res_add;
  res_add = op_1 + op_2;
  printf("res_add =%D(*,*)\n",digitsof(res_add),precisionof(res_add),res_add);

}

res_add =11.87655
```

See the *Run-Time Library Reference* for information about `printf()` and %D(*,*).

When one of the operands is a pointer, the compiler converts the other operand to an address offset. (See “To and From Pointer Types” on page 116.) The result is a pointer of the same type as the pointer operand. For example, after the addition, `ptr` will point to the third element of the array:

```
   int array[5];
   int *ptr;
   ptr = array + 2;
```

## Subtraction −

The **-** (subtraction) operator supplies the difference of its operands. Both operands must have an arithmetic type. Otherwise, the left operand must have a pointer type and the right operand must have the same pointer type or an integral type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. (See “To and From Pointer Types” on page 116.) The result is a pointer of the same type as the pointer operand.

If both operands are pointers to the same type, the compiler converts the result to a 32-bit integral type. The result represents the number of objects separating the two addresses. Behavior is undefined if the pointers do not refer to objects in the same array. In pointer arithmetic, subtracting two addresses to determine the offset always results in a 32-bit signed integer.

**Notes on pointers:**

1. You cannot use function pointers and the AS/400 pointer types system pointer, invocation pointer, label pointer, and suspend pointer as operands in a subtraction expression.
2. In pointer arithmetic, subtracting two addresses to determine the offset will always result in a 32-bit signed integer.

When both operands are packed decimal types, the resulting type is formed according to the rules in Table 8 on page 91 and Table 9 on page 91

**Note:** No negative zero results from the operation. For example:

```
8d - 8d  <  +00d
```

The following example shows you how to subtract packed decimal variables. The expected output is:

```
#include <decimal.h>           /* Packed Decimal Header File */
#include <stdio.h>
int main(void)
{
  decimal(10,2) op_1 = 12d;
  decimal(24,12) op_3 = 12.34d;
  decimal(25,12) res_sub;
  res_sub = op_3 - op_1;
  printf("res_sub =%D(*,*)\n",digitsof(res_sub),precisionof(res_sub),res_sub);

}

res_sub =0.340000000000
```

See the *Run-Time Library Reference* for information about `printf()` and %D(*,*).

## Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue.

The << (bitwise left shift) operator indicates that the bits are to be shifted to the left. The >> (bitwise right shift) operator indicates that the bits are to be shifted to the right.

Each operand must have an integral type. The compiler performs integral promotions on the operands. Then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions).

If the right operand has a negative value or a value that is greater than or equal to the width in bits of the expression being shifted, the result is undefined.

If the right operand has the value `0`, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `l_op` has the value `4019`, the bit pattern (in 16-bit format) of `l_op` is:

```
0000111110110011
```

The expression `l_op << 3` supplies:

```
0111110110011000
```

If the left operand has an `unsigned` type, the `>>` operator fills vacated bits with zeros. Otherwise, the compiler will fill the vacated bits of a signed value with a copy of the value's sign bit. For example, if `l_op` has the value `-25`, the bit pattern (in 16-bit format) of `l_op` is:

```
1111111111100111
```

The expression `l_op >> 3` supplies:

```
1111111111111100
```

## Relational < > <= >=

The relational operators compare two operands and determine the validity of a relationship. If the relationship that is stated by the operator is true, the value of the result is `1`. Otherwise, the value of the result is `0`.

The following table describes the relational operators:

*Table 10. Relational Operators*

| **Operator** | **Usage** |
|---|---|
| < | Indicates whether the value of the left operand is less than the value of the right operand. |
| > | Indicates whether the value of the left operand is greater than the value of the right operand. |
| <= | Indicates whether the value of the left operand is less than or equal to the value of the right operand. |
| >= | Indicates whether the value of the left operand is greater than or equal to the value of the right operand. |

Both operands must have arithmetic types or be pointers to the same object type. The result has type `int`.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

**Note:** Function pointers and the AS/400 pointer types system pointer, call pointer, label pointer, and suspend pointer cannot be used as operands in a relational expression.

Relational operators have left-to-right associativity. Therefore, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of `a` is less than the value of `b`, the first relationship is true and supplies the value `1`. The compiler then compares the value `1` with the value of `c`.

The following example shows you how to use a relational operator (less than, <) with packed decimals. In this example, packed decimal types are compared with other arithmetic types (integer, float, double, long double). The implicit conversions are performed using the arithmetic conversion rules in “Usual Arithmetic Conversions” on page 105. Leading zeros in the example are shown to indicate the size of the number of digits in the packed decimal. You do not need to enter leading zeros in your packed decimalvariable.
The expected output is:

```
#include <decimal.h>
#include <stdio.h>
decimal(10,3) pdval = 0000023.423d;    /* Packed decimal declaration*/
int ival = 1233;                       /* Integer declaration      */
float fval = 1234.34f;                 /* Float declaration        */
double dval = 251.5832;                /* Double declaration       */
long double lval = 37486.234;          /* Long double declaration  */
int main(void)
{
  decimal(15,6) value = 000485860.085999d;
  if (pdval < ival) printf("pdval is the smallest !\n"); /* Perform relational   */
  if (pdval < fval) printf("pdval is the smallest !\n"); /* operation between    */
  if (pdval < dval) printf("pdval is the smallest !\n"); /* other data types and */
  if (pdval < lval) printf("pdval is the smallest !\n"); /* packed decimal.      */
  if (pdval < value) printf("pdval is the smallest !\n");
}
```

```
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
```

## Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship that is stated by an equality operator is true, the value of the result is `1`. Otherwise, the value of the result is `0`.

The following table describes the equality operators:

*Table 11. Equality Operators*

| Operator | Usage |
|---|---|
| `==` | Indicates whether the value of the left operand is equal to the value of the right operand. |
| `!=` | Indicates whether the value of the left operand is not equal to the value of the right operand. |

Both operands must have arithmetic types or be pointers to the same type. Otherwise, one operand must have a pointer type and the other operand must be a pointer to void or `NULL`. The result has type `int`.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value `0`, the expression is true only if the pointer operand evaluates to `NULL`.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Where the operands have types and values suitable for the relational operators, the semantics for relational operators applies.

**Note:** Positive zero and negative zero compare equal. For example, the expression `(-0.00d== +0.00000d) always evaluates to TRUE.`

The following example shows you how packed decimal variables are initialized and compared for equality.
The expected output is:

```
#include <decimal.h>
#include <stdio.h>
decimal(1)    op_1 = +0d;              /* Declare and initialize     */
decimal(1)    op_2 = -0d;              /* packed decimal variables.  */
decimal(9,4)  op_3 =  00012.3400d;
decimal(4,2)  op_4 =  12.34d;
int main(void)
{
  if (op_1 == op_2)
  {
    printf("op_1 equals op_2\n");
  }
  if (op_3 != op_4)
  {
    printf("op_3 not equal op_4\n");
  }
}
```

```
op_1 equals op_2
```

## Bitwise AND &

The `&` (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are `1`'s, the corresponding bit of the result is set to `1`. Otherwise, it sets the corresponding result bit to `0`.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.

The compiler may rearrange the operands in an expression that contains more than one bitwise AND operator. This is because the bitwise AND operator has both associative and commutative properties.

The following example shows the values of `a`, `b`, and the result of `a & b` represented as 16-bit binary numbers:

| | |
|---|---|
| bit pattern of `a` | `0000000001011100` |
| bit pattern of `b` | `0000000000101110` |
| bit pattern of `a & b` | `0000000000001100` |

## Bitwise Exclusive OR ˆ

The bitwise exclusive OR operator compares each bit of its first operand to the corresponding bit of the second operand. In EBCDIC, the ¬ symbol represents the ˆ symbol. If both bits are `1`'s or both bits are `0`'s, the corresponding bit of the result is set to `0`. Otherwise, it sets the corresponding result bit to `1`.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

The compiler may rearrange the operands in an expression that contains more than one bitwise exclusive OR operator. This is done because the bitwise exclusive OR operator has both associative and commutative properties.

The following example shows the values of `a`, `b`, and the result of `a ˆ b` represented as 16-bit binary numbers:

| | |
|---|---|
| bit pattern of `a` | `0000000001011100` |
| bit pattern of `b` | `0000000000101110` |
| bit pattern of `a ˆ b` | `0000000001110010` |

## Bitwise Inclusive OR ¦

The ¦ (bitwise inclusive OR) operator compares the values (in binary format) of each operand. It yields a value whose bit pattern shows which bits in either of the operands has the value `1`. If both of the bits are `0`, the result of that bit is `0`; otherwise, the result is `1`.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

The compiler may rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. This is because the bitwise inclusive OR operator has both associative and commutative properties.

The following example shows the values of `a`, `b`, and the result of `a ¦ b` represented as 16-bit binary numbers:

| | |
|---|---|
| bit pattern of `a` | `0000000001011100` |
| bit pattern of `b` | `0000000000101110` |
| bit pattern of `a ¦ b` | `0000000001111110` |

## Logical AND &&

The `&&` (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value `1`. Otherwise, the result has the value `0`.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

The logical AND operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to `0`, the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

| **Expression** | **Result** |
|---|---|
| `1 && 0` | `0` |
| `1 && 4` | `1` |
| `0 && 0` | `0` |

The following example uses the logical AND operator to avoid a divide-by-zero situation:

```
y && (x / y)
```

The expression `x / y` is not evaluated when `y` is `0`.

**Note:** The logical AND (`&&`) should not be confused with the bitwise AND (`&`) operator. For example:

```
   1 && 4 evaluates to 1
while
   1 & 4 evaluates to 0
```

## Logical OR ¦¦

The `¦¦` (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value `1`. Otherwise, the result has the value `0`.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

The logical OR operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

| **Expression** | **Result** |
|---|---|
| `1 ¦¦ 0` | `1` |
| `1 ¦¦ 4` | `1` |
| `0 ¦¦ 0` | `0` |

The following example uses the logical OR operator to conditionally increment `y`:

```
++x ¦¦ ++y;
```

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero quantity.

**Note:** The logical OR (`¦¦`) should not be confused with the bitwise OR (`¦`) operator. For example:

```
   1 ¦¦ 4 evaluates to 1
while
   1 ¦ 4 evaluates to 5
```

## Conditional Expression ? :

A **conditional expression** is a compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value `0`.

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. All expressions that occur between the `?` and `:` are treated as one expression.

The first operand must have a scalar type. The second and third operands must have arithmetic types, compatible structure type, compatible union type, or compatible pointer type. A type is compatible when it has the same type but not necessarily the same qualifiers (`volatile`, `const`, or `_Packed`). Also, the second and third operands may be a pointer and a `NULL` pointer constant, or a pointer to an object and a pointer to void.

The first expression is evaluated first. If the first expression has a nonzero value, the second expression is evaluated, and the third operand is not evaluated. Its result is converted to the result type. If the expression is an arithmetic type, the usual arithmetic conversions on the second expression are performed. If the first expression has a zero value, the third expression is evaluated with the usual arithmetic conversions performed on it if it has an arithmetic type.

The types of the second and third operands determine the type of the result as shown in Table 12.

*Table 12. Type of the Conditional Expression*

| Type of One Operand | Type of Other Operand | Type of Result |
|---|---|---|
| Arithmetic | Arithmetic | Arithmetic after usual arithmetic conversions |
| `struct/union` type | Compatible `struct/union` type | `struct/union` type with all the qualifiers on both operands |
| `void` | `void` | `void` |
| Pointer to compatible type | Pointer to compatible type | Pointer to type with all the qualifiers specified for the type |
| Pointer to type | `NULL` pointer | Pointer to type |
| Pointer to object or incomplete type | Pointer to `void` | Pointer to `void` with all the qualifiers specified for the type |

Conditional expressions have right-to-left associativity.

**Examples**

The following expression determines which variable has the greater value, `y` or `z`, and assigns the greater value to the variable `x`:

```
x = (y > z) ? y : z;
```

The preceding expression is equivalent to the following statement:

```
if (y > z)
   x = y;
else
   x = z;
```

The following expression calls the function `printf()`, which receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, the `printf()` function receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

## Assignment Expression

An **assignment expression** stores a value in the object that is designated by the left operand.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed. The result of an assignment expression is not an lvalue.

All assignment operators have the same precedence and have right-to-left associativity.

There are two types of assignment operators: simple assignment and compound assignment. The following sections describe these operators.

### Simple Assignment =

The simple assignment operator stores the value of the right operand in the object that is designated by the left operand.

Both operands must have arithmetic types, the same structure type, or the same union type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant `0` or `NULL`.

If both operands have arithmetic types, the system converts the type of the right operand to the type of the left operand before the assignment.

If the left operand is a pointer and the right operand is the constant `0`, the result is `NULL`.

Pointers to void can appear on either side of the simple assignment operator.

A packed structure or union can be assigned to a nonpacked structure or union of the same type. A nonpacked structure or union can be assigned to a packed structure or union of the same type.

If one operand is packed and the other is not, the layout of the right operand is remapped to match the layout of the left. This remapping of structures may degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

**Note:** If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked. See “Initializers of Pointers” on page 53 for more information on assignments with pointers.

You can assign values to operands with the type qualifier `volatile`. You cannot assign a pointer of an object with the type qualifier `const` to a pointer of an object without the `const` type qualifier such as in the following example:

```
const int *p1;
int *p2;
p2 = p1;  /* is not allowed */
p1 = p2;  /* note that this is allowed */
```

The following example assigns the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

The following example assigns in order the value `0` to `d`, the value of `d` to `c`, the value of `c` to `b`, and the value of `b` to `a`:

```
a = b = c = d = 0;
```

**Note:** The assignment (`=`) operator should not be confused with the equality comparison (`==`) operator. For example:

**`if(x == 3)`** evaluates to 1 if x is equal to three

while

**`if(x = 3)`** is taken to be true because `(x = 3)` evaluates to a nonzero value (3). The expression also assigns the value 3 to `x`.

## Compound Assignment

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and give the result of that operation to the left operand.

If the left operand of the `+=` and `-=` operators is a pointer, the right operand must have an integral type; otherwise, both operands must have an arithmetic type.

Both operands of the `*=`, `/=`, and `%=` operators must have an arithmetic type.

Both operands of the `<<=`, `>>=`, `&=`, `ˆ=`, and `¦=` operators must have an integral type.

**Note:** The expression `a *= b + c` is equivalent to `a = a * (b + c)`, and **not** `a = a * b + c`.

The following table lists the compound assignment operators and shows an expression using each operator:

*Table 13. Compound Assignment Operators*

| Operator | Example | Equivalent Expression |
|---|---|---|
| `+=` | `index += 2` | `index = index + 2` |
| `-=` | `*(pointer++) -= 1` | `*pointer = *(pointer++) - 1` |
| `*=` | `bonus *= increase` | `bonus = bonus * increase` |
| `/=` | `time /= hours` | `time = time / hours` |

Table 13. Compound Assignment Operators (continued)

| Operator | Example | Equivalent Expression |
|---|---|---|
| `%=` | `allowance %= 1000` | `allowance = allowance % 1000` |
| `<<=` | `result <<= num` | `result = result << num` |
| `>>=` | `form >>= 1` | `form = form >> 1` |
| `&=` | `mask &= 2` | `mask = mask & 2` |
| `ˆ=` | `test ˆ= pre_test` | `test = test ˆ pre_test` |
| `¦=` | `flag ¦= ON` | `flag = flag ¦ ON` |

Although the equivalent expression column shows the left operands (from the example column) that is evaluated twice, the left operand is evaluated only once.

## Comma Expression ,

A **comma expression** contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

If `omega` had the value `11`, the following example would increment `y` and assign the value `3` to `alpha`:

```
alpha = (y++, omega % 4);
```

Any number of expressions that are separated by commas can form a single expression. The compiler evaluates the leftmost expression first. The value of the rightmost expression becomes the value of the entire expression. For example, the value of the following expression is `rotate(direction)`:

```
intensity++, shade * increment, rotate(direction);
```

### Restrictions

You can place comma expressions within lists that contain commas (for example, argument lists and initializer lists). However, because the comma has a special meaning, you must place parentheses around comma expressions in these lists. The comma expression `t = 3, t + 2` is in the following function call:

```
f(a, (t = 3, t + 2), c);
```

The arguments to function `f()` are: the value of `a`, the value `5`, and the value of `c`.

# Chapter 6. Conversions

Many C language operators cause *conversions*. A conversion changes the form of a value and its type. For example, when you add values that have different data types, the compiler converts the types of the objects to the same type before adding the values. Addition of a `short int` value and an `int` value causes the compiler to convert the `short int` value to the `int` type.

Conversions may occur, for example, when:

- A cast operation is performed.
- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has different type from the assigned value.
- A prototyped function is given an argument that has a different type from the parameter.
- The type of the expression that is specified on a function's `return` statement has a different type from the defined return type for the function.

Although the C language contains some guidelines for conversions, many conversions have implementation-specific aspects. These implementation specific aspects occur because:

- The sizes of the data types vary.
- The manner of handling signed data varies.
- The data formats vary.

## Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type. It can also extend short values to the integer size used in machine operations. The conversions that are performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These conversions are known as standard arithmetic conversions because they apply to the types of values that are ordinarily used in arithmetic.

**Integer promotions** are performed when you use a `char`, `short int`, `int`, bit field, or an enumeration type wherever an `int` or `unsigned int` is expected. If an `int` can hold the value, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. All other arithmetic types are unchanged by integer promotions.

"Type Conversions" on page 106 outlines the path of each type of conversion.

Arithmetic conversion proceeds in the following order:

1. If one operand has `long double` type, the other operand is converted to `long double` type.
2. If one operand has `double` type, the other operand is converted to `double`.
3. If one operand has `float` type, the other operand is converted to `float`.
4. If one operand has packed decimal type, the other operand is converted to packed decimal.
5. If one operand has `unsigned long long int` type, the compiler converts the other operand to `unsigned long long int`.

6. If one operand has `long long int` type, the compiler converts the other operand `long long int`.
7. If one operand has `unsigned long int` type, the other operand is converted to `unsigned long int`.
8. If one operand has `unsigned int` type and the other operand has `long int` type and the value of the `unsigned int` can be represented in a `long int`, the operand with `unsigned int` type is converted to `long int`.
9. If one operand has `unsigned int` type and the other operand has `long int` type and the value of the `unsigned int` cannot be represented in a `long int`. Both operands are converted to `unsigned long int`.
10. If one operand has `long int` type, the other operand is converted to `long int`.
11. If one operand has `unsigned int` type, the other operand is converted to `unsigned int`.
12. If both operands have `int` type, the result is type `int`.

## Type Conversions

Type conversions are the assignment of a value to a variable of a different data type, when:

- A value is explicitly cast to another type.
- An operator converts the type of its operand or operands before performing an operation.
- A value is passed as an argument to a function.

The following sections outline the rules governing each kind of conversion.

## Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. ILE C allows conversions by assignment between integer, floating-point and packed decimal types, even when the conversion entails loss of information.

The methods of carrying out the conversions depend on the type, as follows.

## From Signed Integer Types

C converts a signed integer to a shorter signed integer by truncating the high-order bits. The compiler converts a signed integer to a longer signed integer by sign-extension. Conversion of signed integers to floating-point values may result in a loss of precision because floating point representations are inherently inexact. To convert a signed integer to an unsigned integer, you must convert the signed integer to the size of the unsigned integer. The compiler interprets the result as an unsigned value.

The following chart summarizes conversions from signed integer types:

*Table 14. Conversions from Signed Integer*

| From | To | Method |
|---|---|---|
| `signed char` | `short` | Sign-extend. |
| | `int` | Sign-extend. |
| | `long` | Sign-extend. |
| | `long long` | Sign-extend. |
| | `unsigned char` | Preserve pattern; high-order bit loses function as sign bit. |
| | `unsigned short` | Sign-extend to `short`; convert `short` to `unsigned short`. |
| | `unsigned long` | Sign-extend to `long`; convert `long` to `unsigned long`. |
| | `unsigned long long` | Sign-extend to `long long`; convert `long long` to `unsigned long long`. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(10,0)` and then convert to the required packed decimal type. |
| | `float` | Sign-extend to `long`; convert `long` to `float`. |
| | `double` | Sign-extend to `long`; convert `long` to `double`. |
| | `long double` | Sign-extend to `long`; convert `long` to `long double`. |
| `short` | `signed char` | Preserve low-order byte. |
| | `int` | Sign-extend. |
| | `long` | Sign-extend. |
| | `long long` | Sign-extend. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve bit pattern; high-order bit loses function as sign bit. |
| | `unsigned long` | Sign-extend to `long`; convert `long` to `unsigned long`. |
| | `unsigned long long` | Sign-extend to `long long`; convert `long long` to `unsigned long long`. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(10,0)` and then convert to the required packed decimal type. |
| | `float` | Sign-extend to `long`; convert `long` to `float`. |
| | `double` | Sign-extend to `long`; convert `long` to `double`. |
| | `long double` | Sign-extend to `long`; convert `long` to `long double`. |

*Table 14. Conversions from Signed Integer (continued)*

| From | To | Method |
|---|---|---|
| `int` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `long` | No conversion. |
| | `long long` | Sign-extend. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned long` | Preserve bit pattern; high-order bit loses function as sign bit. |
| | `unsigned long long` | Sign-extend to `long long`; convert `long long` to `unsigned long long`. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(10,0)` and then convert to the required packed decimal type. |
| | `float` | Represent as a `float`; if the `long` cannot be represented exactly, some loss of precision occurs. |
| | `double` | Represent as a `double`; if the `long` cannot be represented exactly, some loss of precision occurs. |
| | `long double` | Represent as a `long double`; if the `long` cannot be represented exactly, some loss of precision occurs. |
| `long` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `int` | No conversion. |
| | `long long` | Sign extend. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned long` | Preserve bit pattern; high-order bit loses function as sign bit. |
| | `unsigned long long` | Sign-extend to `long long`; convert `long long` to `unsigned long long`. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(10,0)` and then convert to the required packed decimal type. |
| | `float` | Represent as a `float`; if the `long` cannot be represented exactly, some loss of precision occurs. |
| | `double` | Represent as a `double`; if the `long` cannot be represented exactly, some loss of precision occurs. |
| | `long double` | Represent as a `long double`; if the `long` cannot be represented exactly, some loss of precision occurs. |

*Table 14. Conversions from Signed Integer (continued)*

| From | To | Method |
|---|---|---|
| `long long` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `int` | Preserve low-order bytes. |
| | `long` | Preserve low-order bytes. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned int` | Preserve low-order bytes. |
| | `unsigned long` | Preserve low-order bytes. |
| | `unsigned long long` | Preserve bit pattern; high-order bit loses function as sign bit. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(20,0)` and then convert to the required packed decimal type. |
| | `float` | Represent as a `float`; if the `long long` cannot be represented exactly, some loss of precision occurs. |
| | `double` | Represent as a `double`; if the `long long` cannot be represented exactly, some loss of precision occurs. |
| | `long double` | Represent as a `long double`; if the `long long` cannot be represented exactly, some loss of precision occurs. |

## From Unsigned Integer Types

The compiler converts an unsigned integer to a shorter unsigned or signed integer by truncating the high-order bits. The compiler converts an unsigned integer to a longer unsigned or signed integer by setting the high-order bits to `0`. The compiler converts unsigned values to floating-point values by first converting to a signed integer of the same size. The compiler then converts the signed value to a floating-point value. Conversion of unsigned integers to floating-point values may result in a loss of precision because floating point representations are inherently inexact.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value changes if the sign bit is set.

The following chart summarizes conversions from unsigned integer types:

*Table 15. Conversions from Unsigned Integer Types*

| From | To | Method |
|---|---|---|
| `unsigned char` | `signed char` | Preserve bit pattern; high-order bit becomes sign bit. |
| | `short` | Zero-extend; preserve the bit pattern. |
| | `int` | Zero-extend; preserve the bit pattern. |
| | `long` | Zero-extend; preserve the bit pattern. |
| | `long long` | Zero-extend; preserve the bit pattern. |
| | `unsigned short` | Zero-extend; preserve the bit pattern. |
| | `decimal(`*n*`,`*p*`)` | Converted to `decimal(10,0)` and then converted to the required packed decimal type. |
| | `unsigned int` | Zero-extend; preserve the bit pattern. |
| | `unsigned long` | Zero-extend; preserve the bit pattern. |
| | `unsigned long long` | Zero-extend; preserve the bit pattern. |
| | `float` | Convert to `long`; convert `long` to `float`. |
| | `double` | Convert to `long`; convert `long` to `double`. |
| | `long double` | Convert to `long`; convert `long` to `long double`. |
| `unsigned short` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve bit pattern; high-order bit becomes sign bit. |
| | `int` | Zero-extend; preserve the bit pattern. |
| | `long` | Zero-extend; preserve the bit pattern. |
| | `long long` | Zero-extend; preserve the bit pattern. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned int` | Zero-extend. |
| | `unsigned long` | Zero-extend. |
| | `unsigned long long` | Zero-extend. |
| | `decimal(`*n*`,`*p*`)` | Converted to `decimal(10,0)` and then converted to the required packed decimal type. |
| | `float` | Convert to `long`; convert `long` to `float`. |
| | `double` | Convert to `long`; convert `long` to `double`. |
| | `long double` | Convert to `long`; convert `long` to `long double`. |

*Table 15. Conversions from Unsigned Integer Types (continued)*

| From | To | Method |
|---|---|---|
| `unsigned int` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `int` | Preserve bit pattern; high-order bit becomes sign. |
| | `long` | Preserve bit pattern; high-order bit becomes sign. |
| | `long long` | Zero-extend. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned long long` | Zero-extend. |
| | `decimal(`*n*`,`*p*`)` | Converted to `decimal(10,0)` and then converted to the required packed decimal type. |
| | `float` | Convert `int` to `float`. |
| | `double` | Convert `int` to `double`. |
| | `long double` | Convert `int` to `long double`. |
| `unsigned long` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `int` | Preserve bit pattern; high-order bit becomes sign. |
| | `long` | Preserve bit pattern; high-order bit becomes sign. |
| | `long long` | Zero-extend. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned int` | No conversion. |
| | `unsigned long long` | Zero-extend. |
| | `decimal(`*n*`,`*p*`)` | Converted to `decimal(10,0)` and then converted to the required packed decimal type. |
| | `float` | Convert `long` to `float`. |
| | `double` | Convert `long` to `double`. |
| | `long double` | Convert `long` to `long double`. |

Table 15. Conversions from Unsigned Integer Types (continued)

| From | To | Method |
|---|---|---|
| `unsigned long long` | `signed char` | Preserve low-order byte. |
| | `short` | Preserve low-order bytes. |
| | `int` | Preserve low-order bytes. |
| | `long` | Preserve low-order bytes. |
| | `long long` | Preserve bit pattern; high-order bit becomes sign bit. |
| | `unsigned char` | Preserve low-order byte. |
| | `unsigned short` | Preserve low-order bytes. |
| | `unsigned int` | Preserve low-order bytes. |
| | `unsigned long` | Preserve low-order bytes. |
| | `decimal(`*n*`,`*p*`)` | Convert to `decimal(20,0)` and then convert to the required packed decimal type. |
| | `float` | Represent as a `float`; if the `unsigned long long` cannot be represented exactly, some loss of precision occurs. |
| | `double` | Represent as a `double`; if the `unsigned long long` cannot be represented exactly, some loss of precision occurs. |
| | `long double` | Represent as a `long double`; if the `unsigned long long` cannot be represented exactly, some loss of precision occurs. |

## From Floating-Point Types

A `float` value that is converted to a `double` undergoes no change in value. A `double` that is converted to a `float` is represented exactly, if possible. If C cannot exactly represent the `double` value as a `float`, the number loses precision. If the value is too large to fit into a `float`, the number is undefined. If a floating-point value is converted to a packed decimal with a smaller precision, the fractional part of the floating point number is truncated. If the value is too large to fit into a packed decimal, the number is undefined.

A `float` has 6 significant digits. A `double` has 15 significant digits and is represented internally as a value having 15 significant digits. As a result, when a `double` is converted to a packed decimal, it is the internal representation of the `double` value which is used. If this value of the `double` cannot be represented in the packed decimal, then the fractional part of the internal `double` value is truncated.

A floating-point value is converted to an integer value by converting to an `unsigned long`. The decimal fraction portion of the floating-point value is discarded in the conversion. If the result is still too large to fit, the result of the conversion is undefined.

The following chart summarizes conversions from floating-point types:

*Table 16. Conversions from Floating-Point Types*

| From | To | Method |
|---|---|---|
| `float` | `signed char` | Convert to `long`; convert `long` to `signed char`. |
| | `short` | Convert to `long`; convert `long` to `short`. |
| | `int` | Truncate at decimal point; if result is too large to be represented as a `int`, result is undefined. |
| | `long` | Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined. |
| | `long long` | Truncate at decimal point; if result is too large to be represented as a `long long`, the result is undefined. |
| | `unsigned short` | Convert to `unsigned long`; convert `unsigned long` to `unsigned short`. |
| | `unsigned int` | Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined. |
| | `unsigned long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long`, result is undefined. |
| | `unsigned long long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long long`, the result is undefined. |
| | `decimal(`*n*`,`*p*`)` | Convert to the required packed decimal type. Truncate the fractional part of the float to be converted if it cannot be represented in the packed decimal format. If the result is too large to be represented as a decimal(*n*,*p*), the result is undefined. |
| | `double` | Represent as a `double`. |

*Table 16. Conversions from Floating-Point Types (continued)*

| From | To | Method |
|---|---|---|
| `double` | `signed char` | Convert to `float`; convert `float` to `char`. |
| | `short` | Convert to `float`; convert `float` to `short`. |
| | `int` | Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined. |
| | `long` | Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined. |
| | `long long` | Truncate at decimal point; if result is too large to be represented as a `long long`, result is undefined. |
| | `unsigned short` | Convert to `unsigned long`; convert `unsigned long` to `unsigned short`. |
| | `unsigned int` | Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined. |
| | `unsigned long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long`, result is undefined. |
| | `unsigned long long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long long`, result is undefined. |
| | `decimal(`*n*`,`*p*`)` | Convert to the required packed decimal type. Truncate the fractional part of the double to be converted if it cannot be represented in the packed decimal format. If the result is too large to be represented as a decimal(*n*,*p*), the result is undefined. |
| | `float` | Represent as a float; if the `double` value cannot be represented exactly as a `float`, loss of precision occurs; if the value is too large to be represented in a `float`, the result is undefined. |
| | `long double` | Represent as a `long double`. |

*Table 16. Conversions from Floating-Point Types (continued)*

| From | To | Method |
|---|---|---|
| `long double` | `signed char` | Convert to `double`; convert `double` to `float`; convert `float` to `char`. |
| | `short` | Convert to `double`; convert `double` to `float`; convert `float` to `short`. |
| | `int` | Truncate at decimal point; if result is too large to be represented as a `int`, result is undefined. |
| | `long` | Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined. |
| | `long long` | Truncate at decimal point; if result is too large to be represented as a `long long`, result is undefined. |
| | `unsigned short` | Convert to `double`; convert `double` to `unsigned long`; convert `unsigned long` to `unsigned short`. |
| | `unsigned int` | Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined. |
| | `unsigned long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long`, result is undefined. |
| | `unsigned long long` | Truncate at decimal point; if result is too large to be represented as an `unsigned long long`, result is undefined. |
| | `decimal(`*n*`,`*p*`)` | Convert to the required packed decimal type. Truncate the fractional part of the long double to be converted if it cannot be represented in the packed decimal format. If the result is too large to be represented as a decimal(*n*,*p*), the result is undefined. |
| | `float` | Convert to `double`; represent as a `float`; if the `long double` value cannot be represented exactly as a `float`, loss of precision occurs; if the value is too large to be represented in a `float`, the result is undefined. |
| | `double` | Represent as a `double`; If the result is too large to be represented as a `double`, result is undefined. |
| | `long double` | Represent as a `long double`; If the result is too large to be represented as a `long double`, result is undefined. |

The following chart summarizes conversions from packed decimal types:

Table 17. Conversions from Packed Decimal Types

| From | To | Method |
|---|---|---|
| decimal(*n*,*p*) | signed char | Truncate the fractional part and then convert to signed char. |
| | short | Truncate the fractional part and then convert to short. |
| | int | Truncate the fractional part and then convert to int. |
| | long | Truncate the fractional part and then convert to long. |
| | long long | Truncate the fractional part and then convert to long long. |
| | unsigned short | Truncate the fractional part and then convert to unsigned short. |
| | unsigned char | Truncate the fractional part and then convert to unsigned char. |
| | unsigned int | Truncate the fractional part and then convert to unsigned int. |
| | unsigned long | Truncate the fractional part and then convert to unsigned long. |
| | unsigned long long | Truncate the fractional part and then convert to unsigned long long. |
| | decimal(*n*,*p*) | Convert to the required packed decimal type, truncating from the integral or fractional part, if the integral or the fractional part of the packed decimal target is smaller. |
| | float | Convert to float. If the value is too big to be represented exactly as a float, the result is undefined. |
| | double | Convert to double. If the value is too big to be represented exactly as a double, the result is undefined. |
| | long double | Convert to long double. If the value is too big to be represented exactly as a long double, the result is undefined. |

## To and From Pointer Types

You can convert a pointer to one type of value to a pointer to a different type.

Conversion of AS/400 pointers is subject to certain restrictions. See the *ILE C for AS/400 Programmer's Guide* for more information on AS/400 pointer conversion.

You can convert a pointer value to an integral value. The path of the conversion depends on the size of the pointer and the size of the integral type.

The conversion of an integer value to an address offset (in an expression with an integral type operand and a pointer type operand) is system dependent.

A pointer to a constant or a volatile object should never be assigned to a non-constant or non-volatile object.

A pointer to `void` can be converted to or from a pointer to any incomplete or object type.

## From Other Types

When you define a value using the `enum` type specifier, the value is treated as an `int`. Conversions to and from an `enum` value proceed as for the `int` type.

When a packed structure or union is assigned to a nonpacked structure or union of the same type or an nonpacked structure is assigned to, a packed structure or union of the same type, the layout of the right operand is remapped to match the layout of the left.

No other conversions between structure or union types are allowed.

The `void` type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to `void` by assignment. However, a value can be explicitly cast to `void`.

# Chapter 7. C Language Statements

This chapter describes the `label` identifier and statement, and the following C language statements.

- Block
- break
- continue
- do
- Expression
- for
- goto
- if
- Null
- return
- switch
- while

## Labels

A **label** is an identifier that allows your program to transfer control to other statements within the same function. It is the only type of identifier that has function scope (see “Scope” on page 4). The `goto` or `switch` statements transfers control to the statement that follows the label. In ILE C, the control is transferred to a label when an exception occurs while the function that contains the label is running. This can only happen if the label has been named as the branch point handler on a #pragma exception_handler directive. A label has the form:

```
►►──identifier──:──statement──────────────────────────────────────────►◄
```

For example, the following are labels:

```
comment_complete: ;               /* Example of null statement label */
test_for_null: if (NULL == ptr) /* Example of statement label      */
```

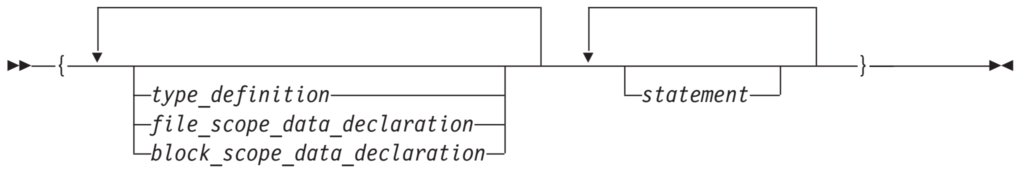
The `case` and `default` labels have a specific use and are described later in this chapter. See “switch” on page 131.

**Related Information**

- “goto” on page 127
- “switch” on page 131
- “exception_handler” on page 163

## Block

A **block statement** enables you to group any number of data definitions, declarations, and statements into one statement. When you enclose definitions, declarations, and statements within a single set of braces, everything within the braces is treated as a single statement. You can place a block wherever a statement is allowed. The block statement has the form:



All definitions and declarations occur at the beginning of a block before statements. Statements must follow the definitions and declarations. A block is considered a single statement.

If you redefine a data object inside a nested block, the inner object hides the outer object while the inner block is run. Defining several variables that have the same identifier can make a program difficult to understand and maintain. Therefore, you should limit such redefinitions of identifiers within nested blocks.

If a data object is usable within a block, all nested blocks can use that data object (unless that data object identifier is redefined).

Initialization of an `auto` or `register` variable occurs each time the block is run from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed. You cannot initialize an `extern` variable within a block.

**Examples**

The following example shows how the values of data objects change in nested blocks:

```
#include <stdio.h>

int main(void)
{
   int x = 1;                     /* Initialize x to 1  */
   int y = 3;

   if (y > 0)
   {
      int x = 2;                  /* Initialize x to 2  */
      printf("second x = %4d\n", x);
   }
   printf("first  x = %4d\n", x);
}
```

The preceding example produces the following output:

```
second x =    2
first  x =    1
```

Two variables that are named `x` are defined in `main()`. The definition of `x` on line `5` keeps storage throughout the processing of `main()`. However, because the definition of `x` on line `10` occurs within a nested block, line `11` recognizes `x` as the variable defined on line `10`. Line `13` is not part of the nested block. Thus, line `13` recognizes `x` as the variable defined on line `5`.

## break

A **break statement** enables you to stop and exit from a loop or `switch` statement. This statement can be used from any point within the loop or `switch` other than the logical end. A `break` statement has the form:

```
►►──break──;─────────────────────────────────────────────────────────►◄
```

In a looping statement ( `do`, `for`, or `while`), the `break` statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the `break` statement ends only the smallest enclosing `do`, `for`, `switch`, or `while` statement.

In a `switch` body, the `break` statement ends the processing of the `switch` body and gives control to the next statement outside the `switch` body.

### Restrictions

You can place a `break` statement only in the body of a looping statement (`do, for` or `while`). A `break` statement can also be used in the body of a `switch` statement.

### Examples

The following example shows a `break` statement in the action part of a `for` statement. If the `i`th element of the array `string` is equal to `'\0'`, the `break` statement causes the `for` statement to end.

```
for (i = 0; i < 5; i++)
{
   if (string[i] == '\0')
      break;
   length++;
}
```

The preceding `for` statement is equivalent to the following `for` statement, if `string` does not contain any embedded null characters:

```
for (i = 0; i < 5; i++)
{
   if (string[i] != '\0')
      length++;
}
```

The following example shows a `break` statement in a nested looping statement. The outer loop sequences an array of pointers to strings. The inner loop examines each character of the string. When the `break` statement is run, the inner loop ends, and control returns to the outer loop.

```
/*
** This program counts the characters in the strings that are
** part of an array of pointers to characters.  The count stops
** when one of the digits 0 through 9 is encountered
** and resumes at the beginning of the next string.
*/

#include <stdio.h>

#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;

   for (i = 0; i < SIZE; i++)        /* for each string    */
                                     /* for each character */
      for (pointer = strings[i]; *pointer != '\0'; ++pointer)
      {                              /* if a number        */
         if (*pointer >= '0' && *pointer <= '9')
            break;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);
}
```

The preceding program produces the following output:

```
letter count = 4
```

The following example is a `switch` statement that contains several `break` statements. Each `break` statement indicates the end of a specific clause and ends the processing of the `switch` statement.

```
enum {morning, afternoon, evening} timeofday;

switch (timeofday)
{
   case (morning):
      printf("Good Morning\n");
      break;

   case (evening):
      printf("Good Evening\n");
      break;

   default:
      printf("Good Day, eh\n");
      break;
}
```

**Related Information**

- “do” on page 124
- “for” on page 125
- “switch” on page 131
- “while” on page 134

## continue

A **continue statement** enables you to end the current iteration of a loop. Program control is passed from the location in the body of the loop in which the `continue` statement is found to the end of the loop body. A `continue` statement has the form:

```
►►──continue──;────────────────────────────────────────────────────►◄
```

The `continue` statement ends the processing of the action part of a `do`, `for`, or `while` statement. This moves control to the condition part of the statement. If the looping statement is a `for` statement, control moves to the third expression in the condition part of the statement. Then control moves to the second expression (the test) in the condition part of the statement.

Within nested statements, the `continue` statement ends only the current iteration of the `do`, `for`, or `while` statement immediately enclosing it.

### Restrictions

You can place a `continue` statement only within the body of a looping statement (`do`, `for` or `while`).

### Examples

The following example shows a `continue` statement in a `for` statement. The `continue` statement causes the system to skip over those elements of the array `rates` that have values less than or equal to `1`.

```
#include <stdio.h>

#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }
}
```

The preceding program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a `continue` statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop is ended. Processing continues with the third expression of the inner loop (See "for" on page 125). The inner loop is ended when the `'\0'` escape sequence is encountered.

```
/************************************************************************
**     This program counts the characters in strings that are part     **
**     of an array of pointers to characters.  The count excludes      **
**     the digits 0 through 9.                                         **
************************************************************************/

#include <stdio.h>

#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)                /* for each string         */
                                             /* for each character */
      for (pointer = strings[i]; *pointer != '\0'; ++pointer)
      {                                 /* if a number             */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);
}
```

The preceding program produces the following output:

```
letter count = 5
```

Compare the preceding program with the program on page “break” on page 121. The program on page “break” on page 121 shows the use of the `break` statement, but performs a similar function.

**Related Information**

- “do”
- “for” on page 125
- “while” on page 134

## do

A **do statement** repeatedly runs a statement until the test expression evaluates to `0`. Because of the order of processing, the statement is run at least once.

```
►►──do──statement──while──(──expression──)──;──────────────────────────────►◄
```

The body of the loop is run before the `while` clause (the controlling expression) is evaluated. Further processing of the `do` statement depends on the value of the `while` clause. If the `while` clause does not evaluate to `0`, the statement is run again. Otherwise, the statement is no longer run.

The controlling expression must be of scalar type.

A `break`, `return`, or `goto` statement can cause the execution of a `do` statement to end, even when the `while` clause does not evaluate to `0`.

**Example**

The following statement prompts the user to enter a 1. If the user enters a 1, the statement ends. Otherwise, the statement displays another prompt.

```
#include <stdio.h>

int main (void)
{
   int reply1;

   do
   {
      printf("Enter a 1\n");
      scanf("%d", &reply1);
   } while (reply1 != 1);
}
```

**Related Information**

- “break” on page 121
- “continue” on page 123
- “while” on page 134

## Expression

An **expression statement** contains an expression. Expressions are described in “Chapter 5. Expressions and Operators” on page 79. An expression statement has the form:

```
►►──┬────────────┬─;──────────────────────────────────────────────►◄
    └─expression─┘
```

An expression statement evaluates the given expression. An expression statement is used to assign the value of the expression to a variable or to call a function.

**Examples**

```
printf("Account Number: \n");                  /* A call to printf          */
marks = dollars * exch_rate;                   /* An assignment to marks    */
(difference < 0) ? ++losses : ++gain;          /* A conditional increment   */
switches = flags ¦ BIT_MASK;                   /* An assignment to switches */
```

**Related Information**

- “Chapter 5. Expressions and Operators” on page 79

## for

A **for statement** enables you to do the following:

- Evaluate an expression prior to the first iteration of the statement (“initialization”)
- Specify an expression to determine whether or not the statement should be run (“controlling part”)
- Evaluate an expression after each iteration of the statement

A `for` statement has the form:

```
►►──for──(──┬─────────────┬──;──┬─────────────┬──;──┬─────────────┬──────────►
            └─expression1─┘     └─expression2─┘     └─expression3─┘

►──)──statement──────────────────────────────────────────────────────────────►◄
```

The compiler evaluates *expression1* only once. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

The compiler evaluates *expression2* before each run of the statement. *expression2* must evaluate to a scalar type. If this expression evaluates to `0`, the `for` statement is ended and control moves to the statement following the `for` statement. Otherwise, the statement is run. If you omit *expression2*, it will be as if the expression had been replaced by a nonzero constant and the `for` statement will not be terminated by failure of this condition.

The compiler evaluates *expression3* after each run of the statement. You can use this expression to increase, decrease, or reinitialize a variable. If you do not want to evaluate an expression after each iteration of the statement, you can omit this expression.

A `break`, `return`, or `goto` statement can cause the execution of a `for` statement to end, even when the second expression does not evaluate to `0`. If you omit *expression2*, you must use a `break`, `return`, or `goto` statement to stop the processing of the `for` statement.

**Examples**

The following `for` statement prints the value of `count` `20` times. The `for` statement initially sets the value of `count` to `1`. After each run of the statement, `count` is increased.

```
for (count = 1; count <= 20; count++)
   printf("count = %d\n", count);
```

For comparison purposes, the preceding example can be written using the following sequence of statements to accomplish the same task. Note the use of the `while` statement instead of the `for` statement.

```
count = 1;
while (count <= 20)
{
   printf("count = %d\n", count);
   count++;
}
```

The following `for` statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d\n", index, list[index]);
}
```

The following `for` statement will continue running until `scanf()` receives the letter `e`:

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '\n')
      continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c\n", letter);
}
```

The following `for` statement contains multiple initializations and increments. The comma operator makes this construction possible.

```
for (i = 0, j = 50; i < 10; ++i, j += 50)
{
    printf("i = %2d and j = %3d\n", i, j);
}
```

The following example shows a nested `for` statement. The outer statement is run as long as the value of `row` is less than `5`. Each time the outer `for` statement is run, the inner `for` statement sets the initial value of `column` to zero. The statement of the inner `for` statement is run `3` times. The inner statement is run as long as the value of `column` is less than `3`. This example prints the values of an array that has the dimensions `[5][3]`:

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d\n", table[row][column]);
```

**Related Information**

- "break" on page 121
- "continue" on page 123

## goto

A **goto statement** causes your program to unconditionally transfer control to the statement that is associated with the label that is specified on the `goto` statement. A `goto` statement has the form:

```
►►──goto──identifier──;──────────────────────────────────────────►◄
```

The `goto` statement transfers control to the statement that is indicated by the identifier.

**Notes**

Use the `goto` statement sparingly. Because the `goto` statement can interfere with the normal top-to-bottom sequence of processing, it makes a program more difficult to read and maintain. Often, a `break` statement, a `continue` statement, or a function call can eliminate the need for a `goto` statement.

If you use a `goto` statement to transfer control to a statement inside a loop or block, initializations of automatic storage for the loop do not take place and the result is undefined. The label must appear in the same function as the `goto`.

### Examples

The following example shows a `goto` statement that is used to jump out of a nested loop. A `goto` statement is not necessary in this function.

```
void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) ¦¦ (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

## if

An **if statement** allows you to conditionally run a statement when the specified test expression evaluates to a nonzero value. The expression must have a scalar type. You may optionally specify an `else` clause on the `if` statement. If the test expression evaluates to `0` and an `else` clause exists, the statement associated with the `else` clause is run. An `if` statement has the form:

```
►►──if──(──expression──)──statement──┬──────────────────┬──────────────────►◄
                                      └──else──statement─┘
```

When `if` statements are nested and `else` clauses are present, a given `else` is associated with the closest preceding `if` statement within the same block.

### Examples

The following example causes `grade` to receive the value `A` if the value of `score` is greater than or equal to `90`.

```
if (score >= 90)
   grade = 'A';
```

The following example displays `number is positive` if the value of `number` is greater than or equal to `0`. Otherwise, the example displays `number is negative`.

```
if (number >= 0)
   printf("number is positive\n");
else
   printf("number is negative\n");
```

The following example shows a nested `if` statement:

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
```

The following example shows an `if` statement that does not have an `else` clause. Because an `else` clause always associates with the closest `if` statement, braces may be necessary to force a particular `else` clause to associate with the correct `if` statement. In this example, omitting the braces would cause the `else` clause to associate with the nested `if` statement.

```
if (gallons > 0) {
   if (miles > gallons)
      mpg = miles/gallons;
}
else
   mpg = 0;
```

The following example shows an `if` statement nested within an `else` clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire `if` statement ends.

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

## Null Statement

A **null statement** performs no operation and has the form:

```
▶▶──;──────────────────────────────────────────────────────────▶◀
```

You can use a null statement in a looping statement to show a nonexistent action or in a labeled statement to hold the label.

**Example**

The following example initializes the elements of the array `price`. Because the initializations occur within the `for` expressions, a statement is only needed to finish the `for` syntax; no operations are required:

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

## return

A **return statement** ends the processing of the current function and returns control to the caller of the function. A `return` statement has the form:

```
►►──return──┬────────────┬──;──────────────────────────────────────────►◄
            └─expression─┘
```

A `return` statement ends the processing of the current function.

A `return` statement is optional. If the system reaches the end of a function without encountering a `return` statement, control is passed to the caller as if a `return` statement without an expression were encountered.

A function can contain multiple `return` statements.

**Value**

If an expression is present on a `return` statement, the value of the expression is returned to the caller. If the data type of the expression is different from the data type of the function, conversion of the return value takes place as if the value of the expression were assigned to an object with the same data type as the function.

If an expression is not present on a `return` statement, the value of the `return` statement is not defined. If an expression is not given on a `return` statement and the calling function is expecting a value to be returned, the resulting behavior is undefined.

You cannot use a `return` statement with an expression when the function is declared as returning type `void`.

**Examples**

```
return;            /* Returns no value            */
return result;     /* Returns the value of result */
return 1;          /* Returns the value 1         */
return (x * x);    /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match()` returns the value of `i`. If a match does not exist, the function `match()` returns the value `-1` (negative one).

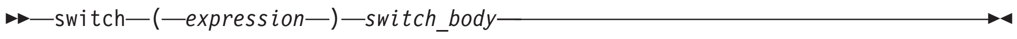
```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

**Related Information**

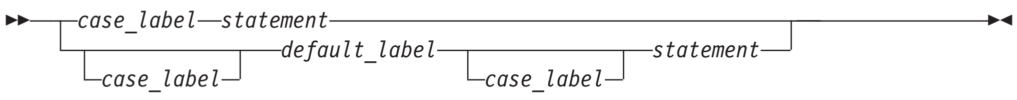
- “Chapter 4. Functions” on page 69
- “Expression” on page 125

## switch

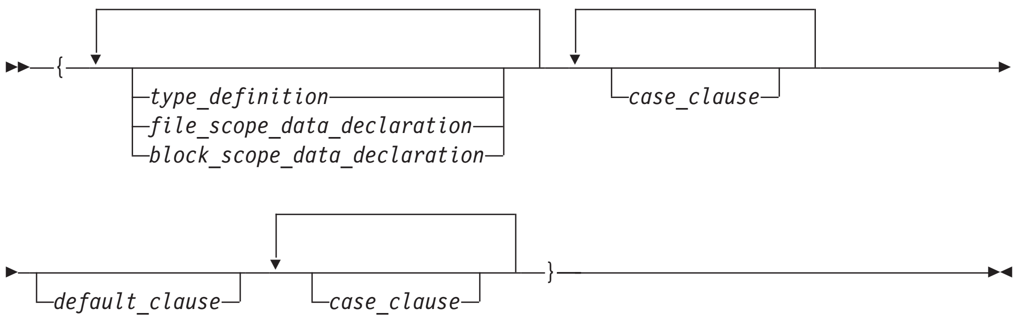
A **switch statement** enables you to transfer control to different statements within the switch body that depends on the value of the switch expression. The switch expression must have an integral type. Within the body of the switch statement, there are case labels that consist of a label, a case expression (that evaluates to an integral value), and statements, plus an optional default label. If the value of the switch expression equals one of the case expressions' values, the statements following that case expression are run. Otherwise, the default label statements, if any, are run. A switch statement has the form:



A *switch body* can have a simple or complex form. The simple form contains any number of case labels that are mixed with an optional default label. The simple form ends with a single statement. The simple form of the switch body is rarely used in C language programs because only the final case or default label can be followed by a statement. An if statement usually can replace a switch statement that has a simple switch body. The simple form of a switch body is shown below:

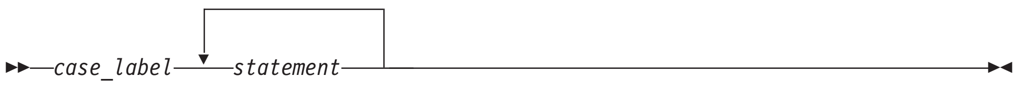


The complex form of a switch body, that is enclosed in braces, can contain definitions, declarations, case clauses, and a default clause. Each case and default clause can contain statements. The complex form of a switch body is shown below:

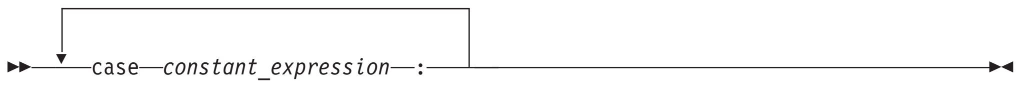


**Note:** If you include an initializer within a *type_definition*, *extern_definition* or *internal_data_definition*, it is ignored.

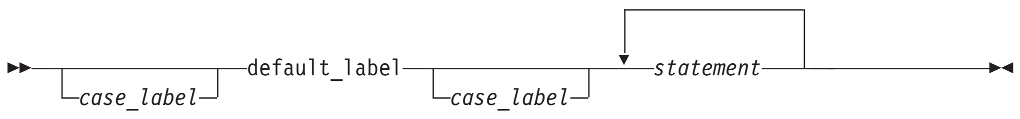
A *case_clause* contains a case label followed by any number of statements. A case clause has the form:



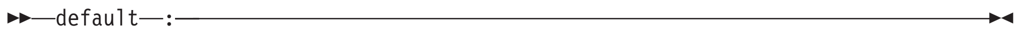
A *case_label* contains the word `case` that is followed by a constant expression and a colon. Anywhere you can place one `case` label; you can place multiple `case` labels. A `case` label has the form:



A *default_clause* contains a `default` label that is followed by one or more statements. You can place a `case` label on either side of the `default` label. A *default_clause* has the form:



A *default_label* contains the word `default` and a colon. A `switch` statement can have only one `default` label. A `default` label has the form:



The `switch` statement passes control to the statement that follows one of the labels or to the statement that follows the `switch` body. The value of the expression that precedes the `switch` body determines which statement receives control. This expression is called the *switch expression*.

The value of the `switch` expression is compared with the value of the expression in each `case` label. If a matching value is found, control passes to the statement following the `case` label that contains the matching value. If the system does not find a matching value and a `default` label appears anywhere in the `switch` body, control passes to the `default` labelled statement. Otherwise, no part of the `switch` body runs.

If control passes to a statement in the `switch` body, control does not pass from the `switch` body until a `break` statement is encountered or the last statement in the `switch` body is processed.

An integral promotion is performed on the controlling expression, if necessary. All expressions in the `case` statements are converted to the same type as the controlling expression.

**Restrictions**

The `switch` expression and the `case` expressions must have an integral type. The value of each `case` expression must represent a different value and must be a constant expression.

Only one `default` label can occur in each `switch` statement.

You can place data definitions at the beginning of the `switch` body. However, the compiler does not initialize `auto` and `register` variables at the beginning of a `switch` body.

**Examples**

The following switch statement contains several case clauses and one default clause. Each clause contains a function call and a break statement. The break statements prevent control from passing down through each statement in the switch body.

If the switch expression evaluated to '/', the switch statement would call the function divide. Control would then pass to the statement following the switch body.

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

   default:
      printf("invalid key\n");
      break;
}
```

If the switch expression matches a case expression, the statements following the case expression are executed until a break statement is encountered or the end of the switch body is reached. In the following example, break statements are not present. If the value of text[i] is equal to 'A', all three counters are increased. If the value of text[i] is equal to 'a', lettera and total are increased. Only total is increased if text[i] is not equal to 'A' or 'a'.

```
char text[100];
int capa, lettera, total;

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
       case 'A':
         capa++;
       case 'a':
         lettera++;
       default:
         total++;
    }
}
```

The following switch statement performs the same statements for more than one case label:

```
int month;
switch (month)
{
   case 12:
   case 1:
   case 2:
      printf("month %d is a winter month\n", month);
      break;

   case 3:
   case 4:
   case 5:
      printf("month %d is a spring month\n", month);
      break;

   case 6:
   case 7:
   case 8:
      printf("month %d is a summer month\n", month);
      break;

   case 9:
   case 10:
   case 11:
      printf("month %d is a fall month\n", month);
      break;

   default:
      printf("not a valid month\n");
      break;
}
```

If the expression `month` had the value `3`, control would be passed to the statement:

```
printf("month %d is a spring month\n", month);
```

The `break` statement would pass control to the statement that follows the `switch` body.

**Related Information**

- "break" on page 121.

## while

A **while statement** enables you to repeatedly run the body of a loop until the controlling expression evaluates to `0`. A `while` statement has the form:

```
►►──while──(──expression──)──statement───────────────────────────────►◄
```

The expression is evaluated to determine whether or not the body of the loop should be run. The expression must be a scalar type. If the expression evaluates to `0`, the statement ends, and the body of the loop is never run. Otherwise, the body is run. After the body has been run, control returns to the expression. Further processing of the action depends on the value of the condition.

A `break`, `return`, or `goto` statement can cause the running of a `while` statement to end, even when the condition does not evaluate to `0`.

**Examples**

In the following program, `item[index]` triples each time the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the `while` statement ends.

```
#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))

#include <stdio.h>

int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d\n", index, item[index]);
      ++index;
   }
}
```

**Related Information**

- “break” on page 121
- “continue” on page 123

# Chapter 8. Preprocessor Directives

This chapter describes the C preprocessor directives. **Preprocessing** is a step in the compilation process that enables you to:

- Replace tokens in the current file with specified replacement tokens. A token is a series of characters that are delimited by white space. The only white space that is allowed on a preprocessor directive is the space, horizontal tab, and comments.
- Imbed files within the current file
- Conditionally compile sections of the current file
- Change the line number of the next line of source and change the file name of the current file
- Generate diagnostic messages

The preprocessor recognizes the following directives:

- `#define`
- `#undef`
- `#error`
- `#include`
- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`
- `#line`
- `#pragma`

**Note:** The # is not part of the directive's name and can be separated from the name with white spaces.

Most preprocessor directives can appear anywhere in a program. Some #pragma directives have restrictions on where they can appear and how often they can appear in a program. See the descriptions of the individual #pragma directives for more information.

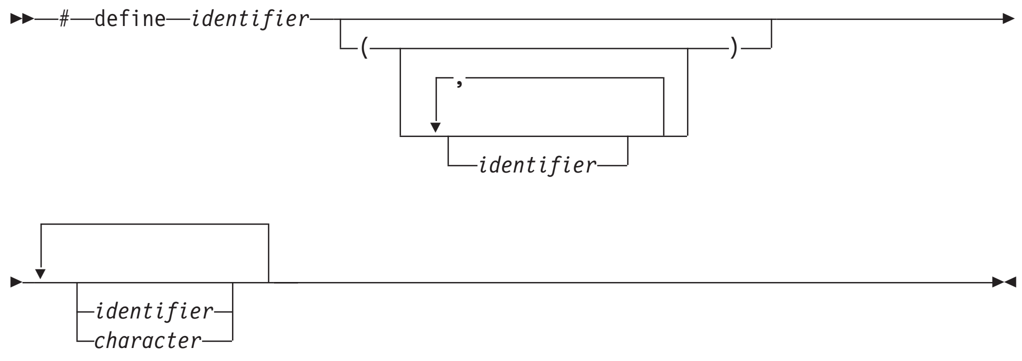
## Preprocessor Directive Format

Preprocessor directives begin with the `#` token that is followed by a preprocessor keyword. The `#` token must appear as the first character that is not white space on a line.

A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the `\` character appears as the last character in the preprocessor line, the preprocessor interprets the `\` and the new-line character as a continuation marker and interprets the following line as a continuation of the current preprocessor line.

## #define

A **preprocessor define directive** directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens. A preprocessor `#define` directive has the form:



The `#define` directive can contain an object-like definition or a function-like definition.

## Object-Like Macro Definition

An **object-like macro definition** replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000`:

```
#define COUNT 1000
```

This definition would cause the preprocessor to change the following statement (if the statement appears after the previous definition):

```
int arry[COUNT];
```

In the output of the preprocessor, the preceding statement would appear as:

```
int arry[1000];
```

The following definition refers to the previously defined identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

## Function-Like Macro Definition

**Function-like macro definition:**

An identifier followed by a parenthesized parameter list and the replacement tokens. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, you should not have more than 31 parameters for a macro.

**Function-like macro call:**

An identifier followed by a parenthesized list of arguments. A comma must separate each argument. Once the preprocessor identifies a function-like macro call, argument substitution takes place. The correspond argument replaces the parameter in the replacement code. Any macro calls that are contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

The following line defines the macro `SUM` as having two parameters `a` and `b` and the replacement tokens `(a + b)`:

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, the preceding statements would appear as:

```
c = (x + y);
c = d * (x + y);
```

**Notes**

A macro call must have the same number of arguments as the corresponding macro definition has parameters.

In the macro call argument list, commas that appear as character constants, in string constants or surrounded by parentheses, do not separate arguments.

The scope of a macro definition begins at the definition and does not end until it encounters a corresponding `#undef` directive. If there is no corresponding `#undef` directive, the scope of the macro definition lasts until the end of the compilation.

A recursive macro is not fully expanded. For example, the definition

```
   #define x(a,b) x(a+1,b+1) + 4
```

would expand

```
   x(20,10)
```

to

```
   x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `static` from subsequent lines in the current file:

```
#define static
```

You can change the definition of a defined identifier or macro with a second preprocessor `#define` directive only if the second preprocessor `#define` directive is preceded by a preprocessor `#undef` directive. See “#undef” on page 140. The `#undef` directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro calls.

### Examples

The following program contains two macro definitions and a macro call that refers to both of the defined macros:

```
#include <stdio.h>
#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;
int main(void)
{
  int x = 2;
  int y = 3;
  PRNT(SQR(x),y);
}
```

After being interpreted by the preprocessor, the preceding program is replaced by code equivalent to the following:

```
int main(void)
{
  int x = 2;
  int y = 3;
  {
     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);
  }
}
```

Processing of this program produces the following output:

```
value 1 = 4
value 2 = 3
```

### Related Information

- “#undef”
- “# Operator” on page 144
- “## Operator” on page 145

## #undef

A **preprocessor undef directive** causes the preprocessor to end the scope of a preprocessor definition. A preprocessor `#undef` directive has the form:

```
►►—#—undef—identifier———————————————————————————————————►◄
```

`#undef` is ignored if the identifier is not currently defined as a macro.

### Examples

The following directives define `BUFFER` and `SQR`:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify the preceding definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers `BUFFER` and `SQR` that follow these `#undef` directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an `#undef` directive, the identifier can be used in a new `#define` directive.

**Related Information**

- “#define” on page 138

## Predefined Macros

The C language provides the following predefined macro names.

**`__DATE__`** A character string literal containing the date when the source file was compiled. The date will be in the form:

```
"Mmm dd yyyy"
```

where:

`Mmm` represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`).

`dd` represents the day. If the day is less than 10, the first `d` will be a blank character.

`yyyy` represents the year.

**`__FILE__`** A character string literal containing the name of the source file.

**`__IFS_IO__`** The integer 1.

**Note:** This macro is defined when SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO) is specified on the CRTCMOD or CRTBNDC command.

**`__IFS64_IO__`** The integer 1.

**Note:** This macro is defined when SYSIFCOPT(*IFS64IO) is specified on the CRTCMOD or CRTBNDC command. When this macro is defined, _LARGE_FILES and _LARGE_FILE_API are also defined in the relevant IBM-supplied header files.

**`__ILEC400__`** A macro that is defined only by the ILE C compiler. You can use this macro in source code that is compiled for several platforms to block off code that is to be compiled only for the AS/400 platform with #ifdef `__ILEC400__` or #if defined (`__ILEC400__`) preprocessor directives.

**`__ILEC400_TGTVRM__`**
A macro defined only by the ILE C compiler, as an integral value that maps to the version/release/modification of the OS/400® that the module or program being compiled is intended to run on. The target release, VxRyMz, translates to an __ILEC400_TGTVRM__

value of xyz, where x, y, and z are integer values. For example, a target release of V3R7M0 will cause the macro to have an integral value of 370.

**__LINE__** An integer that represents the current source line number.

**__OS400__** The integer 1.

**__OS400_TGTVRM__**
A macro defined only by the ILE C compiler, as an integral value that maps to the version/release/modification of the OS/400® that the module or program being compiled is intended to run on. The target release, VxRyMz, translates to an __OS400_TGTVRM__ value of xyz, where x, y, and z are integer values.

**Note:** This macro is also defined by the ILE C++ compiler.

**__POSIX_LOCALE__**
The integer 1.

**Note:** This macro is defined when LOCALETYPE(*LOCALE) is specified on the CRTCMOD or CRTBNDC command line.

**__STDC__** The integer `1`.

**Note:** This macro is undefined if the `langlvl` pragma is set to anything other than `ANSI`.

| **__TERASPACE__** The integer 1.

| **Note:** This macro is defined when TERASPACE(*YES *TSIFC) is
| specified on the CRTCMOD or CRTBNDC command line.

| **__TIME__** A character string literal containing the time when the source file was compiled. The time will be in the form:

```
"hh:mm:ss"
```

where:

`hh` represents the hour.

`mm` represents the minutes.

`ss` represents the seconds.

**__TIMESTAMP__** A character string literal containing the date and time when the source file was last changed.

The date and time will be in the form:

```
"Day Mmm dd hh:mm:ss yyyy"
```

where:

`Day` represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`).

`Mmm` represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`).

dd represents the day. If the day is less than 10, the first d will be a blank character.

hh represents the hour.

mm represents the minutes.

ss represents the seconds.

yyyy represents the year.

**Note:** Other C compilers may not supported this macro. If the macro is supported on other C compilers, the date and time values may be different than those that are shown here.

| **__UCS2__** The integer 1.

| **Note:** This macro is defined when LOCALETYPE(*LOCALEUCS2)
| is specified on the CRTCMOD or CRTBNDC command line.

| **_IS_QSYSINC_INSTALLED**
A macro that is defined only by the ILE C compiler. This macro is defined when the QSYSINC library is successfully added to the product portion of the library list.

**Notes**

The predefined macro names consist of two underscore (__) characters immediately preceding the name, the name in uppercase letters, and two underscore characters immediately following the name.

The value of __LINE__ will change during compilation as the compiler processes subsequent lines of your source program. Also, the value of __FILE__, and __TIMESTAMP__ will change as the compiler processes any #include files that are part of your source program.

You can also change __LINE__ and __FILE__ using the #line preprocessor directive.

**Restrictions**

Predefined macro names cannot be the subject of a #define or #undef preprocessor directive.

**Examples**

The following printf() statements will display the values of the predefined macros __LINE__, __FILE__, __TIME__, and __DATE__ and will print a message indicating the program's conformance to ANSI standards based on __STDC__:

```
#pragma langlvl(ANSI)
#include <stdio.h>
#ifdef __STDC__
#   define CONFORM    "conforms"
#else
#   define CONFORM    "does not conform"
#endif
int main(void)
{
  printf("Line %d of file %s has been executed\n", __LINE__, __FILE__);
  printf("This file was compiled at %s on %s\n", __TIME__, __DATE__);
  printf("This program %s to ANSI standards\n", CONFORM);
}
```

**Related Information**

- "#define" on page 138
- "#undef" on page 140
- "#line" on page 152

## # Operator

The # (single number sign) operator converts a parameter of a function-like macro (see "Function-Like Macro Definition" on page 138) into a character string literal. If macro ABC is defined using the following directive:

```
#define ABC(x)   #x
```

all subsequent calls of the macro ABC would be expanded into a character string literal that contains the argument that are passed to ABC. For example:

| Invocation | Result of Macro Expansion |
|---|---|
| ABC(1) | "1" |
| ABC(Hello there) | "Hello there" |

When you use the # operator in a function-like macro definition, the following rules apply:

1. A parameter in a function-like macro that is preceded by the # operator will be converted into a character string literal containing the argument passed to the macro.
2. White-space characters that appear before or after the argument that is passed to the macro will be deleted.
3. Multiple white-space characters that are imbedded within the argument that is passed to the macro will be replaced by a single space character.
4. If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character will be inserted before the original \ when the macro is expanded.
5. If the argument passed to the macro contains a " (double quotation mark) character, a \ character will be inserted before the " when the macro is expanded.
6. The conversion of an argument into a string literal occurs before macro expansion on that argument.
7. If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

8. If the result of the replacement is not a valid character string literal, the behavior is undefined.

### Examples

The following examples demonstrate the rules that are given in the preceding paragraph.

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

| Invocation | Result of Macro Expansion |
|---|---|
| `STR(\n "\n" '\n')` | `"\n \"\\n\" '\\n'"` |
| `STR(ONE)` | "ONE" |
| `XSTR(ONE)` | "1" |
| `XSTR("hello")` | "\"hello\"" |

### Related Information

- “#define” on page 138
- “#undef” on page 140

## ## Operator

The `##` (double number sign) operator is used to concatenate two tokens in a macro call (text or arguments) given in a macro definition. If a macro `XY` was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for `x` will be concatenated with the first token of the argument for `y`.

For example,

| Invocation | Result of Macro Expansion |
|---|---|
| `XY(1, 2)` | `12` |
| `XY(Green, house)` | `Greenhouse` |

When you use the `##` operator, the following rules apply:

1. The `##` operator cannot be the very first or very last item in the replacement list of a macro definition.
2. The last token of the item that precedes the `##` operator is concatenated with first token of the item that follows the `##` operator.
3. Concatenation takes place before any macros in arguments are expanded.
4. If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
5. If more than one `##` operator or `#` operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

### Example

The following examples demonstrate the rules that are given in the preceding paragraph.

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

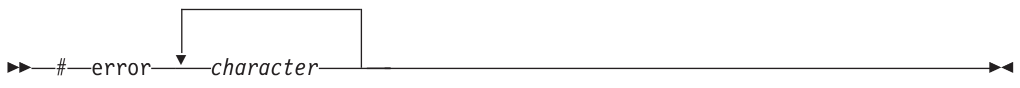
| Invocation | Result of Macro Expansion |
|---|---|
| `ArgArg(var, 1)` | `"var1"` |
| `ArgText(var)` | `"varTEXT"` |
| `TextArg(var)` | `"TEXTvar"` |
| `TextText` | `"TEXTtext"` |
| `ArgArg(Jitter, bug)` | `3` |

### Related Information

- “#define” on page 138

## #error

A **preprocessor error directive** causes the preprocessor to generate an error message and causes the compilation to fail. The `#error` directive has the form:



### Examples

The following directive:

```
#error Error in TESTPGM1 - This section should not be compiled
```

generates the error message: `Error in TESTPGM1 - This section should not be compiled`

### Usage

You can use the `#error` directive as a safety check during compilation. For example, if your program uses **preprocessor conditional compilation directives** (see “Conditional Compilation” on page 149), you can place `#error` directives in the source file to make the compilation fail if a section of the program is reached that should be bypassed.

### Related Information

- “Conditional Compilation” on page 149

## #include

A **preprocessor include directive** causes the preprocessor to replace the directive with the contents of the specified file. A preprocessor `#include` directive has the form:

```
►►──#include──┬─<filename>─┬──────────────────────────────────────────────────►◄
              └─"filename"─┘
```

The following table indicates the search path that is taken by the ILE C compiler for AS/400 source physical files. See the default file names and search paths below.

| Filename | Member | File | Library |
|---|---|---|---|
| *mbr* | *mbr* | default file | default search |
| *file/mbr*[1] | *mbr* | *file* | default search |
| *mbr.file* | *mbr* | *file* | default search |
| *lib/file/mbr* | *mbr* | *file* | *lib* |
| *lib/file(mbr)* | *mbr* | *file* | *lib* |

**Note:**

[1] If the include file format <file/mbr.h> is used, the compiler searches for *mbr* in the file in the library list first. If *mbr* is not found, then the compiler searches for *mbr.h* in the same file in the library list. Only ″h″ or ″H″ are allowed as member name extensions.

If library and file are not specified, the preprocessor uses a specific search path depending on which delimiter surrounds the *filename*. The < > delimiter specifies the name as a system include file. The " " delimiter specifies the name as a user include file.

The following describes the search paths for the #include directive used by the ILE C compiler.

Default File Names When the Library and File are not Named (member name only):

| Include Type | Default File Name |
|---|---|
| **< >** | QCSRC |
| **" "** | The source file of the root source member, where root source member is the library, file, and member determined by the CRTBNDC or CRTCMOD processing of the SRCFILE keyword. |

Default Search Paths When the Filename is not Library Qualified:

| Include Type | Search Path |
|---|---|
| **< >** | Searches the current library list (*LIBL) |
| **" "** | Checks the library containing the root source member; if not found there, the compiler searches the user portion of the library list, using either the filename specified or the file name of the root source member (if no filename is specified); if not found, the compiler searches the library list (*LIBL) using the specified filename. |

Search Paths When the Filename is Library Qualified (lib/file/mbr):

| Include Type | Search Path |
|---|---|
| **< >** | Searches for lib/file/mbr only |
| **" "** | Searches for the member in the library and file named. If not found, searches the user portion of the library list, using the file and member names specified. |

User includes are treated the same as system includes when the CRTCMOD or CRTBNDC command option *SYSINCPATH has been specified.

The preprocessor resolves macros on a `#include` directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >. For example:

```
#define MONTH <july.h>
#include MONTH
```

**Usage**

If there are a number of declarations used by several files, you can place all these definitions in one file and `#include` that file in each file that uses the definitions. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

One of the ways you can combine the use of preprocessor directives is demonstrated in the following example. A `#define` is used to define a macro that represents the name of the C standard I/O header file. A `#include` is then used to make the header file available to the C program.

```
#define  IO_HEADER   <stdio.h>
      .
      .
      .
#include IO_HEADER   /* equivalent to specifying #include <stdio.h> */
      .
      .
      .
```

## Using the #include Directive When Compiling Source in an Integrated File System File

You can use the `SRCSTMF` keyword to specify an Integrated File System file at compile time. The #include processing differs from source physical file processing in that the library list is not searched. An INCLUDE environment variable, if it exists, or the compiler’s default search path described below is used to resolve header files.

The compiler's default include path is `/QIBM/ProdData/ILEC:/QIBM/ProdData/ILEC/include:/QIBM/include`. It is used when the INCLUDE environment variable either has a value of *NULL or does not exist in the job. The compiler's default include path should always be part of the INCLUDE environment variable.

While attempting to open the include file, the compiler uses each directory specified in the environment variable. If, during the search, the open fails for any reason, no message will be given. Instead, the next directory after the colon (:) will be checked.

#include files use the delimiters ″″ or <>.

The algorithm to search for include files is:

```
if file is fully qualified (a slash / starts the name) then
  attempt to open the fully qualified file
else
  if "" is delimiter, check job current directory
  if not found:
    Loop through the list of directories in the INCLUDE environment variable
      or the default include path
    until the file is found
      or the end of the include path is encountered
endif
```

## Conditional Compilation

A **preprocessor conditional compilation directive** causes the preprocessor to conditionally suppress the compilation of portions of source code. Such directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be ignored. The directives are:

- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`

For each `#if`, `#ifdef`, and `#ifndef` directive, there are zero or more `#elif` directives, zero or one `#else` directive, and one matching `#endif` directive. All the matching directives are considered to be at the same nesting level.

You can have nested conditional compilation directives. So if you have the following directives, the first `#else` will be matched with the `#if` directive.

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined */
#   if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                 /* tokens added if MACNAME is defined and TEST >  10 */
#   endif
#else
                 /*  tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

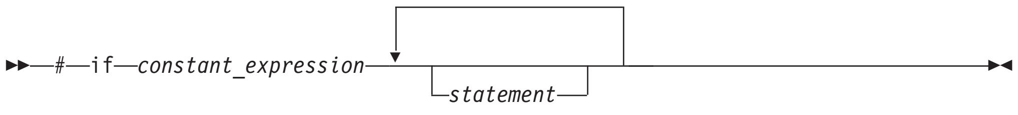
When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a `#else` directive, the block following the `#else` directive is processed. If none of the blocks at that nesting level has been processed and there is no `#else` directive, the entire nesting level is ignored.

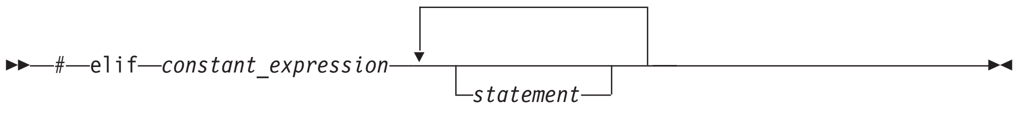
## #if, #elif

The `#if` and `#elif` directives compare the value of the expression to zero. All macros are expanded, any `defined` expressions are processed and all remaining identifiers are replaced with the token `0`.

The preprocessor `#if` directive has the following form:



The preprocessor `#elif` directive has the following form:



The expressions that are tested must be integer constant expressions that follow these rules:

- The expression can contain the `defined` unary operator. The constant expression can contain the unary operator `defined`. This operator can be used only with the preprocessor keyword `#if`. The following expressions evaluate to `1` if the *identifier* is defined in the preprocessor, otherwise to `0`:

  ```
  defined identifier
  defined(identifier)
  ```
- The expression can contain defined macros.
- The preprocessor evaluates the constant expression using `long ints`. The preprocessor does not support `long long ints`.

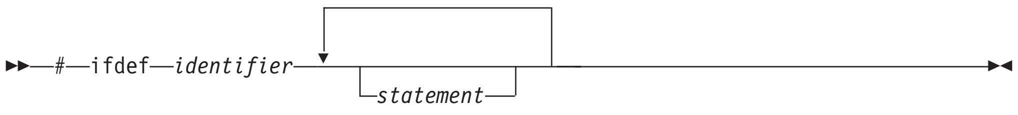
If the constant expression evaluates to a nonzero value, the tokens that immediately follow the condition are passed on to the compiler.

## #ifdef

The `#ifdef` directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor `#ifdef` directive has the following form:



The following example defines `MAX_LEN` to be `75` if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be `50`.

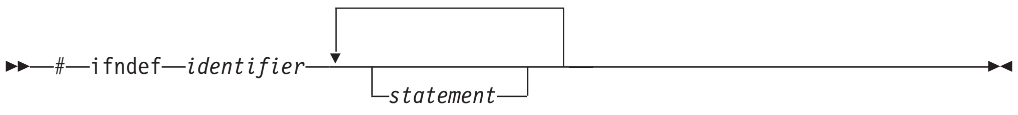
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

## #ifndef

The `#ifndef` directive checks for the existence of macro definitions.

If the identifier that is specified is not defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor `#ifndef` directive has the following form:



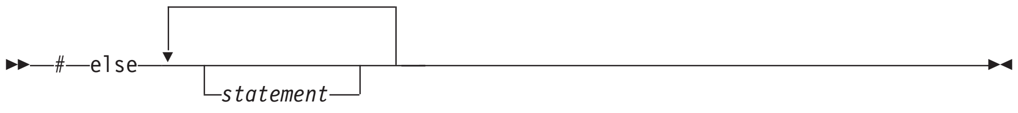
The following example defines `MAX_LEN` to be `50` if `EXTENDED` is not defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be `75`.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

## #else

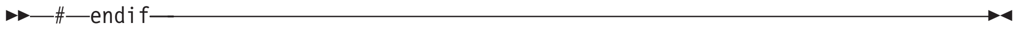
If the condition specified in the `#if`, `#ifdef`, or `#ifndef`directive evaluates to `0`, and the conditional compilation directive contains a preprocessor `#else` directive, the source text located between the preprocessor `#else` directive and the preprocessor `#endif` directive is selected by the preprocessor to be passed on to the compiler.

The preprocessor `#else` directive has the form:



## #endif

The preprocessor `#endif` directive ends the conditional compilation directive. The preprocessor `#endif` directive has the form:



The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

The following program contains preprocessor conditional compilation directives:

```
#include <stdio.h>
int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;
   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#endif
   }
}
```

## #line

A **preprocessor line control directive** causes the compiler to view the line number of the next source line as the specified number. A preprocessor `#line` directive has the form:

```
►►──#──line──decimal_constant──┬──────────────────┬──────────────────────────►◄
                               └─"──filename──"───┘
```

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

The token sequence on a `#line` directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

**Example**

You can use `#line` control directives to make the compiler provide more meaningful error messages. The following program uses `#line` control directives to give each function an easily recognizable line number:

```
#include <stdio.h>
#define LINE200 200
int main(void)
{
   func_1();
   func_2();
}
#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",__LINE__);
}
#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d\n",__LINE__);
}
```

The preceding program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

## # (Null Directive)

The **null directive** performs no action. The null directive consists of a single `#` on a line of its own.

**Example**

In the following example, if `MINVAL` is a defined macro name, no action is performed. If `MINVAL` is not a defined identifier, it is defined 1.

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

## #pragma

The following section describes **#pragma preprocessor directives** available for use with the ILE C compiler.

A pragma is an implementation-defined instruction to the compiler. It has the general form given below, where **character sequence** is a series of characters giving a specific compiler instruction and arguments, if any.

```
                  ┌──────────────────────┐
                  ▼                      │
►►──#──pragma──────character_sequence────┴──────────────────────────────►◄
```

The *character sequence* on a pragma is not subject to macro substitutions. More than one pragma construct can be specified on a single `#pragma` directive.

## argopt

```
►►──#pragma argopt──(──┬─function_name──────────────┬──)──────────────────►◄
                       ├─typedef_of_function_name─┤
                       ├─typedef_of_function_ptr──┤
                       └─function_ptr─────────────┘
```

**Description**

Argument Optimization (argopt) is a pragma which may improve run-time performance. Applied to a bound procedure, optimizations can be achieved by:

- Passing space pointer parameters in to general-purpose registers (GPRs).
- Storing a space pointer returned from a function in to a GPR.

**Parameters**

*function_name* Specifies the name of the function for which optimized procedure parameter passing is to be specified. The function can be either a static function, an externally-defined function, or a function defined in the current compilation unit that will be called from outside the current compilation unit.

*typedef_of_function_name*
Specifies the name of the typedef of the function for which optimized procedure parameter passing is to be specified.

*typedef_of_function_ptr*
Specifies the name of the typedef of the function pointer for which optimized procedure parameter passing is to be specified.

*function_ptr* Specifies the name of the function pointer for which optimized procedure parameter passing is to be specified.

**Notes on Usage**

Specifying #pragma argopt directive does not guarantee that your program will be optimized. Participation in argopt is dependent on the translator.

A function must be declared (prototyped), or defined before it can be named in a #pragma argopt directive.

A structure greater than 8 bytes will not fit in a GPR.

Void pointers will not be optimized since they are not space pointers.

The #pragma argopt cannot be specified for functions which have OS-linkage or built-in linkage (for functions which have a #pragma linkage (function_name, OS) directive or #pragma linkage(function_name, builtin) directive associated with them, and vice versa).

The #pragma argopt will be ignored for functions which are named as handler functions in #pragma exception_handler or #pragma cancel_handler directives, and error handling functions such as `signal()` and `atexit()`. The #pragma argopt directive cannot be applied to functions with a variable argument list.

**#pragma argopt scoping**

The #pragma argopt is placed where the function, the function pointer, typedef of a function pointer or typedef of an function is visible (can be used) within a region of the program code called its scope. #pragma argopt scope is determined its placement in the code. An error will be issued when #pragma argopt is not within the scope of the declaration. The #pragma argopt directive can fall within file, block, or structure scope.

```
#include <stdio.h>

long func3(long y)
{
printf("In func3()\n");
printf("hex=%x,integer=%d\n",y, y);
}
#pragma argopt (func3)           /* file scope of function  */
int main(void)
 {
 int i, a=0;
 typedef long (*func_ptr) (long);
 #pragma argopt (func_ptr)      /* block scope of typedef   */
                                /* of function pointer      */
 struct funcstr
    {
     long (*func_ptr2) (long);
     #pragma argopt (func_ptr2) /* struct scope of function */
                                /* pointer                  */
     };
struct funcstr func_ptr3;
```

*Figure 4. Example using #pragma argopt (Part 1 of 2)*

```
for (i=0; i<99; i++)
 {
  a = i*i;
  if (i == 7)
    {
    func_ptr3.func_ptr2 = func3;
    func_ptr3.func_ptr2( i );
    }
 }
 return i;
}
```

*Figure 4. Example using #pragma argopt (Part 2 of 2)*

## argument

```
►►──#──pragma──argument──(──function_name──┬─,──OS──┬─────────────┬──┬──)──────►◄
                                           │        └─,──nowiden──┘  │
                                           ├─,──VREF──┬─────────────┬┤
                                           │          └─,──nowiden──┘│
                                           └─,──nowiden──────────────┘
```

### Description

Specifies the argument-passing mechanism to be used for the procedure or typedef that is named as the first parameter. If a typedef is named as the first parameter, it must be the typedef of a function.

This pragma simplifies calls to bound procedures in other ILE languages.

### Parameters

| | |
|---|---|
| *function_name* | Specifies the name of the function that is defined in an external program. |
| **OS** | OS indicates that arguments are passed using the same method as the #pragma linkage OS. Non-address arguments are copied to temporary locations, widened (unless nowiden has been specified) and the address of the copy is passed to the called procedure. Arguments that are addresses or pointers are passed directly to the called procedure. |
| **VREF** | VREF indicates that all arguments are copied to temporary locations. The addresses of the temporary locations, rather than the values of the arguments, are passed to the called function. |
| **nowiden** | Specifies that the arguments are not to be widened before they are passed. This parameter can be used by itself without specifying an argument type. For example, #pragma argument (myfunc, nowiden), to indicate that arguments on calls to myfunc are to be passed by value, unwidened. |

### Notes on Usage

A function name specified in the #pragma argument directive should not be defined in the current compilation unit.

Warnings are issued, and the #pragma argument directive is ignored if any of the following occurs:

- The *function_name* in the directive is not the name of a procedure or a typedef of a procedure.
- A typedef named in the directive has been used in the declaration or definition of a procedure before being used in the directive.
- #pragma argument directive has already been specified for this function.

## cancel_handler

```
►►──#──pragma──cancel_handler──(──function_name──┬─,──0──────────┬──)──────────►◄
                                                 └─,──com_area───┘
```

**Description**

Specifies that the function named is to be enabled as a user-defined ILE cancel handler at the point in the code where the #pragma cancel_handler directive is located.

Any cancel handler that is enabled by a #pragma cancel_handler directive is implicitly disabled when the call to the function containing the directive is finished. The call is removed from the call stack, if the handler has not been explicitly disabled by the #pragma disable_handler directive.

**Parameters**

| | |
|---|---|
| *function_name* | Specifies the name of the function to be used as a user-defined ILE cancel handler. |
| *com_area* | Used to pass information to the exception handler. If no *com_area* is required, specify zero as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types: integral, float, double, struct, union, array, enum, pointer, or packed decimal. The com_area should be declared with the volatile qualifier. It cannot be a member of a structure or a union. |

See the *Run-Time Library Reference* for information about <except.h> and the typedef _CNL_Hndlr_Parms_T, a pointer which is passed to the cancel handler.

**Notes on Usage**

This #pragma directive can only occur at a C language statement boundary and inside a function definition.

The compiler issues an error message if any of the following occurs:

- The directive occurs outside a C function body or inside a C statement.
- The handler function is not declared or defined.
- The identifier that is named as the handler function is not a function.
- The *com_area* variable is not declared.
- The *com_area* variable does not have a valid object type.

See the *ILE C for AS/400 Programmer's Guide* for examples and more information about using the #pragma cancel_handler directive.

## chars

```
►►──#──pragma──chars──(──┬─signed───┬──)──────────────────────────────────►◄
                         └─unsigned─┘
```

**Description**

Specifies that the compiler is to treat all `char` objects as `signed` or `unsigned`. This pragma must appear before any C code or directive (except for the #line directive) in a source file.

**Parameters**

| | |
|---|---|
| **`signed`** | All `char` objects are treated as `signed` integers. |
| **`unsigned`** | All `char` objects are treated as `unsigned` integers. |

## checkout

```
►►──#──pragma──checkout──(──┬─suspend─┬──)────────────────────────────────►◄
                            └─resume──┘
```

**Description**

Specifies whether or not the compiler should give informational messages indicating possible programming errors when a CHECKOUT option other than *NONE is specified on the CRTCMOD or CRTBNDC command.

**Parameters**

| | |
|---|---|
| **`suspend`** | Specifies that the compiler suspend informational messages on the CHECKOUT keyword of the CRTCMOD or CRTBNDC command. |
| **`resume`** | Specifies that the compiler resume informational messages on the CHECKOUT keyword of the CRTCMOD or CRTBNDC command. |

**Notes on Usage**

#pragma checkout directives can be nested. This means that a #pragma checkout (suspend) directive will have no effect if a previously specified #pragma checkout (suspend) directive is still in effect. This is also true for the #pragma checkout resume directive.

**Example**

```
/*  Assume CHECKOUT(*PPTRACE) had been specified                       */
#pragma checkout(suspend)  /* No CHECKOUT diagnostics are performed    */
   ...
#pragma checkout(suspend)  /* No effect                                */
   ...
#pragma checkout(resume)   /* No effect                                */
   ...
#pragma checkout(resume)   /* CHECKOUT(*PPTRACE) diagnostics continue  */
```

## comment

```
►►──#──pragma──comment──(──┬─┬─compiler──┬────────────────────────────┬──)───────►◄
                           │ ├─date──────┤                            │
                           │ └─timestamp─┘                            │
                           └─┬─copyright─┬──┬──────────────────────┬──┘
                             └─user──────┘  └─,──"──characters──"──┘
```

**Description**

Emits a comment into the program or service program object. This can be shown by DSPPGM or DSPSRVPGM with DETAIL(*COPYRIGHT). This pragma must appear before any C code or directive (except for the #line directive) in a source file. This pragma has a 256-byte limit.

**Parameters**

Valid settings for the `comment` pragma can be:

| | |
|---|---|
| **`compiler`** | The name and version of the compiler is emitted into the end of the generated program object. |
| **`date`** | The date and time of compilation is emitted into the end of the generated program object. |
| **`timestamp`** | The last modification date and time of the source is emitted into the end of the generated program object. |
| **`copyright`** | The text that is specified by *characters* is placed by the compiler into the generated program object and is loaded into memory when the program is run. |
| **`user`** | The text specified by *characters* is placed by the compiler into the generated object. However, it is not loaded into memory when the program is run. |

**Notes on Usage**

- The copyright and user comment types are virtually the same on the AS/400. One has no advantage over the other.
- The maximum number of characters in the text portion of a #pragma comment(copyright) or #pragma comment(user) directive is 256. (This is an AS/400 restriction.)
- The maximum number of #pragma comment directives that can appear in a single compilation unit is 8. (This is an AS/400 restriction.)

## convert

```
►►──#──pragma──convert──(──ccsid──)──────────────────────────────────────────────►◄
```

**Description**

Specifies the Coded Character Set Identifier (CCSID) to use for converting the string literals from that point onward in a source file during compilation. The conversion continues until the end of the source file or until another #pragma convert directive is specified. Use #pragma convert (0) to disable the previous

#pragma convert directive. The CCSID of the string literals before conversion is the same CCSID as the root source member. CCSIDs 905 and 1026 are not supported. The CCSID can be either EBCDIC or ASCII.

**Parameters**

| | |
|---|---|
| *ccsid* | Specifies the coded character set identifier to use for converting the strings and literals in the source file. The value may range between 0 and 65535. See the *ILE C for AS/400 Run-Time Library Reference* manual for more information about code pages. |

**Notes on Usage**

The run-time library functions that parse format strings (such as `printf()` and `scanf()`) cannot use ASCII format strings. Therefore, all format strings must be in EBCDIC.

String and character constants that are specified in hex, for example (0xC1), are not converted.

Substitution characters will not be used when converting to a target CCSID that does not contain the same symbol set as the source CCSID. The compilation will fail.

If a CCSID with the value 65535 is specified, the CCSID of the root source member is assumed. If the source file CCSID value is 65535, the job CCSID is assumed for the source file. If the file CCSID is 65535 and the job CCSID is not 65535, the job CCSID is assumed for the file CCSID. If the file is 65535 and the job is also 65535, but the system CCSID value is not 65535, the system CCSID value is assumed for the file CCSID. If the file, job and system CCSID values are 65535, CCSID 037 is assumed.

If the LOCALETYPE(*LOCALEUCS2) option is specified on the CRTCMOD or CRTBNDC command line, wide-character literals are not converted. See the ″Using Unicode Support for Wide-Character Literals″ section in the *ILE C for AS/400 Programmer's Guide* for more information.

## descriptor

```
►►──#──pragma──descriptor──(──void──function_name──(──┤ od_specifiers ├──)──)───────►◄
```

**od_specifiers:**

```
      ┌──────────────────────────┐
      ▼                          │
├──────┬─" "──┬────────,──────┴──────────────────────────────────────────────────────┤
       └─void─┘
```

**Description**

An operational descriptor is an optional piece of information that is associated with a function argument. This information is used to describe an argument's attributes, for example, its data type, and length. The #pragma descriptor directive is used to identify functions whose arguments have operational descriptors.

Operational descriptors are useful when passing arguments to functions that are written in other languages that may have a different definition of the data types of the arguments. For example, C defines a string as a contiguous sequence of characters ended by and including the first null character. In another language, a string may be defined as consisting of a length specifier and a character sequence. When passing a string from a C function to a function written in another language, an operational descriptor can be provided with the argument to allow the called function to determine the length and type of the string being passed.

The ILE C compiler generates operational descriptors for arguments that are passed to a function specified in a #pragma descriptor directive. The generated descriptor contains the descriptor type, data type, and length for each argument that is identified as requiring an operational descriptor. The information in an operational descriptor can be retrieved by the called function using the ILE APIs CEEGSI and CEEDOD. See the *System API Reference* for information about the ILE APIs for operational descriptors.

For the operational descriptor to determine the correct string length when passed through a function, the string has to be initialized. If the string is not initialized, the first element will be auto-initialized to NULL '\0'; consequently, the operational descripter will generate a string length of 1.

The ILE C compiler supports operational descriptors for describing strings.

**Note:** A character string in ILE C is defined by using any one of the following:
- char string_name[n]
- char * string_name
- A string literal.

**Parameters**

*function_name* The name of the function whose arguments require operational descriptors.

*od_specifiers* A list of symbols, that consists of either "" or *void*, separated by commas, that specify which of a function's arguments are to have operational descriptors. An *od_specifier* list is similar to the argument list of a function except that an *od_specifier* list for a function can have fewer specifiers than its argument list. If a string operational descriptor is required for an argument, "" should be specified in the equivalent position for the *od_specifier* parameter. If an operational descriptor is not required for an argument then *void* is specified for that parameter in the equivalent position for the *od_specifier* list.

**Notes on Usage**

The compiler issues a warning and ignores the #pragma descriptor directive if any of the following conditions occur:
- The function with the name *function_name* is not prototyped before its #pragma descriptor directive.
- A call to the function with *function_name* occurs before its #pragma descriptor directive.
- The function with the name *function_name* is a static function or a user entry procedure, that is `main()`.

When using operational descriptors consider the following:

- Operational descriptors are only generated for functions that are called by their function name. Functions that are called by function pointer do not have operational descriptors generated.
- If there are fewer *od_specifiers* than function arguments, the remaining *od_specifiers* default to *void*.
- If a function requires a variable number of arguments, the #pragma descriptor directive can specify that operational descriptors are to be generated for the required arguments but not for the variable arguments.
- It is not valid to do pointer arithmetic on a literal or array while it is also used as an argument that requires an operational descriptor, unless explicitly cast to char *. For example, if F is a function that takes as an argument a string, and F requires an operational descriptor for this argument, then the argument on the following call to F is not valid: F(a + 1) where ″a″ is defined as char a[10].

## disable_handler

```
►►──#──pragma──disable_handler─────────────────────────────────────────────►◄
```

**Description**

Disables the handler most recently enabled by either the exception_handler or cancel_handler pragma.

This directive is only needed when a handler has to be explicitly disabled before the end of a function. This is done since all enabled handlers are implicitly disabled at the end of the function in which they are enabled.

**Notes on Usage**

This pragma can only occur at a C language statement boundary and inside a function definition. The compiler issues an error message if the #pragma disable_handler is specified when no handler is currently enabled.

## enumsize

```
►►──#pragma──enumsize──(───────)───────────────────────────────────────────►◄
                          ├─1─┤
                          ├─2─┤
                          └─4─┘
```

**Description**

Specifies the number of bytes the compiler uses to represent enumerations. The pragma affects all subsequent enum definitions until the end of the compilation unit or until another #pragma enumsize directive is encountered. If more than one pragma is used, the most recently encountered pragma is in effect. If the size is not specified in the pragma, the compiler uses the default size, that is, the minimum number of bytes required to represent each value of the enumeration.

If the number of bytes specified in the pragma is less than that required to represent each value of the enumeration, the compiler will issue a warning and set the size of the enumeration type to 4 bytes.

The pragma may appear anywhere that a preprocessor directive is valid.

**Example**

The size of an enumeration variable is determined by the size of its enumeration type. For example:

```
#include <stdio.h>
#pragma enumsize (4)
enum size {Small, Big};   /* size of enum is 4 bytes */
int main(void)
{
#pragma enumsize()
enum length{Long,Short};  /* size of enum is 1 byte  */
enum size s1;
enum length l1;
printf("sizeof s1 = %d;size of l1 = %d\n",
      sizeof(s1),sizeof(l1));
}
```

The output of this example is:

```
sizeof s1 = 4; size of l1 = 1
```

## exception_handler

```
►►──#──pragma──exception_handler──(──┬─function_name─┬──┬─,──0────────┬──────────►
                                     └─label─────────┘  └─,──com_area─┘

►──,──class1──,──class2──┬──────────────────────────────────────┬──)───────────►◄
                         └─,──ctl_action──┬───────────────┬─────┘
                                          └─,──msgid_list─┘
```

**Description**

Enables a user-defined ILE exception handler at the point in the code where the #pragma exception_handler is located.

Any exception handlers enabled by #pragma exception_handler that are not disabled using #pragma disable_handler are implicitly disabled at the end of the function in which they are enabled.

**Parameters**

| | |
|---|---|
| *function* | Specifies the name of the function to be used as a user-defined ILE exception handler. |
| *label* | Specifies the name of the label to be used as a user-defined ILE exception handler. The label must be defined within the function where the #pragma exception_handler is enabled. When the handler gets control, the exception is implicitly handled and control resumes at the label defined by the handler in the invocation containing the #pragma exception_handler directive. The call stack is canceled from the newest call to, but not including, the call containing the #pragma exception_handler directive. The label can |

be placed anywhere in the statement part of the function definition, regardless of the position of the #pragma exception_handler.

*com_area* Used for the communications area. If no *com_area* should be specified, zero is used as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types: integral, float, double, struct, union, array, enum, pointer, or packed decimal. The com_area should be declared with the volatile qualifier. It cannot be a member of a structure or a union.

*class1, class2* Specifies the first four bytes and the last four bytes, respectively, of the exception mask. The <except.h> header file describes the values that you can use for the class masks. It also contains macro definitions for these values. *class1* and *class2* have to evaluate to integer constant expressions after any necessary macro expansions. You can monitor for the valid *class2* values of:

- _C2_MH_ESCAPE
- _C2_MH_STATUS
- _C2_MH_NOTIFY, and
- _C2_FUNCTION_CHECK.

*ctl_action* Specifies an integer constant to indicate what action should take place for this exception handler. If handler is a function, the default value is _CTLA_INVOKE. If handler is a label, the default value is _CTLA_HANDLE. This parameter is optional.

The following are valid exception control actions that are defined in the <except.h> header file:

| **#define name** | **Value** | **Action** |
|---|---|---|
| _CTLA_INVOKE | 1 | This control action will cause the function named on the directive to be invoked and will not handle the exception. If the exception is not explicitly handled, processing will continue. This is valid for functions only. |
| _CTLA_HANDLE | 2 | The exception is handled and messages are logged prior to calling the handler. The exception will no longer be active when handler gets control. Exception processing ends when the exception handler returns. This is valid for functions and labels. |
| _CTLA_HANDLE_NO_MSG | 3 | The exception is handled but messages are not logged prior to calling the handler. The exception will no longer be active when handler gets control and exception messages are not logged. Msg_Ref_Key in the typedef _INTRPT_Hndlr_Parms_T is set to zero. Exception processing ends when the exception handler returns. This is valid for functions and labels. |

| #define name | Value | Action |
|---|---|---|
| _CTLA_IGNORE | 131 | The exception is handled and messages are logged. Control is not passed to the handler function named on the directive and exception will no longer be active. Execution resumes at the instruction immediately following the instruction that caused the exception. This is valid for functions only. |
| _CTLA_IGNORE_NO_MSG | 132 | The exception is handled and messages are not logged. Control is not passed to the handler function named on the directive and exception will no longer be active. Execution resumes at the instruction immediately following the instruction that caused the exception. This is valid for functions only. |

*msgid_list*
Specifies an optional string literal that contains the list of message identifiers. The exception handler will take effect only when an exception occurs whose identifiers matches one of the identifiers on the list of message identifiers. The list is a series of 7-character message identifiers where the first three characters are the message prefix and the last four are the message number. Each message identifier is separated by one or more spaces or commas. This parameter is optional, but if it is specified, *ctl_action* must also be specified.

For the exception handler to get control, the selection criteria for *class1* and *class2* must be satisfied. If the *msgid_list* is specified, the exception must also match at least one of the message identifiers in the list, based on the following criteria:

- The message identifier matches the exception exactly.
- A message identifier, whose two rightmost characters are 00, will match any exception identifier that has the same five leftmost characters. For example, a message identifier of CPF5100 will match any exceptions whose message identifier begins with CPF51.
- A message identifier, whose four rightmost characters are 0000, will match any exception identifier that has the same prefix. For example, a message identifier of CPF0000 will match any exception whose message identifier has the prefix CPF (CPF0000 to CPF9999).
- If *msgid_list* is specified, but the exception that is generated is not one specified in the list, the exception handler will not get control.

**Notes on Usage**

The macro _C1_ALL, defined in the <except.h> header file, can be used as the equivalent of all the valid *class1* exception masks. The macro _C2_ALL, defined in the <except.h> header file, can be used as the equivalent of all four of the valid *class2* exception masks.

You can use the binary OR operator to monitor for different types of messages. For example,

```
#pragma exception_handler(myhandler, my_comarea, 0, _C2_MH_ESCAPE | \
                    _C2_MH_STATUS | _C2_MH_NOTIFY, _CTLA_IGNORE, "MCH0000")
```

will set up an exception monitor for three of the four *class2* exception classes that can be monitored.

The compiler issues an error message if any of the following occurs:

- The directive occurs outside a C function body or inside a C statement.
- The handler that is named is not a declared function or a defined label.
- The *com_area* variable has not been declared or does not have a valid object type.
- Either of the exception class masks is not a valid integral constant
- The *ctl_action* is one of the disallowed values when the handler that is specified is a label (_CTLA_INVOKE, _CTLA_IGNORE, _CTLA_IGNORE_NO_MSG).
- The *msgid_list* is specified, but the *ctl_action* is not.
- A message in the *msgid_list* is not valid. Message prefixes that are not in uppercase are not considered valid.
- The messages in the string are not separated by a blank or comma.
- The string is not enclosed in “ ” or is longer than 4K bytes.

See the *ILE C for AS/400 Programmer’s Guide* for examples and more information about using the #pragma exception_handler directive.

## inline (function)

```
►►──#──pragma──inline──(──function_name──)──────────────────────────────────►◄
```

### Description

The #pragma inline directive specifies that function_name is to be inlined. The pragma can appear anywhere in the source, but must be at file scope. The pragma has no effect if the INLINE(*ON) parameter is not specified on the CRTCMOD or CRTBNDC command. If #pragma inline is specified for a function, the inliner will force the function specified to be inlined on every call. The function will be inlined in both selective (*NOAUTO) and automatic (*AUTO) INLINE mode.

Inlining replaces function calls with the actual code of the function. It reduces function call overhead, and exposes more code to the optimizer, allowing more opportunities for optimization.

### Notes on Usage

- Inlining takes place only if compiler optimization is set to level 30 or higher.
- Directly recursive functions will not be inlined. Indirectly recursive functions will be inlined until direct recursion is encountered.
- Functions calls with variable argument lists will not be inlined if arguments are encountered in the variable portion of the argument list.
- If a function is called through a function pointer, then inlining will not occur.
- The pragma inline directive will be ignored if function_name is not defined in the same compilation unit that contains the pragma.
- A function’s definition will be discarded if the function is static, if it has not had its address taken and if it has been inlined everywhere it is called. This action can decrease the size of the module and program object where the function is used.

See the "Function Call Performance" appendix of the *ILE C for AS/400 Programmer's Guide* for more information on function inlining.

## langlvl

```
►►──#──pragma──langlvl──(──┬─ANSI─────┬──)────────────────────────────►◄
                           ├─SAA──────┤
                           ├─SAAL2────┤
                           └─EXTENDED─┘
```

**Description**

This pragma must appear before any C code or directive (except for the #line directive) in a source file. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level.

**Parameters**

| | |
|---|---|
| **ANSI** | defines the preprocessor variables `__ANSI__` and `__STDC__` and undefines other `langlvl` variables. Product-specific extensions will not be introduced; for example, the prototype for the `hypot()` function will not be declared in `<math.h>`. |
| **SAA** | defines the preprocessor variables `__SAA__` and `__SAA_L2__` and undefines other `langlvl` variables. |
| **SAAL2** | defines the preprocessor variable `__SAA_L2__` and undefines other `langlvl` variables. |
| **EXTENDED** | defines the preprocessor variable `__EXTENDED__` and undefines other `langlvl` variables. The default language level is `EXTENDED`. |

## linkage

```
►►──#──pragma──linkage──(──name1──,──name2──)─────────────────────────►◄
```

**Description**

Specifies that the compiler is to use a different set of linkage conventions. *name1* specifies the name of a function, and *name2* specifies the name of the linkage convention. Variations of specifying *name2* are described in the following sections. This #pragma linkage directive must be placed before any definition of or call to *name1*.

## linkage (function_name, builtin)

```
►►──#──pragma──linkage──(──function_name──,──builtin──┬─────────────┬──)──────►◄
                                                      └─,──nowiden──┘
```

**Description**

Indicates that references to function_name are to be treated as referring to a built-in rather than referring to a bound procedure or dynamically-called program.

**Parameters**

| | |
|---|---|
| *function_name* | Specifies the name of the function affected by this pragma. |
| **builtin** | Specifies that the function named in *function_name* will be treated as a built-in function. The ILE C standard header files contain both the prototypes and the #pragma directives for these built-in (MI) functions that can be used on the AS/400 system. |
| **nowiden** | Specifies that the arguments are not to be widened before they are copied and passed. This parameter is optional. |

**Notes on Usage**

If this pragma specifies a function that is not a built-in function, the compiler issues a warning message, and the pragma is ignored. The compiler treats the function, if the function is called in the compilation unit, as an ordinary C function call.

For more information on bulletins, see the *ILE C/C++ MI Library Reference*.

## linkage (program_name, OS)

```
►►──#──pragma──linkage──(──program_name──,──OS──┬─────────────┬──)─────────────►◄
                                                 └─,──nowiden──┘
```

**Description**

Specifies that the external program, *program_name*, is called using OS/400 calling conventions.

**Parameter**

| | |
|---|---|
| *program_name* | Specifies the name of the external program. This must be specified in uppercase and be less than 11 characters long unless the #pragma map directive is specified to meet OS/400 program naming conventions. If the program name that is specified is too long, it will be truncated. Names greater than 255 characters are truncated to 255 characters during #pragma linkage processing. Specify #pragma map on the 255-character-truncated name. |
| **OS** | Specifies that the external program is called using OS/400 calling conventions. |
| **nowiden** | Specifies that the arguments are not to be widened before they are copied and passed. This parameter is optional. |

**Notes on Usage**

This pragma enables an ILE C program to call other ILE C, EPM, or OPM programs. Arguments on the call are passed according to the following OS/400 argument-passing conventions:

- Non-address arguments are copied to temporary locations, widened (unless `nowiden` has been specified) and the address of the copy is passed to the called program
- Address arguments are passed directly to the called program.

The compiler issues a warning message and ignores the #pragma linkage directive if:

- The program is declared with a return type other than `int` or `void`.
- The #pragma argument directive is already specified for the function.

## linkage (typedef_name, OS)

```
►►──#──pragma──linkage──(──typedef_name──,──OS──┬─────────────┬──)──────────────►◄
                                                └─,──nowiden──┘
```

**Description**

Specifies that the OS linkage convention is to be associated with the typedef name. This typedef can then be used when declaring external functions (programs) that have OS linkage.

**Parameters**

| | |
|---|---|
| *typedef_name* | Specifies the name of the typedef affected by this pragma. |
| **`OS`** | Specifies that the external program is called using OS/400 calling conventions. |
| **`nowiden`** | Specifies that the arguments are not to be widened before they are passed. This parameter is optional. |

**Notes on Usage**

A #pragma linkage OS directive with a typedef name is ignored. A warning is issued by the compiler if the typedef is used in a declaration before it is named in the #pragma linkage OS directive.

If the function named in the pragma is defined in the current compilation unit, the function definition is used and the #pragma linkage directive for that function is ignored. A warning message is issued.

## map

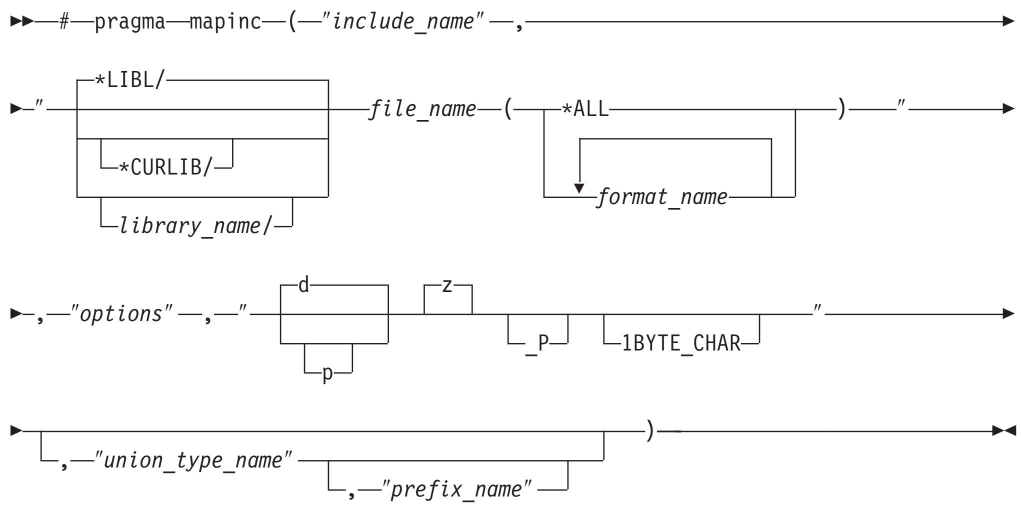
```
►►──#──pragma──map──(──name1──,──"──name2──"──)───────────────────────────────►◄
```

**Description**

Specifies that the compiler is to replace the external symbol (that is used in your C source) *name1* with the external symbol *name2*. Case significance is preserved only for those systems, including ILE C, that support case distinction for external symbols.

The #pragma map directive supports library-qualified external program names. See #pragma linkage(program_name, OS) on “linkage (program_name, OS)” on page 168 for more information.

## mapinc



### Description

Indicates that external AS/400 file descriptions (DDS) are to be included in an ILE C module. The directive identifies the file and DDS record formats, and provides information on the fields to be included. This pragma, along with its associated include directive, causes the ILE C compiler to automatically generate typedefs from the record formats that are specified in the external file descriptions.

### Parameters

| | |
|---|---|
| *include_name* | This is the name that you refer to on the #include directive in the source program (this is only a tag). |
| *library_name* | This is the name of the library that contains the externally described file |
| *file_name* | This is the name of the externally described file. |
| *format_name* | This is a required parameter which indicates the DDS record format that is to be included in your program. You can include more than one record format (format1 format2), or all the formats in a file (*ALL). |
| *options* | The possible *options* are: |

| | |
|---|---|
| **input** | Fields declared as either INPUT or BOTH in the DDS are included in the typedef structure. Response indicators are included in the input structure when the keyword INDARA is not specified in the external file description (DDS source) for device files. |
| **output** | Fields declared as either OUTPUT or BOTH in DDS are included in the typedef structure. Option indicators are included in the output structure when the keyword INDARA is not specified in the external file description (DDS source) for device files. |
| **both** | Fields declared as INPUT, OUTPUT, or BOTH in DDS are included in the typedef structure. Option |

and response indicators are included in both structures when the keyword INDARA is not specified in the external file description (DDS source) for device files.

| | |
|---|---|
| **key** | Fields that are declared as keys in the external file description are included. This option is only valid for database files and DDM files. |
| **indicators** | A separate 99-byte structure for indicators is created when the indicator option is specified. This option is only valid for device files. |
| **lname** | This option allows the use of file names of up to 128 characters in length. If the file name has more than 10 characters then the name will be converted to an associated short name. The short name will be used to extract the external file definition. When the file has a short name of 10 characters or less the name is not converted to an associated short name. Record field names up to 30 characters in length will be generated in the typedefs by the compiler. |
| **lvlchk** | A typedef of an array of struct is generated (type name _LVLCHK_T) for the level check information. A pointer to an object of type _LVLCHK_T is also generated and is initialized with the level check information (format name and level identifier). |
| **nullflds** | If there is at least one null-capable field in the record format of the DDS, a null map typedef is generated containing a character field for every field in the format. With this typedef, the user can specify which fields are to be considered null (set value of each null field to 1, otherwise set to zero). Also, if the key option is used along with option nullflds, and there is at least one null-capable key field in the format, an additional typedef is generated containing a character field for every key field in the format. |

For physical and logical files you can specify `input`, `both`, `key`, `lvlchk`, and `nullflds`. For device files you can specify `input`, `output`, `both`, `indicator`, and `lvlchk`.

The data type can be one or more of the following and must be separated by spaces.

| | |
|---|---|
| **d** | Packed decimal data type. |
| **p** | Packed fields from DDS are declared as character fields. |
| **z** | Zoned fields from DDS are declared as character fields. This is the default because ILE C does not have zoned data type. |
| **_P** | Packed structure is generated. |

**1BYTE_CHAR**
Generates a single byte character field for one byte characters that are defined in DDS.

**" "** Default values of d and z are used.

*union_type_name*
A union definition of the included type definitions is created with the name union_type_name_t. This parameter is optional.

*prefix_name* Specifies the first part of the generated typedef structure name. If the prefix is not specified, the library and file_name are used.

**Notes on Usage**

See "Using Externally Described Files in Your ILE C Program" in the *ILE C for AS/400 Programmer's Guide* for more information about using the #pragma mapinc directive with externally described files.

## margins

```
►►──#──pragma──margins──(──left margin──,──┬─right margin─┬──)──────────────────►◄
                                           └─*────────────┘
```

**Description**

Specifies the left and right margins to be used as the first and last column, respectively, when scanning the records of the source member where the #pragma directive occurs.

The margin setting applies only to the source member in which it is located and has no effect on any source members named on include directives in the member.

**Parameters**

| | |
|---|---|
| *left margin* | Must be a number greater than zero but less than 32 754. The *left margin* should be less than the *right margin*. |
| *right margin* | Must be a number greater than zero but less than 32 754, or an asterisk (*). The *right margin* should be greater than the *left margin*. The compiler scans between the left margin and the right margin. The compiler scans from the left margin specified to the end of the input record, if an asterisk is specified as the value of *right margin*. |

**Notes on Usage**

The #pragma margins directive takes effect on the line following the directive and remains in effect until another #pragma margins or nomargins directive is encountered or the end of the source member is reached.

The #pragma margins directive also overrides the values specified on the MARGINS keyword of the CRTCMOD or CRTBNDC command while the #pragma directive is in effect.

The #pragma margins and sequence directives can be used together. If these two #pragma directives reserve the same columns, the #pragma sequence directive has priority, and the columns are reserved for sequence numbers. For example, if the

#pragma margins directive specifies margins of 1 and 20, and the #pragma sequence directive specifies columns 15 to 25 for sequence numbers, the margins in effect are 1 and 14, and the columns reserved for sequence numbers are 15 to 25.

If the margins specified are not in the supported range or the margins contain non-numeric values, a warning message is issued during compilation and the directive is ignored.

## noargv0

```
►►──#──pragma──noargv0──────────────────────────────────────────────────►◄
```

**Description**

Specifies that the source program does not make use of argv[0]. This pragma can improve performance of applications that have a large number of small C programs, or a small program that is called many times.

**Notes on Usage**

The #pragma noargv0 must appear in the compilation unit where the `main()` function is defined, otherwise it is ignored.

argv[0] will be NULL when the noargv0 pragma directive is in effect. Other arguments in the argument vector will not be affected by this directive. If the #pragma noargv0 directive is not specified, argv[0] will contain the name of the program that is currently running.

## noinline (function)

```
►►──#──pragma──noinline──(──function_name──)────────────────────────────►◄
```

**Description**

Specifies that a function will not be inlined. The settings on the INLINE parameter of the CRTCMOD or CRTBNDC command will be ignored for this function_name.

**Notes on Usage**

The first pragma specified will be the one that is used. If #pragma inline is specified for a function after #pragma noinline has been specified for it, a warning will be issued to indicate that #pragma noinline has already been specified for that function.

The #pragma noinline directive can only occur at file scope.

The pragma will be ignored, and a warning that is issued if it is not found at file scope.

## nomargins

```
►►──#──pragma──nomargins─────────────────────────────────────────────►◄
```

**Description**

Specifies that the entire input record is to be scanned for input.

**Notes on Usage**

The #pragma nomargins directive takes effect on the line following the directive and remains in effect until a #pragma margins directive is encountered or the end of the source member is reached.

The #pragma nomargins directive also overrides the values specified on the MARGINS keyword of the CRTCMOD or CRTBNDC command while the #pragma directive is in effect.

## nosequence

```
►►──#──pragma──nosequence────────────────────────────────────────────►◄
```

**Description**

Specifies that the input record does not contain sequence numbers.

**Notes on Usage**

The #pragma nosequence directive takes effect on the line following the directive and remains in effect until a #pragma sequence directive is encountered or the end of the source member is reached.

The #pragma nosequence directive also overrides the values specified on the SEQCOL keyword of the CRTCMOD or CRTBNDC command while the #pragma directive is in effect.

## nosigtrunc

```
►►──#──pragma──nosigtrunc────────────────────────────────────────────►◄
```

**Description**

Specifies that no exception is generated at run time when overflow occurs with packed decimals in arithmetic operations, assignments, casting, initialization, or function calls. This directive suppresses the signal that is raised in packed decimaloverflow. The #pragma nosigtrunc directive can only occur at filescope. A warning message will be issued if the #pragma nosigtrunc directive is encountered at function, block or function prototype scope, and the directive will be ignored.

**Notes on Usage**

This #pragma directive has file scope and must be placed outside a function definition; otherwise it is ignored. A warning message may still be issued during compilation for some packed decimal operations if overflow is likely to occur. See the "Using Packed Decimal Data in Your ILE CPrograms" chapter of the *ILE C for AS/400 Programmer’s Guide* for more information about packed decimalerrors.

## operational descriptor

See “descriptor” on page 160.

## page

```
►►──#──pragma──page──(──┬───┬──)──────────────────────────────────────►◄
                        └─n─┘
```

**Description**

Skips *n* pages of the generated source listing. If *n* is not specified, the next page is started. The skipped lines are only shown on a hardcopy listing.

## pagesize

**Description**

```
►►──#──pragma──pagesize──(──┬───┬──)──────────────────────────────────►◄
                            └─n─┘
```

Sets the number of lines per page to *n* for the generated source listing. The first seven lines are reserved for compiler generated information such as the page number and product identification. The source code listing starts on the eighth line. The `pagesize` pragma may not affect the option listing page (that is sometimes called the Prolog).

## pointer

```
►►──#──pragma──pointer──(──typedef_name──,──pointer_type──)───────────►◄
```

**Description**

Allows the use of the AS/400 pointer types: space pointer, system pointer, invocation pointer, label pointer, suspend pointer, and open pointer. A variable that is declared with a typedef that is named in the #pragma pointer directive has the pointer type associated with typedef_name in the directive. The <pointer.h> header file contains typedefs and #pragma directives for these pointer types. Including this header file in ILE C source code allows you to use these typedefs directly for declaring pointer variables of these types.

**Parameters**

*pointer_type* which can be one of:

| | |
|---|---|
| **SPCPTR** | Space pointer |

| | |
|---|---|
| **OPENPTR** | Open pointer |
| **SYSPTR** | System pointer |
| **INVPTR** | Invocation pointer |
| **LBLPTR** | Label code pointer |
| **SUSPENDPTR** | Suspend pointer |

**Notes on Usage**

The compiler issues a warning and ignores the #pragma pointer directive if any of the following errors occur:

- The pointer type that is named in the directive is not one of SPCPTR, SYSPTR, INVPTR, LBLPTR, SUSPENDPTR, or OPENPTR.
- The typedef named is not declared before the #pragma pointer directive.
- The identifier that is named as the first parameter of the directive is not a typedef.
- The typedef named is not a typedef of a void pointer.
- The typedef named is used in a declaration before the #pragma pointer directive.

The typedef named must be defined at file scope.

See the *ILE C for AS/400 Programmer's Guide* for more information about using AS/400 pointers.

## sequence

```
►►──#──pragma──sequence──(──left_column──,──┬─right_column─┬──)──────────────►◄
                                            └─*────────────┘
```

**Description**

Specifies the columns of the input record that are to contain sequence numbers. The column setting applies only to the source setting in which it is located and has no effect on any source members named on include directives in the member.

**Parameters**

| | |
|---|---|
| *left column* | Must be greater than zero but less than 32 754. The *left column* should be less than the *right column*. |
| *right column* | Must be greater than zero but less than 32 754. The *right column* should be greater than or equal to the *left column*. An asterisk (*) that is specified as the *right column* value indicates that sequence numbers are contained between *left column* and the end of the input record. |

**Notes on Usage**

The #pragma sequence directive takes effect on the line following the directive. It remains in effect until another #pragma sequence or nosequence directive is encountered or the end of the source member is reached.

The #pragma sequence directive also overrides the values specified on the SEQCOL keyword of the CRTCMOD or CRTBNDC command while the #pragma directive is in effect.

The #pragma margins and sequence directives can be used together. If these two #pragma directives reserve the same columns, the #pragma sequence directive has priority, and the columns are reserved for sequence numbers. For example, if the #pragma margins directive specifies margins of 1 and 20 and the #pragma sequence directive specifies columns 15 to 25 for sequence numbers, the margins in effect are 1 and 14, and the columns reserved for sequence numbers are 15 to 25.

If the margins specified are not in the supported range or the margins contain non-numeric values, a warning message is issued during compilation and the directive is ignored.

## skip

```
►►──#──pragma──skip──(──┬───┬──)──────────────────────────────────►◄
                        └─n─┘
```

**Description**

Skips the next *n* lines of the generated source listing. The value of *n* must be a positive integer less than 255. If *n* is omitted, one line is skipped. The skipped lines are only shown on a hardcopy listing.

## strings

```
►►──#──pragma──strings──(──┬─readonly──┬──)───────────────────────►◄
                           └─writeable─┘
```

**Description**

Specifies that the compiler may place strings into read-only memory or must place strings into writeable memory. Strings are writeable by default. This pragma must appear before any C code in a file.

**Note:** This pragma will override the *STRDONLY option on the CRTBNDC or CRTCMOD command.

## subtitle

```
►►──#──pragma──subtitle──(──"──subtitle──"──)─────────────────────►◄
```

**Description**

Places the text that is specified by *subtitle* on all subsequent pages of the generated source listing.

## title

```
►►──#──pragma──title──(──"──title──"──)──────────────────────────────────────►◄
```

### Description

Places the text that is specified by *title* on all subsequent pages of the generated source listing.

### Example

```
#pragma langlvl(ANSI)
#pragma title("pragma example")
#pragma pagesize(55)
#pragma map(ABC, "A$$BC@")
```

# Chapter 9. I/O Considerations

This chapter provides information on:

- Data Management Operations on Record Files
- Data Management Operations on Stream Files
- C Streams and File Types
- DDS-to-ILE C Data Type Mappings

## Data Management Operations on Record Files

See the *Data Management* manual for information on the data management operations and ILE C functions available for record files.

## Data Management Operations on Stream Files

See the *Data Management* manual for information on the data management operations and ILE C functions available for stream files.

To use stream files (type=record) with record I/O functions you must cast the FILE pointer to an RFILE pointer.

## C Streams and File Types

The following table summarizes which file types are supported as streams.

*Table 18. Processing C Stream and File Types*

| Stream | Database | Diskette | Tape | Printer | Display | ICF | DDM | Save |
|---|---|---|---|---|---|---|---|---|
| TEXT | Yes | No | No | Yes | No | No | Yes | No |
| BINARY: Character at a time | Yes | No | No | Yes | No | No | Yes | No |
| BINARY: Record at a time | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

## DDS-to-C Data Type Mapping

The following table shows DDS data types and the corresponding ILE C declarations that are used to map fields from externally described files to your ILE C program. The ILE C compiler creates fields in structure definitions based on the DDS data types in the externally described file.

See the *DDS Reference* for more information.

*Table 19. DDS-to-ILE C Data Type Mappings*

| DDS Data Type | Length | Decimal Position | ILE C Declaration |
|---|---|---|---|
| Indicator | 1 | 0 | char INxx_INyy[n]; for unused indicators xx through yy char INxx; for used indicator xx |
| A - alphanumeric | 1-32766 | none | char field[n]; (where n = 1 to 32766) |

*Table 19. DDS-to-ILE C Data Type Mappings (continued)*

| DDS Data Type | Length | Decimal Position | ILE C Declaration |
|---|---|---|---|
| A - alphanumeric variable length VARLEN keyword | 1-32740 | none | `_Packed struct { short len;` `char data[n];` `} field;` `where n is the maximum length of 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 | _Packed struct { short len; char data[n]; } field; where n is the maximum length of field |
| G - graphic variable length VARLEN keyword | 4-1000 | none | _Packed struct { short len; wchar_t data[n]; } field; (where n = 4 to 1000) |
| P - packed decimal | 1-31 | 0-31 | decimal (n,p) where n is length and p is decimal position on option 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) |
| J - DBCS only variable length VARLEN keyword | 4-32740 | none | _Packed struct { short len; char data[n]; } field; (where n = 4 to 32740 and n is an even number) |
| E - DBCS either variable length VARLEN keyword | 4-32740 | none | _Packed struct { short len; char data[n]; } field; (where n = 4 to 32740 and n is an even number) |
| O - DBCS open variable length VARLEN keyword | 4-32740 | none | `_Packed struct { short len;` `char data[n];` `} field;` `(where n = 4 to 32740)` |
| T - time | 8 | none | char field[8]; |
| L - date | 6, 8, or 10 | none | char field[n]; (where n = 6, 8 or 10) |
| Z - time stamp | 26 | none | char field[26]; |

**Note:** 1 The C declaration (float or double) is based on what is specified in the FLTPCN (floating-point precision) keyword in the DDS: *SINGLE (default) is float, *DOUBLE is double.

See the *DDS Reference* for more information on using the new database data types or variable length fields.

# Appendix. AS/400 Control Characters

The following table identifies the internal hexadecimal representation of the AS/400 control sequences that are used by the ILE C compiler and library.

*Table 20. Internal Hexadecimal Representation*

| **Print representation** | **Internal representation** |
|---|---|
| NUL (null) | 0x00 |
| SOH (start of heading) | 0x01 |
| STX (start of text) | 0x02 |
| ETX (end of text) | 0x03 |
| SEL (select) | 0x04 |
| HT (horizontal tab) | 0x05 |
| RNL (required new line) | 0x06 |
| DEL (delete) | 0x07 |
| GE (graphic escape) | 0x08 |
| SPS (superscript) | 0x09 |
| RPT (repeat) | 0x0a |
| VT (vertical tab) | 0x0b |
| FF (form feed) | 0x0c |
| CR (carriage return) | 0x0d |
| SO (shift out) | 0x0e |
| SI (shift in) | 0x0f |
| DLE (data link escape) | 0x10 |
| DC1 (device control 1) | 0x11 |
| DC2 (device control 2) | 0x12 |
| DC3 (device control 3) | 0x13 |
| RES/ENP (restore or enable presentation) | 0x14 |
| NL (new line) | 0x15 |
| BS (backspace) | 0x16 |
| POC (program-operator communication) | 0x17 |
| CAN (cancel) | 0x18 |
| EM (end of medium) | 0x19 |
| UBS (unit backspace) | 0x1a |
| CU1 (customer use 1) | 0x1b |
| IFS (interchange file separator) | 0x1c |
| IGS (interchange group separator) | 0x1d |
| IRS (interchange record separator) | 0x1e |
| IUS/ITB (interchange unit separator or intermediate transmission block) | 0x1f |
| DS (digit select) | 0x20 |
| SOS (start of significance) | 0x21 |
| FS (field separator) | 0x22 |

*Table 20. Internal Hexadecimal Representation (continued)*

| **Print representation** | **Internal representation** |
|---|---|
| WUS (word underscore) | 0x23 |
| BYP/INP (bypass or inhibit presentation) | 0x24 |
| LF (line feed) | 0x25 |
| ETB (end of transmission block) | 0x26 |
| ESC (escape) | 0x27 |
| SA (set attributes) | 0x28 |
| SM/SW (set mode or switch) | 0x2a |
| CSP (control sequence prefix) | 0x2b |
| MFA (modify field attribute) | 0x2c |
| ENQ (enquiry) | 0x2d |
| ACK (acknowledge) | 0x2e |
| BEL (bell) | 0x2f |
| SYN (synchronous idle) | 0x32 |
| IR (index return) | 0x33 |
| PP (presentation position) | 0x34 |
| TRN | 0x35 |
| NBS (numeric backspace) | 0x36 |
| EOT (end of transmission) | 0x37 |
| SBS (subscript) | 0x38 |
| IT (indent tab) | 0x39 |
| RFF (required form feed) | 0x3a |
| CU3 (customer use 3) | 0x3b |
| DC4 (device control 4) | 0x3c |
| NAK (negative acknowledge) | 0x3d |
| SUB (substitute) | 0x3f |
| (blank character) | 0x40 |

# Bibliography

For additional information about topics related to ILE C programming on the AS/400 system, refer to the following IBM AS/400 publications:

- *ADTS/400: Application Development Manager User's Guide*, SC09-2133-01, describes creating and managing projects defined for the Application Development Manager/400 feature, as well as using the program to develop applications.
- *ADTS/400: Programming Development Manager*, SC09-1771-00, provides information about using the Application Development ToolSet/400 programming development manager (PDM) to work with lists of libraries, objects, members, and user-defined options to easily do such operations as copy, delete, and rename. Contains activities and reference material to help the user learn PDM. The most commonly used operations and function keys are explained in detail using examples.
- *ADTS for AS/400: Source Entry Utility*, SC09-2605-00, provides information about using the Application Development ToolSet/400 source entry utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *Application Display Programming*, SC41-5715-00, provides information about:
  - Using DDS to create and maintain displays for applications;
  - Creating and working with display files on the system;
  - Creating online help information;
  - Using UIM to define panels and dialogs for an application;
  - Using panel groups, records, or documents
- *Backup and Recovery*, SC41-5304-03, provides information about setting up and managing the following:
  - Journaling, access path protection, and commitment control
  - User auxiliary storage pools (ASPs)
  - Disk protection (device parity, mirrored, and checksum)

  Provides performance information about backup media and save/restore operations. Also includes advanced backup and recovery topics, such as using save-while-active support, saving and restoring to a different release, and programming tips and techniques.
- *CICS Family: Application Programming Guide*, SC41-5454-00, provides information on application programming for CICS/400®. It includes guidance and reference information on the CICS application programming interface and system programming interface commands, and gives general information about developing new applications and migrating existing applications from other CICS platforms.
- *ILE C/C++ MI Library Reference*, SC09-2418-00, provides information on Machine Interface instructions available in the ILE C compiler that provide system-level programming capabilities.
- *CL Programming*, SC41-5721-02, provides a wide-ranging discussion of AS/400 programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- *CL Reference (Abridged)*, SC41-5722-03, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- *Communications Management*, SC41-5406-02, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.

- *Data Management*, SC41-5710-00, provides information about using files in application programs. Includes information on the following topics:
  - Fundamental structure and concepts of data management support on the system
  - Overrides and file redirection (temporarily making changes to files when an application program is run)
  - Copying files by using system commands to copy data from one place to another
  - Tailoring a system using double-byte data
- *DB2 for AS/400 Database Programming*, SC41-5701-02, provides a detailed discussion of the AS/400 database organization, including information on how to create, describe, and update database files on the system. Also describes how to define files to the system using OS/400 data description specifications (DDS) keywords.
- *DB2 for AS/400 SQL Programming*, SC41-5611-02, provides information about how to use DB2® Query Manager and SQL Development kit licensed program. Shows how to access data in a database library and prepare, run, and test an application program that contains embedded SQL statements. Contains examples of SQL/400® statements and a description of the interactive SQL function. Describes common concepts and rules for using SQL/400 statements in ILE COBOL, PL/I, ILE C, FORTRAN/400®, ILE RPG/400, and REXX.
- *DB2 for AS/400 SQL Reference*, SC41-5612-02, provides information about how to use Structured Query Language/400 DB2 statements and gives details about the proper use of the statements. Examples of statements include syntax diagrams, parameters, and definitions. A list of SQL limits and a description of the SQL communication area (SQLCA) and SQL descriptor area (SQLDA) are also provided.
- *DDS Reference*, SC41-5712-01, provides detailed descriptions for coding the data description specifications (DDS) for file that can be described externally. These files are physical, logical, display, print, and intersystem communication function (ICF) files.
- *Distributed Data Management*, SC41-5307-00, provides information about remote file processing. It describes how to define a remote file to OS/400 distributed data management (DDM), how to create a DDM file, what file utilities are supported through DDM, and the requirements of OS/400 DDM as related to other systems.
- *Experience RPG IV Multimedia Tutorial*, SK2T-2700, is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG code examples are shipped with the tutorial and run directly on the AS/400.
- *GDDM Programming Guide*, SC41-0536-00, provides information about using OS/400 graphical data display manager (GDDM®) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- *GDDM Reference*, SC41-3718-00, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.
- *ICF Programming*, SC41-5442-00, provides information needed to write application programs that use AS/400 communications and the OS/400 intersystem communications function (OS/400-ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *IDDU Use*, SC41-5704-00, describes how to use the AS/400 interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
  - An introduction to computer file and data definition concepts
  - An introduction to the use of IDDU to describe the data used in queries and documents
  - Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
  - Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- *ILE C for AS/400 Programmer's Guide*, SC09-2712-01, provides information on how to

develop applications using the ILE C language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also described. An appendix includes information on migrating source code from EPM C or System C to ILE C.

- *ILE C for AS/400 Run-Time Library Reference*, SC09-2715-00, provides reference information about ILE C library functions, including Standard C library functions and ILE C library extensions. Examples are provided and considerations for programming are also discussed.
- *ILE COBOL for AS/400 Programmer's Guide*, SC09-2540-01, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the AS/400 system. It provides programming information on how to call other ILE COBOL and non-ILE COBOL programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- *ILE COBOL for AS/400 Reference*, SC09-2539-01, provides a description of the ILE COBOL programming language. It provides information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides a description of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- *ILE COBOL for AS/400 Reference Summary*, SX09-1317-01, provides quick reference information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides syntax diagrams of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- *ILE Concepts*, SC41-5606-03, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE RPG for AS/400 Programmer's Guide*, SC09-2507-02, provides information about the ILE RPG programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the AS/400 system. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use AS/400 files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- *ILE RPG for AS/400 Reference*, SC09-2508-02, provides information about the ILE RPG programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG IV specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.
- *ILE RPG for AS/400 Reference Summary*, SX09-1315-01, provides information about the RPG III and RPG IV programming language. This manual contains tables and lists for all specifications and operations in both languages. A key is provided to map RPG III specifications and operations to RPG IV specifications and operations.
- *Local Device Configuration*, SC41-5121-00, provides information about configuring local devices on the AS/400 system. This includes information on how to configure the following:
  - Local work station controllers (including twinaxial controllers)
  - Tape controllers
  - Locally attached devices (including twinaxial devices)
- *Machine Interface Functional Reference*, SC41-5810-00, describes the machine interface instruction set. Describes the functions that can be performed by each instruction and also the necessary information to code each instruction.
- *Printer Device Programming*, SC41-5713-03, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the AS/400 system, printer file and print spooling support for printing operations, and

printer connectivity. Includes considerations for using personal computers, other printing functions such as Business Graphics Utility (BGU), advanced function printing (AFP), and examples of working with the AS/400 system printing elements such as how to move spooled output files from one output queue to a different output queue. Also includes an appendix of control language (CL) commands used to manage printing workload. Fonts available for use with the AS/400 system are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.

- *REXX/400 Programmer's Guide*, SC41-5728-00, provides a wide-ranging discussion of programming with REXX on the AS/400 system. Its primary purpose is to provide useful programming information and examples to those who are new to Procedures Language 400/REXX and to provide those who have used REXX in other computing environments with information about the Procedures Language 400/REXX implementation.
- *ILE RPG for AS/400 Programmer's Guide*, SC09-2507-02, provides information needed to design, code, compile, and test RPG programs on the AS/400 system. The manual provides information on data structures, data formats, file processing, multiple file processing, the automatic report function, RPG command statements, testing and debugging functions, application design techniques, problem analysis, and compiler service information. The differences between the RPG for AS/400 compiler, the System/38® environment RPG III compiler, and the System/36®-compatible RPG II compiler are also discussed.
- *Security - Basic*, SC41-5301-00, explains why security is necessary, defines major concepts, and provides information on planning, implementing, and monitoring basic security on the AS/400 system.
- *Security - Reference*, SC41-5302-03, tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- *Local Device Configuration*, SC41-5121-00, provides step-by-step procedures for initial installation, installing licensed programs, program temporary fixes (PTFs), and secondary languages from IBM. This manual is also for users who already have an AS/400 system with an installed release and want to install a new release.
- *System API Programming*, SC41-5800-00, provides information for the experienced application and system programmers who want to use the OS/400 application programming interfaces (APIs). Provides getting started and examples to help the programmer use APIs.
- *System API Reference*, SC41-5801-03, provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
  - Dynamic Screen Manager
  - Files (database, spooled, hierarchical)
  - Message handling
  - National language support
  - Network management
  - Objects
  - Problem management
  - Registration facility
  - Security
  - Software products
  - Source debug
  - UNIX-type
  - User-defined communications
  - User interface
  - Work management

  Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.
- *System Operation*, SC41-4203-00, provides information about handling messages, working with jobs and printer output, devices communications, working with support functions, cleaning up your system, and so on.
- *Basic System Operation, Administration, and Problem Handling*, SC41-5206-03, provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.
- *Tape and Diskette Device Programming*, SC41-5716-01, provides information to help users develop and support programs that use tape and diskette drives for I/O. Includes

information on device files and descriptions for
tape and diskette devices.

# Index

Program Number:  5769-CX2

Printed in U.S.A.

SC09-2711-01