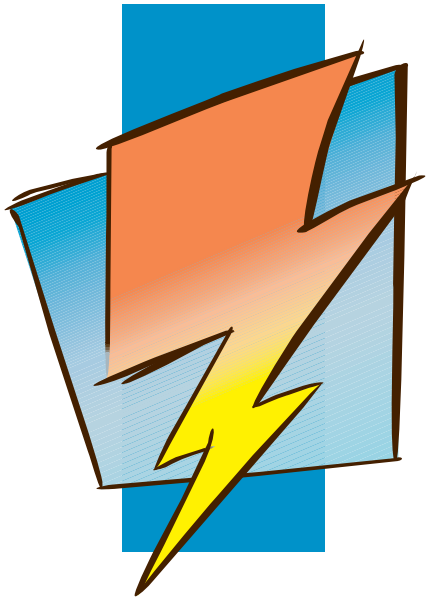
Open Watcom C

Language Reference



Version 2.0

Open Watcom

Notice of Copyright

For more information please visit http://www.openwatcom.org/

Preface

This book describes the C programming language as implemented by the Open Watcom C^{16} and C^{32} compilers for 80x86-based processors. Open Watcom C^{16} and C^{32} are implementations of ANSI/ISO 9899:1990 Programming Language C. The standard was developed by the ANSI X3J11 Technical Committee on the C Programming Language. In addition to the full C language standard, the compiler supports numerous extensions for the 80x86 environment.

This book is intended to be a reference manual and hence a precise description of the C language. It also attempts to remain readable by ordinary humans. When new concepts are introduced, examples are given to provide clarity.

Since C is a programming language that is supposed to aid programmers trying to write portable programs, this book points out those areas of the language that may vary from one system to another. Where possible, the probable behavior of other C compilers is mentioned.

February, 2008.

Trademarks

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction

1 Introduction to C

1.1 History

The C programming language was developed by Dennis Ritchie in 1972 for the UNIX operating system. Over the years, the language has appeared on many other systems, satisfying a need of programmers who want to be able to develop applications that can run in many different environments.

Because the C language was never formally defined, each implementation interpreted the behavior of the language in slightly different ways, and also introduced their own extensions. As a result, the goal of true software portability was not achieved.

In 1982, the American National Standards Committee formed the X3J11 Technical Committee on the C Programming Language, whose purpose was to formally define the C language and its library functions, and to describe its interaction with the execution environment. The C Programming Language standard was completed in 1989.

The Open Watcom C^{16} and C^{32} compiler has evolved from 8086 code generation technology developed and refined at WATCOM International and the University of Waterloo since 1980. The first Open Watcom C^{16} compiler was released in 1988. The first Open Watcom C^{32} compiler was released in 1989.

1.2 Uses

C is sometimes called a "low-level" language, referring to the fact that C programmers tend to think in terms of bits, bytes, addresses and other concepts fundamental to assembly-language programming.

But C is also a "broad spectrum" language. In addition to accessing the basic components of the computer, it also provides features common to many "high-level" languages. Structured program control, data structures and modular program design are recent additions to some high-level languages, but have been part of the C language since its inception.

C gives the programmer the ability to write applications at a level just above the assembly language level, without having to know the assembly language of the machine. Language compilers provided this ability in the past, but the application was often quite "fat", because the code produced by the compiler was never as good as could be written by a good assembly language programmer. But with modern code generation techniques it is often difficult, if not impossible, to distinguish an assembly language program written by a human from the same program generated by a C compiler (based on code size). In fact, some compilers now generate better code than all but the best assembly language programmers.

So, what can C be used for? It can be used to write virtually anything, the same way that assembly language can be used. But other programming languages continue to be used for specific programming applications at which they excel.

C tends to be used for "systems programming", a term that refers to the writing of operating systems, programming languages and other software tools that don’t fall into the class of "applications

programming". A classic example is the UNIX operating system, developed by Bell Laboratories. It is written almost entirely in C and is one of the most portable operating systems available.

C is also used for writing large programs that require more efficiency than the average application. Typical examples are interpreters and compilers for programming languages.

Another area where C is commonly used is large-scale application programs, such as databases, spreadsheets, word processors and so on. These require a high degree of efficiency and compactness, since they are often basic to an individual's or company's computing needs, and therefore consume a lot of computer resources.

It seems that C is used extensively for commercially available products, but C can also be used for any application that just requires more efficiency. For example, a large transaction processing system may be written in COBOL, but to squeeze the last bit of speed out of the system, it may be desirable to rewrite it in C. That application could certainly be written in assembly language, but many programmers now prefer to avoid programming at such a low level, when a C compiler can generate code that is just as efficient.

Finally, of course, a major reason for writing a program in C is that it will run with little or no modification on any system with a C compiler. In the past, with the proliferation of C compilers and no standard to guide their design, it was much more difficult. Today, with the appearance of the ISO standard for the C programming language, a program written entirely in a conforming C implementation should be transportable to a new compiler with relatively little work. Of course, issues like file names, memory layout and command line parameter syntax will vary from one system to another, but a properly designed C application will isolate these parts of the code in "system-dependent" files, which can be changed for each system. (Refer to "Writing Portable Programs".)

1.3 Advantages

C has a number of major advantages over other programming languages.

- Most systems provide a C compiler.

Vendors of computer systems realize that the success of a system is dependent upon the availability of software for that system. With the large body of C-based programs in existence, most vendors provide a C compiler in order to encourage the transporting of some of these programs to their system. For systems that don't provide a C compiler, independent companies may develop a compiler.

With the development of the ISO/ANSI C standard, the trend towards universal availability of C compilers will probably accelerate.

- C programs can be transported easily to other computers and operating systems.

Many programming languages claim transportability. FORTRAN, COBOL and Pascal programs all have standards describing them, so a program written entirely within the standard definition of the language will likely be portable. The same is true of C. However, few languages can match portability with the other advantages of C, including efficiency of generated code and the ability to work close to the machine level.

- Programs written in C are very efficient in both execution speed and code size.

Few languages can match C in efficiency. A good assembly language programmer may be able to produce code better than a C compiler, but he/she will have to spend much more time in the development of the application, because assembly language programming lends itself more easily to errors. Compilers for

other languages may produce efficient code for applications within their scope, but few produce efficient code for *all* applications.

- C programs can get close to the hardware, controlling devices directly if necessary.

Most programs do not need this ability, but if necessary, the program can access particular features of the computer. For example, a fixed memory location may exist that contains a certain value of use to the program. It is easy to access it from C, but not from many other languages. (Of course, if the program is designed to be portable, this section of code will be isolated and clearly marked as depending on the operating system.)

- C programs are easy to maintain.

Assembly language code is difficult to maintain owing to the very low level of programming (registers, addressing modes, branching). C programs provide comparable functionality, but at a higher level. The programmer still thinks in terms of machine capabilities, but without having to know the exact operation of the hardware, leaving the programmer free to concentrate on program design rather than the intimate details of coding on that particular machine.

- C programs are easy to understand.

"Easy" is, of course, a relative term. C programs are definitely easier to understand than the equivalent assembly language program. Another programming language may be easier to understand for a particular kind of application, but in general C is a good choice.

- All of the above advantages apply regardless of the application or the hardware or operating system on which it is running.

This is the biggest advantage. Because C programs are portable, and C is not suited only to a certain class of applications, it is often the best choice for developing an application.

1.4 How to Use This Book

This book is a description of the C programming language as implemented by the Open Watcom C[16] and C[32] compilers for the 80x86 family of processors. It is intended to be an easy-to-read description of the C language. The ISO C standard is the last word on details about the language, but it describes the language in terms that must be interpreted for each implementation of a C compiler.

This book attempts to describe the C language in terms of general behavior, and the specific behavior of the C compiler when the standard describes the behavior as *implementation-defined.*

> Areas that are shaded describe the interpretation of the behavior that the Open Watcom C[16] and C[32] compilers follow.

Programmers who are writing a program that will be ported to other systems should pay particular attention when using these features, since other compilers may behave in other ways. As much as possible, an attempt is made to describe other likely behaviors.

This book does not describe any of the library functions that a C program might use to interact with its environment. In particular, input and output is not described in this manual. The C language does not contain any I/O capabilities. The Open Watcom C Library Reference manual describes all of the library functions, including those used for input and output.

A glossary is included in the appendix, and describes all terms used in the book.

Language Reference

2 Notation

The C programming language contains many useful features, each of which has a number of optional parts. The ISO C standard describes the language in very precise terms, often giving syntax diagrams to describe the features.

This book attempts to describe the C language in more friendly terms. Where possible, features are described using ordinary English. Jargon is avoided, although by necessity, new terminology is introduced throughout the book. A glossary is provided at the end of the book to describe any terms that are used.

Where the variety of features would create excessive amounts of text, simple syntax diagrams are used. It is hoped that these are mostly self-explanatory. However, a brief explanation of the notation used is offered here:

1. Required keywords are in normal lettering style (for example, `enum`).

2. Terms that describe a class of object that replace the term are in italics (for example, *identifier*).

3. When two or more optional forms are available, they are shown as follows:

   form 1
   or
   form 2

4. Any other symbol that appears is required, unless otherwise noted.

The following example is for an enumerated type:

`enum` *identifier*
or
`enum` { *enumeration-constant-list* }
or
`enum` *identifier* { *enumeration-constant-list* }

An enumerated type has three forms:

1. The required keyword `enum` followed by an identifier that names the type. The identifier is chosen by the programmer.

2. The required keyword `enum` followed by a brace-enclosed list of enumeration constants. The braces are required, and *enumeration-constant-list* is described elsewhere.

3. The required keyword `enum` followed by an identifier and a brace-enclosed list of enumeration constants. As with the previous two forms, the identifier may be chosen by the programmer, the braces are required and *enumeration-constant-list* is described elsewhere.

3 Basic Language Elements

The following topics are discussed:

- Character Sets
- Keywords
- Identifiers
- Comments

3.1 Character Sets

The *source character set* contains the characters used during the translation of the C source file into object code. The *execution character set* contains the characters used during the execution of the C program. In most cases, these two character sets are the same, since the program is compiled and executed on the same machine. However, C is sometimes used to *cross-compile,* whereby the compilation of the program occurs on one machine, but the compiler generates code for some other machine. If the two machines have different character sets (say EBCDIC and ASCII), then the compiler will, where appropriate, map characters from the source character set to the execution character set. This mapping is implementation-defined, but generally maps the visual representation of the character.

Regardless of which C compiler is used, the source and execution character sets contain (at least) the following characters:

```
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
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . /
: ; < = > ? [ \ ] ^ _ { | } ~
```

as well as the *space* (blank), *horizontal tab, vertical tab* and *form feed.* Also, a *new line* character will exist for both the source and execution character sets.

Any character other than those previously listed should appear in a source file in a character constant, a string or a comment, otherwise the behavior is undefined.

If the character set of the computer being used to compile the program does not contain a certain character, a *trigraph* sequence may be used to represent it. Refer to the section "Character Constants".

The Open Watcom C16 and C32 compilers use the full IBM PC character set as both the source and execution character sets. The set of values from hexadecimal 00 to 7F constitute the ASCII character set.

3.1.1 Multibyte Characters

A multibyte character, as its name implies, is a character whose representation consists of more than one byte. Multibyte characters allow compilers to provide extended character sets, often for human languages that contain more characters than those found in the one-byte character set.

Multibyte characters are generally restricted to:

- comments,
- string literals,
- character constants,
- header names.

The method for specifying multibyte characters generally varies depending upon the extended character set.

3.2 Keywords

The following words are reserved as part of the C language and are called *keywords*. They may not be used for any kind of identifier, including object names, function names, labels, structure or union tags (names).

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

The Open Watcom compilers also reserve the following extended keywords:

Microsoft compilers compatible

```
__asm           __finally       __pascal
__based         __fortran       __saveregs
__cdecl         __huge          __segment
__declspec      __inline        __segname
__except        __int64         __self
__export        __interrupt     __stdcall
__far           __leave         __syscall
__far16         __loadds        __try
__fastcall      __near          __unaligned
```

IBM compilers compatible

```
_Cdecl          _Finally        _Seg16
_Except         _Leave          _Syscall
```

```
_Export                 _Packed                 _System
_Far16                  _Pascal                 _Try
_Fastcall
```

Open Watcom specific

```
__builtin_isfloat       __ow_imaginary_unit     __watcall
```

Note that, since C is sensitive to the case of letters, changing one or more letters in a keyword to upper case will prevent the compiler from recognizing it as a keyword, thereby allowing it to be used as an identifier. However, this is not a recommended programming practice.

3.3 Identifiers

Identifiers are used as:

- object or variable names,
- function names,
- labels,
- structure, union or enumeration tags,
- the name of a member of a structure or union,
- enumeration constants,
- macro names,
- typedef names.

An identifier is formed by a sequence of the following characters:

- upper-case letters "A" through "Z",
- lower-case letters "a" through "z",
- the digits "0" through "9",
- the underscore "_".

The first character may not be a digit.

An identifier cannot be a member of the list of keywords.

Identifiers can consist of any number of characters, but the compiler is not required to consider more than 31 characters as being significant, provided the identifier does not have *external linkage* (shared by more than one compiled module of the program). If the identifier is external, the compiler is not required to consider more than 6 characters as being significant. External identifiers may be case-sensitive.

Of course, any particular compiler may choose to consider more characters as being significant, but a portable C program will strictly adhere to the above rules. (This restriction is likely to be relaxed in future versions of the ISO C standard and corresponding C compilers.)

The Open Watcom C^{16} and C^{32} compilers do not restrict the number of significant characters for functions or objects with external or internal linkage.

The linker provided with Open Watcom C^{16} and C^{32} restricts the number of significant characters in external identifiers to 40 characters, and by default, distinguishes between identifiers that differ only in the case of the letters. An option may be used to force the linker to ignore case differences.

Any external identifier that starts with the underscore character ("_") may be reserved by the compiler. Any other identifier that starts with two underscores, or an underscore and an upper-case letter may be reserved. Generally, a program should avoid creating identifiers that start with an underscore.

3.4 Comments

A *comment* is identified by `/*` followed by any characters and terminated by `*/`. Comments are recognized anywhere in a program, except inside a character constant or string. Once the `/*` is found, characters are examined only until the `*/` is found. This excludes nesting of comments.

A comment is treated as a "white-space" character, meaning that it is like a space character.

For example, the program fragment,

```
/* Close all the files.
 */
    for( i = 0; i < fcount; i++ ) { /* loop through list */
        fclose( flist[i] );         /* close the file */
    }
```

is equivalent to,

```
    for( i = 0; i < fcount; i++ ) {
        fclose( flist[i] );
    }
```

Comments are sometimes used to temporarily remove a section of code during testing or debugging of a program. For example, the second program fragment could be "commented out" as follows:

```
/*
     for( i = 0; i < fcount; i++ ) {
         fclose( flist[i] );
     }
*/
```

This technique will not work on the first fragment because it contains comments, and comments may not be nested. For these cases, the `#if` directive of the C preprocessor may be used. Refer to the chapter "The Preprocessor" for more details.

The Open Watcom C[16] and C[32] compilers support an extension for comments. The symbol `//` can be used at any point in a physical source line (except inside a character constant or string literal). Any characters from the `//` to the end of the line are treated as comment characters. The comment is terminated by the end of the line. There is no explicit symbol for terminating the comment. For example, the program fragment used at the beginning of this section can be rewritten as,

```
// Close all the files.

    for( i = 0; i < fcount; i++ ) { // loop through list
        fclose( flist[i] );         // close the file
    }
```

This form of comment can be used to "comment out" code without the difficulties encountered with `/*`.

4 Basic Types

The following topics are discussed:

- Declarations of Objects
- Integer Types
- Floating-Point Types
- Enumerated Types
- Arrays
- Strings

4.1 Declarations of Objects

When a name is used in a program, the compiler needs to know what that name represents. A *declaration* describes to the compiler what a name is, including:

- How much storage it occupies (objects) or how much storage is required for the value that is returned (functions), and how the value in that storage is to be interpreted. This is called the *type.* Examples include `int`, `float` and `struct list`.
- Whether the name is visible only within the module being compiled, or throughout the program. This is called the *linkage,* and is part of the *storage class.* The keywords `extern` and `static` determine the linkage.
- For object names, whether the object is created every time the function is called and destroyed every time the function returns. This is called the *storage duration,* and is part of the *storage class.* The keywords `extern`, `static`, `auto` and `register` determine the storage duration.

The placement of the declaration within the program determines whether the declaration applies to all functions within the module, or just to the function within which the declaration appears.

The *definition* of an object is similar to its declaration, except that the storage for the object is reserved. Whether the declaration of an object is also a definition depends upon the placement of the declaration and the attributes of the object.

The usual form for defining (creating) an object is as follows:

storage-class-specifier type-specifier declarator;
or
storage-class-specifier type-specifier declarator = initializer;

The *storage-class-specifier* is optional, and is thoroughly discussed in the chapter "Storage Classes". The *type-specifier* is also optional, and is thoroughly discussed in the next section and in the chapter "Advanced Types". At least one of the *storage-class-specifier* and *type-specifier* must be specified, and they may be specified in either order, although it is recommended that the *storage-class-specifier* always be placed first.

The *declarator* is the name of the object being defined along with other information about its type. There may be several declarators, separated by commas.

The *initializer* is discussed in the chapter "Initialization of Objects".

The following are examples of declarations of objects, along with a brief description of what each one means. A more complete discussion of the terms used may be found in the relevant section.

```
int x;
```

Inside a function
: The object `x` is declared to be an *integer,* with *automatic storage duration*. Its value is available only within the function (or compound statement) in which it is defined. This is also a definition.

Outside a function
: The object `x` is created and declared to be an *integer* with *static storage duration*. Its value is available within the *module* in which it is defined, and has *external linkage* so that any other module may refer to it by using the declaration,

```
extern int x;
```

: This is also a definition.

```
register void * memptr;
```

Inside a function
: The object `memptr` is declared to be a *pointer to* `void` (no particular type of object), and is used frequently in the function. This is also a definition.

Outside a function
: Not valid because of the `register` storage class.

```
auto long int x, y;
```

Inside a function
: The objects `x` and `y` are declared to be *signed long integers* with *automatic storage duration*. This is also a definition.

Outside a function
: Not valid because of the `auto` storage class.

```
static int nums[10];
```

Inside a function
: The object `nums` is declared to be an *array of 10 integers* with *static storage duration*. Its value is only available within the function, and will be preserved between calls to the function. This is also a definition.

Outside a function
: The object `nums` is declared to be an *array of 10 integers* with *static storage duration*. Its value is only available within the *module*. (The difference is the *scope* of the object `nums`.) This is also a definition.

```
extern int x;
```

Inside a function
: The object `x` is declared to be an *integer* with *static storage duration*. No other functions within the current module may refer to `x` unless they also declare it. The object is defined in another module, or elsewhere in this function or module.

Outside a function
: The object `x` is declared to be an *integer* with *static storage duration*. Its value is available to all functions within the module. The object is defined in another module, or elsewhere in this module.

The appendix "Examples of Declarations" contains many more examples of declarations of objects and functions.

4.2 Name Scope

An identifier may be referenced only within its *scope*.

An identifier declared within a function or within a compound statement within a function has *block* scope, and may be referenced only in the block in which it is declared. The object's scope includes any enclosed blocks and terminates at the } which terminates the enclosing block.

An identifier declared within a function prototype (as a parameter to that function) has *function prototype* scope, and may not be referenced elsewhere. Its scope terminates at the) which terminates the prototype.

An identifier declared outside of any function or function prototype has *file* scope, and may be referenced anywhere within the module in which it is declared. If a function contains a declaration for the same identifier, the identifier with file scope is hidden within the function. Following the terminating } of the function, the identifier with file scope becomes visible again.

A label, which must appear within a function, has *function* scope.

4.3 Type Specifiers

Every object has a *type* associated with it. Functions may be defined to return a value, and that value also has a type. The type describes the interpretation of a value of that type, such as whether it is signed or unsigned, a pointer, etc. The type also describes the amount of storage required. Together, the amount of storage and the interpretation of stored values describes the range of values that may be stored in that type.

There are a number of different types defined by the C language. They provide a great deal of power in selecting methods for storing and moving data, and also contribute to the readability of the program.

There are a number of "basic types", those which will appear in virtually every program. More sophisticated types provide methods to describe data structures, and are discussed in the chapter "Advanced Types".

A *type specifier* is one or more of:

```
char
double
float
int
long
short
signed
unsigned
void
enumeration
structure
union
typedef name
```

and may also include the following *type qualifiers*:

```
const
volatile
```

The Open Watcom compilers also provide the following extended *type qualifiers*:

```
__based          __fortran        _Seg16
_Cdecl           __huge           __segment
__cdecl          __inline         __segname
__declspec       __int64          __self
_Export          __interrupt      __stdcall
__export         __loadds         _Syscall
__far            __near           __syscall
_Far16           _Packed          _System
__far16          _Pascal          __unaligned
_Fastcall        __pascal         __watcall
__fastcall       __saveregs
```

For the extended type qualifiers, see the appendix "Compiler Keywords".

Various combinations of these keywords may be used when declaring an object. Refer to the section on the type being defined.

The main types are `char`, `int`, `float` and `double`. The keywords `short`, `long`, `signed`, `unsigned`, `const` and `volatile` modify these types.

4.4 Integer Types

The most commonly used type is the integer. Integers are used for storing most numbers that do not require a decimal point, such as counters, sizes and indices into arrays. The range of integers is limited by the underlying machine architecture and is usually determined by the range of values that can be handled by the most convenient storage type of the hardware. Most 16-bit machines can handle integers in the range `-32768` to `32767`. Larger machines typically handle integers in the range `-2147483648` to `2147483647`.

The general integer type includes a selection of types, specifying whether or not the value is to be considered as signed (negative and positive values) or unsigned (non-negative values), character (holds one character of the character set), short (small range), long (large range) or long long (very large range).

Just specifying the type `int` indicates that the amount of storage should correspond to the most convenient storage type of the hardware. The value is treated as being a signed quantity. According to the C language standard, the minimum range for `int` is `-32767` to `32767`, although a compiler may provide a greater range.

> With Open Watcom C[16], `int` has a range of `-32768` to `32767`.
>
> With Open Watcom C[32], `int` has a range of `-2147483648` to `2147483647`.

Specifying the type `char` indicates that the amount of storage is large enough to store any member of the execution character set. If a member of the required source character set (see "Character Sets") is stored in an object of type `char`, then the value is guaranteed to be positive. Whether or not other characters are positive is implementation-defined. (In other words, whether `char` is signed or unsigned is implementation-defined. If it is necessary for the object of type `char` to be signed or unsigned, then the object should be declared explicitly, as described below.)

> The Open Watcom C[16] and C[32] compilers define `char` to be `unsigned`, allowing objects of that type to store values in the range `0` to `255`. A command line switch may be specified to cause `char` to be treated as `signed`. This switch should only be used when porting a C program from a system where `char` is signed.

The `int` keyword may be specified with the keywords `short` or `long`. These keywords provide additional information about the range of values to be stored in an object of this type. According to the C language standard, a signed short integer has a minimum range of `-32767` to `32767`. A signed long integer has a minimum range of `-2147483647` to `2147483647`. A signed long long integer has a minimum range of `-9223372036854775807` to `9223372036854775807`.

> With Open Watcom C[16] and C[32], `short int` has a range of `-32768` to `32767`, while `long int` has a range of `-2147483648` to `2147483647`, and `long long int` has a range of `-9223372036854775808` to `9223372036854775807`.

The `char` and `int` types may be specified with the keywords `signed` or `unsigned`. These keywords explicitly indicate whether the type represents a signed or unsigned (non-negative) quantity.

The keyword `int` may be omitted from the declaration if one (or more) of the keywords `signed`, `unsigned`, `short` or `long` is specified. In other words, `short` is equivalent to `signed short int` and `unsigned long` is equivalent to `unsigned long int`.

The appendix "Macros for Numerical Limits" discusses a set of macro definitions describing the range and other characteristics of the various numeric types. The macros from the header `<limits.h>`, which describe the integer types, are discussed.

The following table describes all of the various integer types and their ranges as implemented by the Open Watcom C^{16} and C^{32} compilers. Note that the table is in order of increasing storage size.

Type	Minimum Value	Maximum Value
`signed char`	`-128`	`127`
`unsigned char`	`0`	`255`
`char`	`0`	`255`
`short int`	`-32768`	`32767`
`unsigned short int`	`0`	`65535`
`int` (C^{16}) `int` (C^{32})	`-32768` `-2147483648`	`32767` `2147483647`
`unsigned int` (C^{16}) `unsigned int` (C^{32})	`0` `0`	`65535` `4294967295`
`long int`	`-2147483648`	`2147483647`
`unsigned long int`	`0`	`18446744073709551615`
`long long int`	`-9223372036854775808`	`9223372036854775807`
`unsigned long long`	`0`	`18446744073709551615`

With Open Watcom C^{16}, an object of type `int` has the same range as an object of type `short int`.

With Open Watcom C^{32}, an object of type `int` has the same range as an object of type `long int`.

The following are some examples of declarations of objects with integer type:

```
char               a;
unsigned char      b;
signed char        c;
short              d;
unsigned short int e;
int                f,g;
signed             h;
unsigned int       i;
long               j;
unsigned long      k
signed long        l;
unsigned long int  m;
signed long long   n;
long long          o;
unsigned long long p;
long long int      q;
```

4.5 Floating-Point Types

A floating-point number is a number which may contain a decimal point and digits following the decimal point. The range of floating-point numbers is usually considerably larger than that of integers, but the efficiency of integers is usually much greater. Integers are always exact quantities, whereas floating-point numbers sometimes suffer from round-off error and loss of precision.

On some computers, floating-point arithmetic is *emulated* (simulated) by software, rather than hardware. Software emulation can greatly reduce the speed of a program. While this should not affect the portability of a program, a prudent programmer limits the use of floating-point numbers.

There are three floating-point number types, `float`, `double`, and `long double`.

The appendix "Macros for Numerical Limits" discusses a set of macro definitions describing the range and other characteristics of the various numeric types. The macros from the header `<float.h>`, which describe the floating-point types, are discussed.

The following table gives the ranges available on the 80x86/80x87 using the Open Watcom C^{16} and C^{32} compiler. The floating-point format is the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).

Type	Smallest Absolute Value	Largest Absolute Value	Digits Of Accuracy	80x87 Type Name
`float`	`1.1E-38`	`3.4E+38`	`6`	short real
`double`	`2.2E-308`	`1.7E+308`	`15`	long real
`long double`	`2.2E-308`	`1.7E+308`	`15`	long real

By default, the Open Watcom C16 and C32 compilers emulate floating-point arithmetic. If the 8087 or 80x87 Numeric Processor Extension (numeric coprocessor, math chip) will be present at execution time, the compiler can be forced to generate floating-point instructions for the coprocessor by specifying a command line switch, as described in the User's Guide. Other than an improvement in execution speed, the final result should be the same as if the processor is not present.

The following are some examples of declarations of objects with floating-point type:

```
float       a;
double      b;
long double c;
```

4.6 Enumerated Types

Sometimes it is desirable to have a list of constant values representing different things, and the exact values are not relevant. They may need to be unique or may have duplicates. For example, a set of actions, colors or keys might be represented in such a list. An *enumerated type* allows the creation of a list of items.

An *enumerated type* is a set of identifiers that correspond to constants of type `int`. These identifiers are called *enumeration constants.* The first identifier in the set has the value 0, and subsequent identifiers are given the previous value plus one. Wherever a constant of type `int` is allowed, an enumeration constant may be specified.

The following type specifier defines the set of actions available in a simple memo program:

```
enum actions { DISPLAY, EDIT, PURGE };
```

The enumeration constant `DISPLAY` is equivalent to the integer constant 0, and `EDIT` and `PURGE` are equivalent to 1 and 2 respectively.

An enumerated type may be given an optional *tag* (name) with which it may be identified elsewhere in the program. In the example above, the tag of the enumerated type is `actions`, which becomes a new type. If no tag is given, then only those objects listed following the definition of the type may have the enumerated type.

The *name space* for enumerated type tags is different from that of object names, labels and member names of structures and unions, so a tag may be the same identifier as one of these other kinds. An enumerated type tag may not be the same as the tag of a structure or union, or another enumerated type.

Enumeration constants may be given a specific value by specifying '=' followed by the value. For example,

```
enum colors { RED = 1, BLUE = 2, GREEN = 4 };
```

creates the constants `RED, BLUE` and `GREEN` with values 1, 2 and 4 respectively.

```
enum fruits { GRAPE, ORANGE = 6, APPLE, PLUM };
```

creates constants with values 0, 6, 7 and 8.

```
enum fruits { GRAPE, PLUM, RAISIN = GRAPE, PRUNE = PLUM };
```

makes `GRAPE` and `RAISIN` equal to 0, and `PLUM` and `PRUNE` equal to 1.

The formal specification of an enumerated type is as follows:

`enum` *identifier*
or
`enum` { *enumeration-constant-list* }
or
`enum` *identifier* { *enumeration-constant-list* }

enumeration-constant-list:
enumeration-constant
or
enumeration-constant, enumeration-constant-list

enumeration-constant:
identifier
or
identifier = *constant-expression*

The type of an enumeration is implementation-defined, although it must be compatible with an integer type. Many compilers will use `int`.

From the following table, the Open Watcom C[16] and Open Watcom C[32] compilers will choose the smallest type that has sufficient range to represent all of the constants of a particular enumeration:

Type	Smallest Value	Largest Value
signed char	-128	127
unsigned char	0	255
signed short	-32768	32767
unsigned short	0	65535
signed long	-2147483648	2147483647
unsigned long	0	4294967295
signed long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

A command-line option may be used to force all enumerations to `int`.

To create an object with enumerated type, one of two forms may be used. The first form is to create the type as shown above, and then to declare an object as follows:

`enum` *tag object-name*`;`

For example, the declaration,

```
enum fruits fruit;
```

declares the object `fruit` to be the enumerated type `fruits.`

The second form is to list the identifiers of the objects following the closing brace of the enumeration declaration. For example,

```
enum fruits { GRAPE, ORANGE, APPLE, PLUM } fruit;
```

Provided no other objects with the same enumeration are going to be declared, the enumerated type tag `fruits` is not required. The declaration could be specified as,

```
enum { GRAPE, ORANGE, APPLE, PLUM } fruit;
```

An identifier that is an enumeration constant may only appear in one enumeration type. For example, the constant `ORANGE` may not be included in another enumeration, because the compiler would then have two values for `ORANGE.`

4.7 Arrays

An *array* is a collection of objects which are all of the same type. All elements (objects) in the array are stored in contiguous (adjacent) memory.

References to array elements are usually made through *indexing* into the array. To facilitate this, the elements of the array are numbered starting at zero. Hence an array declared with `n` elements is indexed using indices between 0 and `n-1.`

An array may either be given an explicit size (using a constant expression) or its size may be determined by the number of values used to initialize it. Also, it is possible to declare an array without any size information, in the following cases:

- a parameter to a function is declared as "array of *type*" (in which case the compiler alters the type to be "pointer to *type*"),
- an array object has external linkage (extern) and the definition which creates the array is given elsewhere,
- the array is fully declared later in the same module.

An array of undetermined size is an *incomplete type*.

An array declaration is of the following form:

type identifier `[` *constant-expression* `];`
or
type identifier`[] = {` *initializer-list* `};`
or
type identifier`[` *constant-expression* `] = {` *initializer-list* `};`
or
type identifier`[];`

where *type* is the type of each element of the array, *identifier* is the name of the array, *constant-expression* is an expression that evaluates to a positive integer defining the number of elements in the array, and *initializer-list* is a list of values (of type *type*) to be assigned to successive elements of the array.

For example,

```
int values[10];
```

declares `values` to be an array of 10 integers, with indices from 0 to 9. The expression `values[5]` refers to the sixth integer in the array.

```
char text[] = { "some stuff" };
```

declares `text` to be an array of 11 characters, each containing successive letters from `"some stuff".` The value of `text[10]` is `'\0'` (the null character), representing the terminating character in the string (see Strings).

```
extern NODES nodelist[];
```

declares `nodelist` to be an array of `NODES` (defined elsewhere), and the array is of unknown size. In another source file or later in the current file, there must be a corresponding declaration of `nodelist` which defines how big the array actually is.

It is possible to declare multi-dimensional arrays by including more than one set of dimensions. For example,

```
int tbl[2][3];
```

defines a 2-row by 3-column array of integers. In fact, it defines an array of 2 arrays of 3 integers. The values are stored in memory in the following order:

```
tbl[0][0]
tbl[0][1]
tbl[0][2]
tbl[1][0]
tbl[1][1]
tbl[1][2]
```

The rows of the table are stored together. This form of storing an array is called *row-major order.* The expression `tbl[1][2]` refers to the element in the last row and last column of the array.

In an expression, if an array is named without specifying any indices, the value of the array name is the address of its first element. In the example,

```
int   array[10];
int * aptr;

aptr = array;
```

the assignment to `aptr` is equivalent to,

```
aptr = &array[0];
```

Since multi-dimensional arrays are just arrays of arrays, it follows that omission of some, but not all, dimensions is equivalent to taking the address of the first element of the sub-array. In the example,

```
int   array[9][5][2];
int * aptr;

aptr = array[7];
```

the assignment to `aptr` is equivalent to,

```
aptr = &array[7][0][0];
```

Note that no checking of indices is performed at execution time. An invalid index (less than zero or greater than the highest index) will refer to memory as if the array was extended to accommodate the index.

4.8 Strings

A *string* is a special form of the type "array of characters", specifically an array of characters terminated by a *null character*. The null character is a character with the value zero, represented as `\0` within a string, or as the character constant `'\0'`. Because string processing is such a common task in programming, C provides a set of library functions for handling strings.

A string is represented by the address of the first character in the string. The *length* of a string is the number of characters up to, but not including, the null character.

An array can be initialized to be a string using the following form:

type identifier`[] = { "`*string value*`" };`

(The braces are optional.) For example,

```
char ident[] = "This is my program";
```

declares `ident` to be an array of 19 characters, the last of which has the value zero. The string has 18 characters plus the null character.

In the above example, `ident` is an array whose value is a string. However, the quote-enclosed value used to initialize the array is called a *string literal.* String literals are described in the "Constants" chapter.

A string may be used anywhere in a program where a "pointer to `char`" may be used. For example, if the declaration,

```
char * ident;
```

was encountered, the statement,

```
ident = "This is my program";
```

would set the value of `ident` to be the address of the string `"This is my program"`.

5 Constants

A constant is a value which is fixed at compilation time and is often just a number, character or string. Every constant has a type which is determined by its form and value. For example, the value `1` may have the type `signed int`, while the value `400000` may have the type `signed long`. In many cases, the type of the constant does not matter. If, for example, the value `1` is assigned to an object of type `long int`, then the value `1` will be converted to a long integer before the assignment takes place.

5.1 Integer Constants

An integer constant begins with a digit and contains no fractional or exponent part. A prefix may be included which defines whether the constant is in octal, decimal or hexadecimal format.

A constant may be suffixed by `u` or `U` indicating an `unsigned int`, or by `l` or `L` indicating a `long int`, or by both indicating an `unsigned long int`.

If a constant does not start with a zero and contains a sequence of digits, then it is interpreted as a decimal (base 10) constant. These are decimal constants:

```
7
762
98765L
```

If the constant starts with `0x` or `0X` followed by the digits from `0` through `9` and the letters `a` (or `A`) through `f` (or `F`), then the constant is interpreted as a hexadecimal (base 16) constant. The letters `A` through `F` represent the values 10 through 15 respectively. These are hexadecimal constants:

```
0X07FFF
0x12345678L
0xFABE
```

If a constant starts with a zero, then it is an octal constant and may contain only the digits 0 through 7. These are octal constants:

```
017
0735643L
0
```

Note that the constant `0` is actually an octal constant, but is zero in decimal, octal and hexadecimal.

The following table describes what type the compiler will give to a constant. The left column indicates what base (decimal, octal or hexadecimal) is used and what suffixes (`U` or `L`) are present. The right column indicates the types that may be given to such a constant. The type of an integer constant is the first type from the table in which its value can be accurately represented.

Constant	Type
unsuffixed decimal	`int, long, unsigned long`
unsuffixed octal	`int, unsigned int, long, unsigned long`
unsuffixed hexadecimal	`int, unsigned int, long, unsigned long`
suffix `U` only	`unsigned int, unsigned long`
suffix `L` only	`long, unsigned long`
suffixes `U` and `L`	`unsigned long`
suffix `LL` only	`long long, unsigned long long`
suffixes `U` and `LL`	`unsigned long long`

The following table illustrates a number of constants and their interpretation and type:

Constant	Decimal Value	Hexa -decimal Value	Open Watcom C[16] Type	Open Watcom C[32] Type
33	33	21	signed int	signed int
033	27	1B	signed int	signed int
0x33	51	33	signed int	signed int
33333	33333	8235	signed long	signed int
033333	14043	36DB	signed int	signed int
0xA000	40960	A000	unsigned int	signed int
0x33333	209715	33333	signed long	signed int
0x80000000	2147483648	80000000	unsigned long	unsigned int
2147483648	2147483648	80000000	unsigned long	unsigned int
4294967295	4294967295	FFFFFFFF	unsigned long	unsigned int

5.2 Floating-Point Constants

A floating-point constant may be distinguished by the presence of either a period, an `e` or `E`, or both. It consists of a value part (mantissa) optionally followed by an exponent. The mantissa may include a sequence of digits representing a whole number, followed by a period, followed by a sequence of digits representing a fractional part. The exponent must start with an `e` or `E` followed by an optional sign (+ or −), and a digit sequence representing (with the sign) the power of 10 by which the mantissa should be multiplied. Optionally, the suffix `f` or `F` may be added indicating the constant has type `float`, or the suffix `l` or `L` indicating the constant has type `long double`. If no suffix is present then the constant has type `double`.

In the mantissa, either the whole number part or the fractional part must be present. If only the whole number part is present and no period is included then the exponent part must be present.

The following table illustrates a number of floating-point constants and their type:

Constant	Value	Type
`3.14159265`	`3.14159265E0`	`double`
`11E24`	`1.1E25`	`double`
`.5L`	`5E-1`	`long double`
`7.234E-22F`	`7.234E-22`	`float`
`0.`	`0E0`	`double`

5.3 Character Constants

A character constant is usually one character enclosed in single-quotes, and indicates a constant whose value is the representation of the character in the execution character set. A character constant has type `int`.

The character enclosed in quotes may be any character in the source character set. Certain characters in the character set may not be directly representable, since they may be assigned other meanings. These characters can be entered using the following escape sequences:

Character	Character Name	Escape Sequence
'	single quote	\'
"	double quote	" or \"
?	question mark	? or \?
\	backslash	\\
	octal value	*octal digits* (max 3)
	hexadecimal value	\x*hexadecimal digits*

For example,

```
'a'     /* the letter a */
'\''    /* a single quote */
'?'     /* a question mark */
'\?'    /* a question mark */
'\\'    /* a backslash */
```

are all simple character constants.

The following are some character constants containing octal escape sequences, made up of a `\` followed by one, two or three octal digits (the digits `0` through `7`):

```
'\0'
'\377'
'\100'
```

If a character constant containing an octal value is found, but a non-octal character is also present, or if a fourth octal digit is found, it is not part of the octal character already specified, and constitutes a separate character. For example,

```
'\1000'
'\109'
```

the first constant is a two-character constant, consisting of the characters `'\100'` and `'0'` (because an octal value consists of at most three octal digits). The second constant is also a two-character constant, consisting of the characters `'\10'` and `'9'` (because `9` is not an octal digit).

If more than one octal value is to be specified in a character constant, then each octal value must be specified starting with `\`.

The meaning of character constants with more than one character is implementation-defined.

The following are some character constants containing hexadecimal escape sequences, made up of a `\x` followed by one or more hexadecimal digits (the digits `0` through `9`, and the letters `a` through `f` and `A` through `F`). (The values of these character constants are the same as the first examples of octal values presented above.)

```
'\x0'
'\xFF'
'\x40'
```

If a character constant containing a hexadecimal value is found, but a non-hexadecimal character is also present, it is not part of the hexadecimal character already specified, and constitutes a separate character. For example,

```
'\xFAx'
'\xFx'
```

the first constant is a two-character constant, consisting of the characters `'\xFA'` and `'x'` (because `x` is not a hexadecimal digit). The second constant is also a two-character constant, consisting of the characters `'\xF'` and `'x'`.

If more hexadecimal digits are found than are required to specify one character, the behavior is implementation-defined. Specifically, any sequence of hexadecimal characters in a hexadecimal value in a character constant is used to specify the value of one character. If more than one hexadecimal value is to be specified in a character constant, then each hexadecimal value must be specified starting with `\x`.

The meaning of character constants with more than one character is implementation-defined.

In addition to the above escape sequences, the following escape sequences may be used to represent non-graphic characters:

Escape Sequence	Meaning
\a	Causes an audible or visual alert
\b	Back up one character
\f	Move to the start of the next page
\n	Move to the start of the next line
\r	Move to the start of the current line
\t	Move to the next horizontal tab
\v	Move to the next vertical tab

The following trigraph sequences may be used to represent characters not available on all terminals or systems:

Character	Trigraph Sequence
[	??(
]	??)
{	??<
}	??>
|	??!
#	??=
\	??/
^	??'
~	??-

The Open Watcom C[16] and C[32] compilers also allow character constants with more than one character. These may be used to initialize larger types, such as `int`. For example, the program fragment:

```
int code;
code = 'ab';
```

assigns the constant value `'ab'` to the integer object `code`. The letter `b` is placed in the lowest order (least significant) portion of the integer value and the letter `a` is placed in the next highest portion.

Up to four characters may be placed in a character constant. Successive characters, starting from the right-most character in the constant, are placed in successively higher order (more significant) bytes of the result.

Note that a character constant such as `'a'` is different from the corresponding string literal `"a"`. The former is of type `int` and has the value of the letter `a` in the execution character set. The latter is of type "pointer to `char`" and its value is the address of the first character (`a`) of the string literal.

5.3.1 Wide Character Constants

If the value of a character constant is to be a multibyte character from an extended character set, then a *wide character constant* should be specified. Its form is similar to normal character constants, except that the constant is preceded by the character `L`.

The type of a wide character constant is `wchar_t`, which is one of the integral types, and is described in the header `<stddef.h>`.

With Open Watcom C[16] and C[32], `wchar_t` is defined as `unsigned short`.

For example, the constant `L'a'` is a wide character constant containing the letter `a` from the source character set, and has type `wchar_t`. In contrast, the constant `'a'` is a character constant containing the letter `a`, and has type `int`.

How the multibyte character maps onto the wide character value is defined by the `mbtowc` library function.

As shown above, a wide character constant may also contain a single byte character, since an extended character set contains the single byte characters. The single byte character is mapped onto the corresponding wide character code.

5.4 String Literals

A sequence of zero or more characters enclosed within double-quotes is a *string literal.*

Most of the same rules for creating character constants also apply to creating string literals. However, the single-quote may be entered directly or as the `\'` escape sequence. The double-quote must be entered as the `\"` escape sequence.

The value of a string literal is the sequence of characters within the quotes, plus a null character at the end.

The type of a string literal is "array of `char`".

The following are examples of string literals:

```
"Hello there"
"\"Quotes inside string\""
"G'day"
```

If two or more string literals are adjacent, the compiler will join them together into one string literal. The pair of string literals,

```
"Hello" "there"
```

would be joined by the compiler to be,

```
"Hellothere"
```

and is an array of 11 characters, including the single terminating null character.

The joining of adjacent string literals occurs *after* the replacement of escape sequences. In the examples,

```
"\xFAB\xFA" "B"
"\012\01" "2"
```

the first string, after joining, consists of three characters, with the values `'\xFAB'`, `'\xFA'` and `'B'`. The second string, after joining, also consists of three characters, with the values `'\012'`, `'\01'` and `'2'`.

A program should not attempt to modify a string literal, as this behavior is undefined. On computers where memory can be protected, it is likely that string literals will be placed where the program cannot modify them. An attempt to modify them will cause the program to fail. On other computers without such protection, the literal can be modified, but this is generally considered to be a poor programming practice. (Constants should be constant!)

A string literal normally is a string. It is not a string if one of the characters within double-quotes is the null character (`\0`). If such a string literal is treated as a string, then only those characters before the first null character will be considered part of the string. The characters following the first null character will be ignored.

If a source file uses the same string literal in several places, the compiler may combine them so that only one instance of the string exists and each reference refers to that string. In other words, the addresses of each of the string literals would be the same. However, no program should rely on this since other compilers may make each string a separate instance.

> The Open Watcom C16 and C32 compilers combine several instances of the same string literal in the same module into a single string literal, provided that they occur in declarations of constant objects or in statements other than declarations (eg. assignment).

If the program requires that several string literals be the same instance, then an object should be declared as an array of `char` with its value initialized to the string.

5.4.1 Wide String Literals

If any of the characters in a string literal are multibyte characters from an extended character set, then a *wide string literal* should be specified. Its form is similar to normal string literals, except that the string is preceded by the character `L`.

The type of a wide string literal is "array of `wchar_t`". `wchar_t` is one of the integral types, and is described in the header `<stddef.h>`.

> With Open Watcom C16 and C32, `wchar_t` is defined as `unsigned short`.

For example, the string literal `L"ab"` is a wide string literal containing the letters `a` and `b`. Its type is "array [3] of `wchar_t`", and the values of its elements are `L'a'`, `L'b'` and `'\0'`. In contrast, the string literal `"ab"` has type "array [3] of `char`", and the values of its elements are `'a'`, `'b'` and `'\0'`.

How the multibyte characters map onto wide character values is defined by the `mbtowc` library function.

As shown above, a wide string literal may also contain single byte characters, since the extended character set contains the single byte characters. The single byte characters are mapped onto the corresponding wide character codes.

Adjacent wide string literals will be concatenated by the compiler and a null character appended to the end. If a string literal and a wide string literal are adjacent, the behavior when the compiler attempts to concatentate them is undefined.

6 Type Conversion

Whenever two operands are involved in an operation, some kind of *conversion* of one or both of the operands may take place. For example, a `short int` and a `long int` cannot be directly added. Instead, the `short int` must first be converted to a `long int`, then the two values can be added.

Fortunately, C provides most conversions as *implicit* operations. Simply by indicating that the two values are to be added, the C compiler will check their types and generate the appropriate conversions. Sometimes it is necessary, however, to be aware of exactly how C will convert the operands.

Conversion of operands always attempts to preserve the value of the operand. Where preservation of the value is not possible, the compiler will sign-extend signed quantities and discard the high bits of quantities being converted to smaller types.

The rules of type conversions are fully discussed in the following sections.

6.1 Integral Promotion

Rule: A `char`, `short int` or `int` bit-field in either of their signed or unsigned forms, or an object that has an enumerated type, is always converted to an `int`. If the type `int` cannot contain the entire range of the object being converted, then the object will be converted to an `unsigned int`.

A `signed` or `unsigned char` will be converted to a `signed int` without changing the value.

> With Open Watcom C[16], a `short int` has the same range as `int`, therefore a `signed short int` is converted to a `signed int`, and an `unsigned short int` is converted to an `unsigned int`, without changing the value.
>
> With Open Watcom C[32], a `signed` or `unsigned short int` is converted to an `int` without changing the value.

These promotions are called the *integral promotions.*

6.2 Signed and Unsigned Integer Conversion

Rule: If an unsigned integer is converted to an integer type of any size, then, if the value can be represented in the new type, the value remains unchanged.

If an unsigned integer is converted to a longer type (type with greater range), then the value will not change. If it is converted to a type with a smaller range, then provided the value can be represented in the smaller range, the value will remain unchanged. If the value cannot be represented, then if the result type is signed, the result is implementation-defined. If the result type is `unsigned`, the result is the integer modulo (1+the largest unsigned number that can be stored in the shorter type).

With Open Watcom C[16], unsigned integers are promoted to longer types by extending the high-order bits with zeros. They are demoted to shorter types by discarding the high-order portion of the larger type.

Consider the following examples of 32-bit quantities (`unsigned long int`) being converted to 16-bit quantities (`signed short int` or `unsigned short int`):

long	32-bit representation	16-bit representation	`signed` short	`unsigned` short
65538	0x00010002	0x0002	2	2
100000	0x000186A0	0x86A0	−31072	34464

Rule: When a signed integer is converted to an unsigned integer of equal or greater length, if the value is non-negative, the value will be unchanged.

A non-negative value stored in a signed integer may be converted to an equal or larger integer type without affecting the value. A negative value is first converted to the signed type of the same length as the result, then (1+the largest unsigned number that can be stored in the result type) is added to the value to convert it to the unsigned type.

With Open Watcom C[16], signed integers are promoted to longer types by *sign-extending* the value (the high bit of the shorter type is propogated throughout the high bits of the longer type). When the longer type is unsigned, the sign-extended bit-pattern is then treated as an unsigned value.

Consider the following examples of 16-bit signed quantities (`signed short int`) being converted to 32-bit quantities (`signed long int` and `unsigned long int`):

`signed` short	16-bit represention	32-bit representation	`signed` long	`unsigned` long
−2	0xFFFE	0xFFFFFFFE	−2	4294967294
32766	0x7FFE	0x00007FFE	32766	32766

Rule: When a signed integer is converted to a longer signed integer, the value will not change.

Rule: When a signed integer is converted to a shorter type, the result is implementation-defined.

With Open Watcom C[16], signed integers are converted to a shorter type by preserving the low-order (least significant) portion of the larger type.

6.3 Floating-Point to Integer Conversion

Rule: When a floating-point type is converted to integer, the fractional part is discarded. If the value of the integer part cannot be represented in the integer type, then the result is undefined.

Hence, it is valid only to convert a floating-point type to integer within the range of the integer type being converted to. Refer to the section "Integer Types" for details on the range of integers.

6.4 Integer to Floating-Point Conversion

Rule: When the value of an integer type is converted to a floating-point type, and the integer value cannot be represented exactly in the floating-point type, the value will be rounded either up or down.

Rounding of floating-point numbers is implementation-defined. The technique being used by the compiler may be determined from the macro `FLT_ROUNDS` found in the header `<float.h>`. The following table describes the meaning of the various values:

`FLT_ROUNDS`	Technique
`-1`	indeterminable
0	toward zero
1	to nearest number
2	toward positive infinity
3	toward negative infinity

The Open Watcom C[16] and C[32] compilers will round to the nearest number. (The value of `FLT_ROUNDS` is 1.)

Rule: When a floating-point value is converted to a larger floating-point type (`float` to `double`, `float` to `long double`, or `double` to `long double`), the value remains unchanged.

Rule: When any floating-point type is demoted to a floating-point type with a smaller range, then the result will be undefined if the value lies outside the range of the smaller type. If the value lies inside the range, but cannot be represented exactly, then rounding will occur in an implementation-defined manner.

The Open Watcom C[16] and C[32] compilers round to the nearest number. (The value of `FLT_ROUNDS` is 1.)

6.5 Arithmetic Conversion

Whenever two values are used with a binary operator that expects arithmetic types (integer or floating-point), conversions may take place implicitly. Most binary operators work on two values of the same type. If the two values have different types, then the type with the smaller range is always promoted to the type with the greater range. Conceptually, each type is found in the table below and the type found lower in the table is converted to the type found higher in the table.

```
long double
double
float
unsigned long
long
unsigned int
int
```

Note that any types smaller than `int` have *integral promotions* performed on them to promote them to `int`.

The following table illustrates the result type of performing an addition on combinations of various types:

Operation	Result Type
`signed char + signed char`	`signed int`
`unsigned char + signed int`	`signed int`
`signed int + signed int`	`signed int`
`signed int + unsigned int`	`unsigned int`
`unsigned int + signed long`	`signed long`
`signed int + unsigned long`	`unsigned long`
`signed char + float`	`float`
`signed long + double`	`double`
`float + double`	`double`
`float + long double`	`long double`

6.6 Default Argument Promotion

When a call is made to a function, the C compiler checks to see if the function has been defined already, or if a prototype for that function has been found. If so, then the arguments to the function are converted to the specified types. If neither is true, then the arguments to the function are promoted as follows:

- all integer types have the *integral promotions* performed on them, and,
- all arguments of type `float` are promoted to `double`.

If the definition of the function does not have parameters with types that match the promoted types, the behavior is undefined.

7 Advanced Types

The following topics are discussed:

- Structures
- Unions
- Pointers
- Void
- The const and volatile Declarations

7.1 Structures

A *structure* is a type composed of a sequential group of *members* of various types. Like other types, a structure is a model describing storage requirements and interpretations, and does not reserve any storage. Storage is reserved when an object is declared to be an *instance* of the structure.

Each of the members of a structure must have a name, with the exception of *bit-fields.*

> With Open Watcom C16 and C32, a structure member may be unnamed if the member is a structure or union.

A structure may not contain a member with an incomplete type. In particular, it may not contain a member with a type of the structure being defined (otherwise the structure would have indeterminate size), although it may contain a pointer to it.

The structure may be given an optional *tag* with which the structure may be referenced elsewhere in the program. If no tag is given, then only those objects listed following the definition of the structure may have the structure type.

The *name space* for structure tags is different from that of object names, labels and member names, so a tag may be the same identifier as one of these other kinds. A structure tag may not be the same as the tag of a union or enumerated type, or another structure.

Each structure has its own name space, so an identifier may be used as a member name in more than one structure. An identifier that is an object name, structure tag, union tag, union member name, enumeration tag or label may also be used as a member name without ambiguity.

Structures help to organize program data by collecting several related objects into one object. They are also used for linked lists, trees and for describing externally-defined regions of data that the application must access.

The following structure might describe a token identified by parsing a typed command:

```
struct tokendef {
    int    length;
    int    type;
    char   text[80];
};
```

This defines a structure containing three members, an integer containing the token length, another integer containing some encoding of the token type, and the third an array of 80 characters containing the text of the token. The tag of the structure is `tokendef.`

The above definition does not actually create an object containing the structure. Creation of an instance of the structure requires a list of identifiers following the structure definition, or to use `struct tokendef` in place of a type for declaring an object. For example,

```
struct tokendef {
    int    length;
    int    type;
    char   text[80];
} token;
```

is equivalent to,

```
struct tokendef {
    int    length;
    int    type;
    char   text[80];
};

struct tokendef token;
```

Both create the object `token` as an instance of the structure `tokendef.` The *type* of `token` is `struct tokendef.`

References to a member of a structure are made using the *dot* operator (.). The first operand of the . operator is the object containing the structure. The second operand is the name of the member. For example, `token.length` refers to the `length` member of the `tokendef` structure contained in `token.`

If `tokenptr` is declared as,

```
struct tokendef * tokenptr;
```

(`tokenptr` is a pointer to a `tokendef` structure), then,

```
(*tokenptr).length
```

refers to the `length` member of the `tokendef` structure that `tokenptr` points to. Alternatively, to refer to a member of a structure, the *arrow* operator (`->`) is used:

```
tokenptr->length
```

is equivalent to,

```
(*tokenptr).length
```

If a structure contains an unnamed member which is a structure or union, then the members of the inner structure or union are referenced as if they were members of the outer structure. For example,

```
struct outer {
    struct inner {
        int     a, b;
    };
    int c;
} X;
```

The members of `X` are referenced as `X.a`, `X.b` and `X.c`.

Each member of a structure is at a higher address than the previous member. Alignment of members may cause (unnamed) gaps between members, and an unnamed area at the end of the structure.

The Open Watcom C[16] and C[32] compilers provide a command-line switch and a `#pragma` to control the alignment of members of structures. See the User's Guide for details.

In addition, the `_Packed` keyword is provided, and if specified before the `struct` keyword, will force the structure to be packed (no alignment, no gaps) regardless of the setting of the command-line switch or the `#pragma` controlling the alignment of members.

A pointer to an object with a structure type, suitably cast, is also a pointer to the first member of the structure.

A structure declaration of the form,

`struct` *tag;*

can be used to declare a new structure within a block, temporarily hiding the old structure. When the block ends, the previous structure's hidden declaration will be restored. For example,

```
struct thing { int a,b; };
/* ... */
    {
        struct thing;
        struct s1    { struct thing * thingptr; } tptr;
        struct thing { struct s1 *    s1ptr; }    sptr;
    }
```

the original definition of `struct thing` is suppressed in order to create a new definition. Failure to suppress the original definition would result in `thingptr` being a pointer to the old definition of `thing` rather than the new one.

Redefining structures can be confusing and should be avoided.

7.1.1 Bit-fields

A member of a structure can be declared as a *bit-field*, provided the type of the member is `int`, `unsigned int` or `signed int`.

> In addition, the Open Watcom C16 and C32 compilers allow the types `char`, `unsigned char`, `short int` and `unsigned short int` to be bit-fields.

A bit-field declares the member to be a number of bits. A value may be assigned to the bit-field in the same manner as other integral types, provided the value can be stored in the number of bits available. If the value is too big for the bit-field, excess high bits are discarded when the value is stored.

The type of the bit-field determines the treatment of the highest bit of the bit-field. Signed types cause the high bit to be treated as a sign bit, while unsigned types do not treat it as a sign bit. For a bit-field defined with type `int` (and no `signed` or `unsigned` keyword), whether or not the high bit is considered a sign bit is implementation-defined.

> The Open Watcom C16 and C32 compilers treat the high bit of a bit-field of type `int` as a sign bit.

A bit-field is declared by following the member name by a colon and a constant expression which evaluates to a non-negative value that does not exceed the number of bits in the type.

A bit-field may be declared without a name and may be used to align a structure to an imposed form. Such a bit-field cannot be referenced.

If two bit-fields are declared sequentially within the same structure, and they would both fit within the storage unit assigned to them by the compiler, then they are both placed within the same storage unit. If the second bit-field doesn't fit, then whether it is placed in the next storage unit, or partially placed in the same unit as the first and spilled over into the next unit, is implementation-defined.

> The Open Watcom C16 and C32 compilers place a bit-field in the next storage unit if it will not fit in the remaining portion of the previously defined bit-field. Bit-fields are not allowed to straddle storage unit boundaries.

An unnamed member declared as `: 0` prevents the next bit-field from being placed in the same storage unit as the previous bit-field.

The order that bit-fields are placed in the storage unit is implementation-defined.

> The Open Watcom C16 and C32 compilers place bit-fields starting at the low-order end (least significant bit) of the storage unit. If a 1-bit bit-field is placed alone in an `unsigned int` then a value of 1 in the bit-field corresponds to a value of 1 in the integer.

Consider the following structure definition:

```
struct list_el {
    struct list_el * link;
    unsigned short   elnum;
    unsigned int     length    : 3;
    signed int       offset    : 4;
    int              flag      : 1;
    char *           text;
};
```

The structure `list_el` contains the following members:

1. `link` is a pointer to a `list_el` structure, indicating that instances of this structure will probably be used in a linked list,
2. `elnum` is an unsigned short integer,
3. `length` is an unsigned bit-field containing 3 bits, allowing values in the range 0 through 7,
4. `offset` is a signed bit-field containing 4 bits, which will be placed in the same integer with `length`. Since the type is `signed int`, the range of values for this bit-field is −8 through 7,
5. `flag` is a 1-bit field,

   > Since the type is `int`, the Open Watcom C16 and C32 compilers will treat the bit as a sign bit, and the set of values for the bit-field is −1 and 0.

6. `text` is a pointer to character, possibly a string.

7.2 Unions

A *union* is similar to a structure, except that each member of a union is placed starting at the same storage location, rather than in sequentially higher storage locations. (The Pascal term for a union is "variant record".)

The *name space* for union tags is different from that of object names, labels and member names, so a tag may be the same identifier as one of these other kinds. The tag may not be the same identifier as the tag of a structure, enumeration or another union.

Each union has its own name space, so an identifier may be used as a member name in several different unions. An identifier that is an object name, structure tag, structure member name, union tag, enumeration tag or label may also be used as a member name without ambiguity.

> With Open Watcom C16 and C32, unions, like structures, may contain unnamed members that are structures or unions. References to the members of an unnamed structure or union are made as if the members of the inner structure or union were at the outer level.

The size of a union is the size of the largest of the members it contains.

A pointer to an object that is a union points to each of the members of the union. If one or more of the members of the union is a *bit-field*, then a pointer to the object also points to the storage unit in which the bit-field resides.

Storing a value in one member of a union, and then referring to it via another member is only meaningful when the different members have the same type. Members of a union may themselves be structures, and if some or all of the members start with the same members in each structure, then references to those structure members may be made via any of the union members. For example, consider the following structure and union definitions:

```
struct rec1 {
    int       rectype;
    int       v1,v2,v3;
    char *    text;
};

struct rec2 {
    int       rectype;
    short int flags : 8;
    enum      {red, blue, green} hue;
};

union alt_rec {
    struct rec1   val1;
    struct rec2   val2;
};
```

`alt_rec` is a union defining two members `val1` and `val2`, which are two different forms of a record, namely the structures `rec1` and `rec2` respectively. Each of the different record forms starts with the member `rectype.` The following program fragment would be valid:

```
union alt_rec  record;
/* ... */
record.rec1.rectype = 33;
DoSomething( record.rec2.rectype );
```

However, the following fragment would exhibit implementation-defined behavior:

```
record.rec1.v1 = 27;
DoSomethingElse( record.rec2.hue );
```

In other words, unless several members of a union are themselves structures where the first few members are of the same type, a program should not store into a union member and retrieve a value using another union member. Generally, a flag or other indicator is kept to describe which member of the union is currently the "active" member.

7.3 Pointers

A *pointer* to an object is equivalent to the address of the object in the memory of the computer.

An object may be declared to be a pointer to a type of object, or it may be declared to be a pointer to no particular type. The form,

type * *identifier*;

declares the identifier to be a pointer to the given type. If *type* is `void`, then the identifier is a pointer to no particular type of object (a generic pointer).

The following examples illustrate various pointer declarations:

```
int * intptr;
```
`intptr` is a pointer to an `int`.

```
char * charptr;
```
`charptr` is a pointer to a `char`.

```
struct tokendef * token;
```
`token` is a pointer to the structure `tokendef`.

```
char * argv[];
```
`argv` is an array of pointers to `char` or an array of pointers to strings.

```
char ** strptr;
```
`strptr` is a pointer to a pointer to `char`.

```
void * dumpbeg;
```
`dumpbeg` is a pointer, but to no particular type of object.

Any place that a pointer may be used, the constant 0 may also be used. This value is the *null pointer constant.* The value that is used internally to represent a null pointer is guaranteed not to be a pointer to an object. It does not necessarily correspond to the integer value 0. It merely represents a pointer that does not currently point at anything. The macro `NULL`, defined in the header `<stddef.h>`, may also be used in place of 0.

7.3.1 Special Pointer Types for Open Watcom C16

> Note: the following sections only apply to the Open Watcom C16 (16-bit) compiler. For the Open Watcom C32 compiler, see the section "Special Pointer Types for Open Watcom C32".

On the 8086, a normal pointer (16 bits) can only point to a 64K region of the total memory available on the machine. This effectively limits any program to a maximum of 64K of executable code and 64K of data. For many applications, this does not pose a limitation.

Some applications need more than 64K of code or data, or both. The Open Watcom C16 compiler provides a mechanism whereby pointers can be declared that get beyond the 64K limit. This can be done either by specifying an option when compiling the files (see the User's Guide) or by including a special type qualifier keyword in the declaration of the object. Later sections describe these keywords and their use.

The use of the keywords may prevent the program from compiling using other C compilers, in particular when the program is being transported to another system. However, the preprocessor can be used to eliminate the keywords on these other systems.

Before discussing the special pointer types, it is important to understand the different *memory models* that are available and what they mean. The five memory models are referred to as:

small small code (code < 64K), small data (data < 64K)

compact	small code (code < 64K), big data (total data > 64K, all objects < 64K)
medium	big code (code > 64K), small data (data < 64K)
large	big code (code > 64K), big data (total data > 64K, all objects < 64K)
huge	big code (code > 64K), huge data (total data > 64K, objects > 64K)

The following sections discuss the memory models in terms of "small" and "big" code and data sizes. The terms "small", "compact", "medium", "large" and "huge" are simply concise terms used to describe the combinations of code and data sizes available.

7.3.1.1 The Small and Big Code Models

Each program can use either *small code* (less than 64K) or *big code* (more than 64K). Small code means that all functions (together) must fit within the 64K limit on code size. It is possible to call a function using only a 16-bit pointer. This is the default.

Big code removes the restriction, but requires that all functions be called with a 32-bit pointer. A 32-bit pointer consists of two 16-bit quantities, called the *segment* and *offset.* (When the computer uses the segment and offset to refer to an actual memory location, the two values are combined to produce a 20-bit memory address, which allows for the addressing of 1024K of memory.) Because of the larger pointers, the code generated by the big code option takes more space and takes longer to execute.

When the big code option is being used, it is possible to group functions together into several 64K (or smaller) regions. Each module can be its own region, or several modules can be grouped. It is possible to call other functions within the same group using a 16-bit value. These functions are said to be *near.* Functions outside the group can still be called, but must be called using a 32-bit value. These functions are said to be *far.*

When the big code option is given on the command line for compiling the module, ordinary pointers to functions will be defined automatically to be of the larger type, and function calls will be done using the longer (32-bit) form.

It is also possible to use the small code option, and to override certain functions and pointers to functions as being *far.* However, this method may lead to problems. The Open Watcom C[16] compiler generates special function calls that the programmer doesn't see, such as checking for stack overflow when a function is invoked. These calls are either *near* or *far* depending entirely on the memory model chosen when the module is compiled. If the small code model is being used, all calls will be near calls. If, however, several code groups are created with far calls between them, they will all need to access the stack overflow checking routines. The linker can only place these special routines in one of the code groups, leaving the other functions without access to them, causing an error.

To resolve this problem, mixing code models requires that all modules be compiled with the big code model, overriding certain functions as being near. In this manner, the stack checking routines can be placed in any code group, which the other code groups can still access. Alternatively, a command-line switch may be used to turn off stack checking, so no stack checking routines get called.

7.3.1.2 The Small and Big Data Models

Each program can use either *small data* (less than 64K) or *big data* (more than 64K). Small data requires that all objects exist within one 64K region of memory. It is possible to refer to each object using a 16-bit pointer. This is the default.

Big data removes the restriction, but all pointers to data objects require a 32-bit pointer. As with the big code option, extra instructions are required to manipulate the 32-bit pointer, so the generated code will be larger and not as fast.

With either small or big data, each object is restricted in size to a maximum of 64K bytes. However, an object may be declared as *huge,* allowing the object to be bigger than 64K bytes. Pointers to huge objects are the least efficient because of extra code required to handle them, especially when doing pointer arithmetic. Huge objects are discussed in the section "The __huge Keyword".

When the big data option is being used, the program still retains one region up to 64K in size in which objects can be referred to using 16-bit pointers, regardless of the code group being executed. These objects are said to be *near.* Objects outside this region can still be referenced, but must be referred to using a 32-bit value. These objects are said to be *far.*

When the big data option is given on the command line for compiling the module, ordinary pointers to objects other than functions will be defined automatically to be of the larger type.

It is also possible to use the small data option, and to override certain objects as being *far.* The programmer must decide which method is easier to use.

7.3.1.3 Mixing Memory Models

It is possible to mix small and big code and data pointers within one program. In fact, a programmer striving for optimum efficiency will probably mix pointer types. But great care must be taken!

In some applications, the programmer may want the ability to have either big code or big data, but won't want to pay the extra-code penalty required to compile everything accordingly. In the case of big data, the programmer may realize that 99% of the data structures can reside within the 64K limit, and the remaining ones must go beyond that limit. Similarly, it may be desirable to have only a few functions that don't fit within the 64K limit.

When overriding the current memory model, it is *very* important to declare each type properly.

The following sections describe how to override the current memory model.

7.3.1.4 The __far Keyword for Open Watcom C[16]

When the big code memory model is in effect, functions are *far* and pointers to functions are declared automatically to be pointers to *far* functions. Similarly, the big data model causes all pointers to objects (other than functions) to be pointers to *far* objects. However, when either the small code or small data model is being used, the keyword `__far` may be used to override to the big model.

The `__far` keyword is a type qualifier that modifies the token that follows it. If `__far` precedes `*` (as in `__far` *), then the pointer points to something far. Otherwise, if `__far` precedes the identifier of the object or function being declared (as in `__far` x), then the object itself is far.

The keyword `__far` can only be applied to function and object names and the indirection (pointer) symbol `*`. Parameters to functions may *not* be declared as `__far` since they are always in the 64K data area that is near.

Open Watcom C[16] provides the predefined macros `far` and `_far` for convenience and compatibility with the Microsoft C compiler. They may be used in place of `__far`.

The following examples illustrate the use of the `__far` keyword. The examples assume that the small memory model (small code, small data) is being used.

```
int __far * ptr;
```
declares `ptr` to be a pointer to an integer. The object `ptr` is near (addressable using only 16 bits), but the value of the pointer is the address of an integer which is far, and so the pointer contains 32 bits.

```
int * __far fptr;
```
also declares `fptr` to be a pointer to an integer. However, the object `fptr` is far, but the integer that it points to is near.

```
int __far * __far ffptr;
```
declares `ffptr` to be a pointer (which is far) to an integer (which is far).

When declaring a function, placing the keyword `__far` in front of the function name causes the compiler to treat the function as being far. It is important, if the function is called before its definition, that a *function prototype* be included prior to any calls. For example, the declaration,

```
void __far BubbleSort();
```

declares the function `BubbleSort` to be far, meaning that any calls to it must be far calls.

Here are a few more examples. These, too, assume that the small memory model (small code, small data) is being used.

```
struct symbol * __far FSymAlloc( void );
```
declares the function `FSymAlloc` to be far, returning a pointer to a near `symbol` structure.

```
struct symbol __far * __far FFSymAlloc( void );
```
declares the function `FFSymAlloc` to be far, returning a pointer to a far `symbol` structure.

```
void Indirect( float __far fn() );
```
declares the function `Indirect` to be near, taking one parameter `fn` which is a pointer to a far function that returns a `float`.

```
int AdjustLeft( struct symbol * __far symptr );
```
is an invalid declaration, since it attempts to declare `symptr` to be far. All parameters must be near, since they reside in the 64K data area that is always near.

7.3.1.5 The __near Keyword for Open Watcom C[16]

When the small code memory model is in effect, functions are *near,* and pointers to functions are automatically declared to be pointers to *near* functions. Similarly, the small data model causes all pointers to objects (other than functions) to be pointers to *near* objects. However, when either the big code or big data model is being used, the keyword `__near` may be used to override to the small model.

The `__near` keyword is a type qualifier that modifies the token that follows it. If `__near` precedes `*` (as in `__near` *), then the pointer points to something near. Otherwise, if `__near` precedes the identifier of the object or function being declared (as in `__near` x), then the object itself is near.

The keyword __near can only be applied to function and object names and the indirection (pointer) symbol *.

> Open Watcom C16 provides the predefined macros near and _near for convenience and compatibility with the Microsoft C compiler. They may be used in place of __near.

The following examples illustrate the use of the __near keyword. These examples assume that the large memory module (big code, big data) is being used.

```
extern int __near * x;
```
declares the object x to be a pointer to a near integer. (x is not necessarily within the 64K data area that is near, but the integer that it points to is.)

```
extern int * __near nx;
```
declares the object nx to be near, and is a pointer to a far integer. (nx is within the 64K data area that is near, but the integer that it points to might not be.)

```
extern int __near * __near nnx;
```
declares the object nnx to be near, and is a pointer to a near integer. (nnx and the integer that it points to are both within the 64K data area that is near.)

```
struct symbol * __near NSymAlloc( void );
```
declares the function NSymAlloc to be near, and returns a pointer to a far symbol structure.

```
struct symbol __near * __near NNSymAlloc( void );
```
declares the function NNSymAlloc to be near, and returns a pointer to a near symbol structure.

7.3.1.6 The __huge Keyword for Open Watcom C16

Even using the big data model, each object is restricted in size to 64K. Some applications will need to get beyond this limitation. The Open Watcom C16 compiler provides the keyword __huge to describe those objects that exceed 64K in size. The code generated for these objects is less efficient than for __far objects.

The declaration of such objects follows the same pattern as above, with the keyword __huge preceding the name of the object if the object itself is bigger than 64K, or preceding the * if the pointer is to an object that is bigger than 64K.

The keyword __huge can only be applied to arrays. Huge objects may be used in both the small and big data models.

> Open Watcom C16 provides the predefined macros huge and _huge for convenience and compatibility with the Microsoft C compiler. They may be used in place of __huge.

These examples illustrate the use of the __huge keyword. They assume that big code, small data (the medium memory model) is in effect.

```
int __huge iarray[50000];
```
declares the object iarray to be an array of 50000 integers, for a total size of 100000 bytes.

```
int __huge * iptr;
```
declares iptr to be near, and a pointer to an integer that is part of a huge array, such as an element of iarray.

7.3.2 Special Pointer Types for Open Watcom C32

With an 80386 processor in "protect" mode, a normal pointer (32 bits) can point to a 4 gigabyte (4,294,967,296 byte) region of the memory available on the machine. (In practice, memory limits may mean that these regions will be smaller than 4 gigabytes.) These regions are called *segments,* and there may be more than one segment defined for the memory. Each 32-bit pointer is actually an offset within a 4 gigabyte segment, and the offsets within two different segments are generally not related to each other in a known manner.

As an example, the screen memory may be set up so that it resides in a different region of the memory from the program's data. Normal pointers (those within the program's data area) will not be able to access such regions.

Like the 16-bit version of Open Watcom C (for the 8086 and 80286), Open Watcom C32 uses the `__near` and `__far` keywords to describe objects that are either in the normal data space or elsewhere.

Objects or functions that are near require a 32-bit pointer to access them.

Objects or functions that are far require a 48-bit pointer to access them. This 48-bit pointer consists of two parts: a *selector* consisting of 16 bits, and an *offset* consisting of 32 bits. A selector is similar to a segment in a 16-bit program's far pointer, except that the numeric value of the selector does not directly determine the memory region. Instead, the processor uses the selector value in conjunction with a "descriptor table" to determine what region of memory is to be accessed. In the discussion of far pointers on the 80386, the terms selector and segment may be used interchangeably.

Like the 16-bit compiler, the Open Watcom C32 compiler supports the small, compact, medium and large memory models. Throughout the discussions in the following sections, it is assumed that the small memory model is being used, since it is the most likely to be used.

7.3.2.1 The __far Keyword for Open Watcom C32

The `__far` keyword is a type qualifier that modifies the token that follows it. If `__far` precedes `*` (as in `__far *`), then the pointer points to something that is far (not in the normal data region). Otherwise, if `__far` precedes the identifier of the object or function being declared (as in `__far x`), then the object or function is far.

The keyword `__far` can only be applied to function and object names and the indirection (pointer) symbol `*`. Parameters to functions may *not* be declared as `__far`, since they are always in the normal data region.

These examples illustrate the use of the `__far` keyword, and assume that the small memory model is being used.

```
int __far * ptr;
```
declares `ptr` to be a pointer to an integer. The object `ptr` is near but the integer that it points to is far.

```
int * __far fptr;
```
also declares `fptr` to be a pointer to an integer. However, the object `fptr` is far, but the integer that it points to is near.

```
int __far * __far ffptr;
```
declares `ffptr` to be a pointer (which is far) to an integer (which is far).

When declaring a function, placing the keyword `__far` in front of the function name causes the compiler to treat the function as being far. It is important, if the function is called before its definition, that a *function prototype* be included prior to any calls. For example, the declaration,

```
extern void __far SystemService();
```

declares the function `SystemService` to be far, meaning that any calls to it must be far calls.

Here are a few more examples:

```
extern struct systbl * __far FSysTblPtr( void );
```
declares the function `FSysTblPtr` to be far, returning a pointer to a near `systbl` structure.

```
extern struct systbl __far * __far FFSysTblPtr( void );
```
declares the function `FFSysTblPtr` to be far, returning a pointer to a far `systbl` structure.

```
extern void Indirect( char __far fn() );
```
declares the function `Indirect` to be near, taking one parameter `fn` which is a pointer to a far function that returns a `char`.

```
extern int StoreSysTbl( struct systbl * __far sysptr );
```
is an invalid declaration, since it attempts to declare `sysptr` to be far. All parameters must be near, since they reside in the normal data area that is always near.

7.3.2.2 The __near Keyword for Open Watcom C32

The `__near` keyword is a type qualifier that modifies the token that follows it. If `__near` precedes `*` (as in `__near` *), then the pointer points to something that is near (in the normal data region). Otherwise, if `__near` precedes the identifier of the object or function being declared (as in `__near` x), then the object or function is near.

The keyword `__near` can only be applied to function and object names and the indirection (pointer) symbol `*`.

For programmers using the small memory model, the `__near` keyword is not required, but may be useful for making the program more readable.

7.3.2.3 The __far16 and _Seg16 Keywords

With the 80386 processor, a far pointer consists of a 16-bit selector and a 32-bit offset. Open Watcom C32 also supports a special kind of far pointer which consists of a 16-bit selector and a 16-bit offset. These pointers, referred to as *far16* pointers, allow 32-bit code to access code and data running in 16-bit mode.

In the OS/2 operating system (version 2.0 or higher), the first 512 megabytes of the 4 gigabyte segment referenced by the DS register is divided into 8192 areas of 64K bytes each. A far16 pointer consists of a 16-bit selector referring to one of the 64K byte areas, and a 16-bit offset into that area.

For compatibility with Microsoft C, Open Watcom C32 provides the `__far16` keyword. A pointer declared as,

type `__far16 *` *name*`;`

defines an object that is a far16 pointer. If such a pointer is accessed in the 32-bit environment, the compiler will generate the necessary code to convert between the far16 pointer and a "flat" 32-bit pointer.

For example, the declaration,

```
char __far16 * bufptr;
```

declares the object `bufptr` to be a far16 pointer to `char`.

A function declared as,

type `__far16` *func* `(` *parm-list* `);`

declares a 16-bit function. Any calls to such a function from the 32-bit environment will cause the compiler to convert any 32-bit pointer parameters to far16 pointers, and any `int` parameters from 32 bits to 16 bits. (In the 16-bit environment, an object of type `int` is only 16 bits.) Any return value from the function will have its return value converted in an appropriate manner.

For example, the declaration,

```
char * __far16 Scan( char * buffer, int buflen, short err );
```

declares the 16-bit function `Scan`. When this function is called from the 32-bit environment, the `buffer` parameter will be converted from a flat 32-bit pointer to a far16 pointer (which, in the 16-bit environment, would be declared as `char __far *`). The `buflen` parameter will be converted from a 32-bit integer to a 16-bit integer. The `err` parameter will be passed unchanged. Upon returning, the far16 pointer (far pointer in the 16-bit environment) will be converted to a 32-bit pointer which describes the equivalent location in the 32-bit address space.

For compatibility with IBM C Set/2, Open Watcom C^{32} provides the `_Seg16` keyword. Note that `_Seg16` is **not** interchangeable with `__far16`.

A pointer declared as,

type `*` `_Seg16` *name*`;`

defines an object that is a far16 pointer. Note that the `_Seg16` appears on the opposite side of the `*` than the `__far16` keyword described above.

For example,

```
char * _Seg16 bufptr;
```

declares the object `bufptr` to be a far16 pointer to `char` (the same as above).

The `_Seg16` keyword may not be used to describe a 16-bit function. A `#pragma` directive must be used. See the User's Guide for details. A function declared as,

type * _Seg16 *func* (*parm-list*);

declares a 32-bit function that returns a far16 pointer.

For example, the declaration,

```
char * _Seg16 Scan( char * buffer, int buflen, short err );
```

declares the 32-bit function Scan. No conversion of the parameter list will take place. The return value is a far16 pointer.

7.3.3 Based Pointers for Open Watcom C16 and C32

Near pointers are generally the most efficient type of pointer because they are small, and the compiler can assume knowledge about what segment of the computer's memory the pointer (offset) refers to. Far pointers are the most flexible because they allow the programmer to access any part of the computer's memory, without limitation to a particular segment. However, far pointers are bigger and slower because of the additional flexibility.

Based pointers are a compromise between the efficiency of near pointers and the flexibility of far pointers. With based pointers, the programmer takes responsibility to tell the compiler which segment a near pointer (offset) belongs to, but may still access segments of the computer's memory outside of the normal data segment (DGROUP). The result is a pointer type which is as small as and almost as efficient as a near pointer, but with most of the flexibility of a far pointer.

An object declared as a based pointer falls into one of the following categories:

- the based pointer is in the segment described by another object,
- the based pointer, used as a pointer to another object of the same type (as in a linked list), refers to the same segment,
- the based pointer is an offset to no particular segment, and must be combined explicitly with a segment value to produce a valid pointer.

To support based pointers, the following keywords are provided:

```
__based
__segment
__segname
__self
```

The following operator is also provided:

```
:>
```

These keywords and operator are described in the following sections.

Two macros, defined in <malloc.h> are also provided:

```
_NULLSEG
_NULLOFF
```

They are used in a similar manner to NULL, but are used with objects declared as __segment and __based respectively.

7.3.3.1 Segment Constant Based Pointers and Objects

A segment constant based pointer or object has its segment value based on a specific, named segment. A segment constant based object is specified as:

```
type __based( __segname( "segment" ) )  object-name;
```

and a segment constant based pointer is specified as:

```
type __based( __segname( "segment" ) ) *  object-name;
```

where *segment* is the name of the segment in which the pointer or object is based. As shown above, the segment name is always specified as a string. There are three special segment names recognized by the compiler:

```
"_CODE"
"_CONST"
"_DATA"
```

The `"_CODE"` segment is the default code segment. The `"_CONST"` segment is the segment containing constant values. The `"_DATA"` segment is the default data segment. If the segment name is not one of the three recognized names, then a segment will be created with that name. If a segment constant based object is being defined, then it will be placed in the named segment. If a segment constant based pointer is being defined, then it can point at objects in the named segment.

The following examples illustrate segment constant based pointers and objects:

```
int __based( __segname( "_CODE" ) )   ival = 3;
int __based( __segname( "_CODE" ) ) * iptr;
```

`ival` is an object that resides in the default code segment. `iptr` is an object that resides in the data segment (the usual place for data objects), but points at an integer which resides in the default code segment. `iptr` is suitable for pointing at `ival`.

```
char __based( __segname( "GOODTHINGS" ) ) thing;
```

`thing` is an object which resides in the segment `GOODTHINGS`, which will be created if it does not already exist. (The creation of segments is done by the linker, and is a method of grouping objects and functions. Nothing is implicitly created during the execution of the program.)

7.3.3.2 Segment Object Based Pointers

A segment object based pointer derives its segment value from another named object. A segment object based pointer is specified as follows:

type `__based(` *segment* `) *` *name*`;`

where *segment* is an object defined as type `__segment`.

An object of type `__segment` may contain a segment value. Such an object is particularly designed for use with segment object based pointers.

The following example illustrates a segment object based pointer:

```
__segment               seg;
char __based( seg ) * cptr;
```

The object `seg` contains only a segment value. Whenever the object `cptr` is used to point to a character, the actual pointer value will be made up of the segment value found in `seg` and the offset value found in `cptr`. The object `seg` might be assigned values such as the following:

- a constant value (eg. the segment containing screen memory),
- the result of the library function `_bheapseg`,
- the segment portion of another pointer value, by casting it to the type `__segment`.

7.3.3.3 Void Based Pointers

A void based pointer must be explicitly combined with a segment value to produce a reference to a memory location. A void based pointer does not infer its segment value from another object. The `:>` (base) operator is used to combine a segment value and a void based pointer.

For example, on an IBM PC or PS/2 computer, running DOS, with a color monitor, the screen memory begins at segment 0xB800, offset 0. In a video text mode, to examine the first character currently displayed on the screen, the following code could be used:

```
extern void main()
  {
    __segment              screen;
    char __based( void ) * scrptr;

    screen = 0xB800;
    scrptr = 0;
    printf( "Top left character is '%c'.\n",
            *(screen:>scrptr) );
  }
```

The general form of the `:>` operator is:

segment `:>` *offset*

where *segment* is an expression of type `__segment`, and *offset* is an expression of type `__based( void ) *`.

7.3.3.4 Self Based Pointers

A self based pointer infers its segment value from itself. It is particularly useful for structures such as linked lists, where all of the list elements are in the same segment. A self based pointer pointing to one element may be used to access the next element, and the compiler will use the same segment as the original pointer.

The following example illustrates a function which will print the values stored in the last two members of a linked list:

```
struct a {
    struct a __based( __self ) * next;
    int                          number;
};
```

```
extern void PrintLastTwo( struct a far * list )
  {
    __segment                 seg;
    struct a __based( seg ) * aptr;

    seg  = FP_SEG( list );
    aptr = FP_OFF( list );
    for( ; aptr != _NULLOFF; aptr = aptr->next ) {
        if( aptr->next == _NULLOFF ) {
            printf( "Last item is %d\n", aptr->number );
        } else if( aptr->next->next == _NULLOFF ) {
            printf( "Second last item is %d\n", aptr->number );
        }
    }
  }
```

The parameter to the function `PrintLastTwo` is a far pointer, pointing to a linked list structure anywhere in memory. It is assumed that all members of a particular linked list of this type reside in the same segment of the computer's memory. (Another instance of the linked list might reside entirely in a different segment.) The object `seg` is given the segment portion of the far pointer. The object `aptr` is given the offset portion, and is described as being based in the segment stored in `seg.`

The expression `aptr->next` refers to the `next` member of the structure stored in memory at the offset stored in `aptr` and the segment implied by `aptr`, which is the value stored in `seg.` So far, the behavior is no different than if `next` had been declared as,

```
struct a * next;
```

The expression `aptr->next->next` illustrates the difference of using a self based pointer. The first part of the expression (`aptr->next`) occurs as described above. However, using the result to point to the next member occurs by using the offset value found in the `next` member and combining it with the segment value of the *pointer used to get to that member*, which is still the segment implied by `aptr`, which is the value stored in `seg.` If `next` had not been declared using `__based( __self )`, then the second pointing operation would refer to the offset value found in the `next` member, but with the default data segment (DGROUP), which may or may not be the same segment as stored in `seg.`

7.4 Void

The `void` type has several purposes:

1. To declare an object as being a pointer to no particular type. For example,

   ```
   void * membegin;
   ```

   defines `membegin` as being a pointer. It does not point to anything without a *cast* operator. The statement,

   ```
   *(char *) membegin = '\0';
   ```

   will place a zero in the character at which `membegin` points.

2. To declare a function as not returning a value. For example,

   ```
   void rewind( FILE * stream );
   ```

   declares the standard library function `rewind` which takes one parameter and returns nothing.

3. To evaluate an expression for its side-effects, discarding the result of the expression. For example,

```
(void) getchar();
```

calls the library function `getchar`, which normally returns a character. In this case, the character is discarded, effectively advancing one character in the file without caring what character is read. This use of `void` is primarily for readability, because casting the expression to the void type will be done automatically. The above example could also be written as,

```
getchar();
```

The keyword `void` is also used in one other instance. If a function takes no parameters, `void` may be used in the declaration. For example,

```
int getchar( void );
```

declares the standard library function `getchar`, which takes no parameters and returns an integer.

No object (other than a function) may be declared with type `void`.

7.5 The const and volatile Declarations

An object may be declared with the keyword `const`. Such an object may not be modified directly by the program. For objects with static storage duration, this type qualifier describes to the compiler which objects may be placed in read-only memory, if the computer supports such a concept. It also provides the opportunity for the compiler to detect attempts to modify the object. The compiler may also generate better code when it knows that an object will not be modified.

Even though an object is declared to be constant, it is possible to modify its value indirectly by storing its address (using a cast) in another object declared to be a pointer to the same type (without the `const`), and then using the second object to modify the value to which it points. However, this should be done with caution, and may fail on computers with protected memory.

If the declaration of an object does not include `*`, that is to say it is not a pointer of any kind, then the keyword `const` appearing anywhere in the type specifier (including any `typedef`'s) indicates that the object is constant and may not be changed. If the object is a pointer and `const` appears to the left of the `*`, the object is a pointer to a constant value, meaning that the value to which the pointer points may not be modified, although the pointer value may be changed. If `const` appears to the right of the `*`, the object is a constant pointer to a value, meaning that the pointer to the value may not be changed, although what the pointer points to may be changed. If `const` appears on both sides of the `*`, the object is a constant pointer to a constant value, meaning that the pointer and the object to which it points may not be changed.

If the declaration of a structure, union or array includes `const`, then each member of the type, when referred to, is treated as if `const` had been specified.

The declarations,

```
const int   baseyear = 1900;
const int * byptr;
```

declare the object `baseyear` to be an integer whose value is constant and set to 1900, and the object `byptr` to be a pointer to a constant object of integer type. If `byptr` was made to point to another integer that was not, in fact, declared to be constant, then `byptr` could not be used to modify that value. `byptr` may be used to get a value from an integer object, and never to change it. Another way of stating it is that what `byptr` points to is constant, but `byptr` itself is not constant.

The declarations,

```
int         baseyear;
int * const byptr = &baseyear;
```

declare the object `byptr` as a constant pointer to an integer, in this case the object `baseyear.` The value of `baseyear` may be modified via `byptr`, but the value of `byptr` itself may not be changed. In this case, `byptr` itself is constant, but what `byptr` points to is not constant.

An object may be declared with the keyword `volatile`. Such an object may be freely modified by the program, and its value also may be modified through actions outside the program. For example, a flag may be set when a given interrupt occurs. The keyword `volatile` indicates to the compiler that care must be taken when optimizing code referring to the object, so that the meaning of the program is not altered. An object that the compiler might otherwise have been able to keep in a register for an extended period of time will be forced to reside in normal storage so that an external change to it will be reflected in the program's behavior.

If the declaration of an object does not include `*`, that is to say it is not a pointer of any kind, then the keyword `volatile` appearing anywhere in the type specifier (including any `typedef`'s) indicates that the object is volatile and may be changed at any time without the program knowing. If the object is a pointer and `volatile` appears to the left of the `*`, the object is a pointer to a volatile value, meaning that the value to which the pointer points may be changed at any time. If `volatile` appears to the right of the `*`, the object is a volatile pointer to a value, meaning that the pointer to the value may be changed at any time. If `volatile` appears on both the left and the right of the `*`, the object is a volatile pointer to a volatile value, meaning that the pointer or the value to which it points may be changed at any time.

If the declaration of a structure, union or array includes `volatile`, then each member of the type, when referred to, is treated as if `volatile` had been specified.

The declarations,

```
volatile int   attncount;
volatile int * acptr;
```

declare the object `attncount` to be an integer whose value may be altered at any time (say by an asynchronous attention handler), and the object `acptr` to be a pointer to a volatile object of integer type.

If both `const` and `volatile` are included in the declaration of an object, then that object may not be modified by the program, but it may be modified through some external action. An example of such an object is the clock in a computer, which is modified periodically (every clock "tick"), but programs are not allowed to change it.

8 Storage Classes

The *storage class* of an object describes:

- the duration of the existence of the object. An object may exist throughout the execution of the program, or only during the span of time that the function in which it is defined is executing. In the latter case, each time the function is called, a new instance of the object is created, and that object is destroyed when the function returns.

- the *scope* of the object. An object may be declared so that it is only accessible within the function in which it is defined, within the module or throughout the entire program.

A *storage class specifier* is one of:

```
auto
register
extern
static
typedef
```

`typedef` is included in the list of storage class specifiers for convenience, because the syntax of a type definition is the same as for an object declaration. A `typedef` declaration does not create an object, only a synonym for a type, which does not have a storage class associated with it.

Only one of these keywords (excluding `typedef`) may be specified in a declaration of an object.

If an object or function is declared with a storage class, but no type specifier, then the type of the object or function is assumed to be `int`.

While a storage class specifier may be placed following a type specifier, this tends to be difficult to read. It is recommended that the storage class (if present) always be placed first in the declaration. The ISO C standard states that the ability to place the storage class specifier other than at the beginning of the declaration is an obsolescent feature.

8.1 Type Definitions

A *typedef* declaration introduces a synonym for another type. It does not introduce a new type.

The general form of a type definition is:

`typedef` *type-information typedef-name*`;`

The *typedef-name* may be a comma-separated list of identifiers, all of which become synonyms for the type. The names are in the same name space as ordinary object names, and can be redefined in inner blocks. However, this can be confusing and should be avoided.

The simple declaration,

```
typedef signed int COUNTER;
```

declares the identifier `COUNTER` to be equivalent to the type `signed int`. A subsequent declaration like,

```
COUNTER ctr;
```

declares the object `ctr` to be a signed integer. If, later on, it is necessary to change all counters to be long signed integers, then only the `typedef` would have to be changed, as follows:

```
typedef long signed int COUNTER;
```

All declarations of objects of that type will use the new type.

The `typedef` can be used to simplify declarations elsewhere in a program. For example, consider the following structure:

```
struct complex {
    double    real;
    double    imaginary;
};
```

To declare an object to be an instance of the structure requires the following declaration:

```
struct complex cnum;
```

Now consider the following structure definition with a type definition:

```
typedef struct {
    double   real;
    double   imaginary;
} COMPLEX;
```

In this case, the identifier `COMPLEX` refers to the entire structure definition, including the keyword `struct`. Therefore, an object can be declared as follows:

```
COMPLEX cnum;
```

While this is a simple example, it illustrates a method of making object declarations more readable.

Consider the following example, where the object `fnptr` is being declared as a pointer to a function which takes two parameters, a pointer to a structure `dim3` and an integer. The function returns a pointer to the structure `dim3`. The declarations could appear as follows:

```
struct dim3 {
    int  x;
    int  y;
    int  z;
};

struct dim3 * (*fnptr)( struct dim3 *, int );
```

or as:

```
typedef struct {
    int  x;
    int  y;
    int  z;
} DIM3;

DIM3 * (*fnptr)( DIM3 *, int );
```

or as:

```
typedef struct {
    int  x;
    int  y;
    int  z;
} DIM3;

typedef DIM3 * DIM3FN( DIM3 *, int );

DIM3FN * fnptr;
```

The last example simply declares `fnptr` to be a pointer to a `DIM3FN`, while `DIM3FN` is declared to be a function with two parameters, a pointer to a `DIM3` and an integer. The function returns a pointer to a `DIM3`. `DIM3` is declared to be a structure of three co-ordinates.

8.1.1 Compatible Types

Some operations, such as assignment, are restricted to operating on two objects of the same type. If both operands are already the same type, then no special conversion is required. Otherwise, the compiler may alter automatically one or both operands to make them the same type. The integral promotions and arithmetic conversions are examples. Other types may require an explicit cast.

The compiler decides whether or not an explicit cast is required based on the concept of *compatible types.* The following types are compatible:

- two types that are declared exactly the same way,
- two types that differ only in the ordering of the type specifiers, for example, `unsigned long int` and `int long unsigned`,
- two arrays of members of compatible type, where both arrays have the same size, or where one array is declared without size information,

- two functions that return the same type, one containing no parameter information, and the other containing a fixed number of parameters (no "`, ...`") that are not affected by the default argument promotions,

- two structures, defined in separate modules, that have the same number and names of members, in the same order, with compatible types,

- two unions, defined in separate modules, that have the same number and names of members, with compatible types,

- two enumerated types, defined in separate modules, that have the same number of enumeration constants, with the same names and the same values,

- two pointers to compatible types.

8.2 Static Storage Duration

An object with *static storage duration* is created and initialized only once, prior to the execution of the program. Any value stored in such an object is retained throughout the program unless it is explicitly altered by the program (or it is declared with the `volatile` keyword).

Any object that is declared outside the scope of a function has static storage duration.

There are three types of static objects:

1. objects whose values are only available within the function in which they are defined (no linkage). For example,

```
extern void Fn( int x )
{
    static int ObjCount;
/* ... */
}
```

2. objects whose values are only available within the module in which they are defined (internal linkage). For example,

```
static int ObjCount;

extern void Fn( int x )
{
/* ... */
}
```

3. objects whose values are available to all components of the program (external linkage). For example,

```
extern int ObjCount = { 0 };

extern void Fn( int x )
{
/* ... */
}
```

The first two types are defined with the keyword `static`, while the third is defined with the (optional) keyword `extern`.

8.2.1 The static Storage Class

Any declaration of an object may be preceded by the keyword `static`. A declaration inside a function indicates to the compiler that the object has *no linkage,* meaning that it is available only within the function. A declaration not inside any function indicates to the compiler that this object has *internal linkage,* meaning that it is available in all functions within the module in which it is defined. Other modules may not refer to the specific object. They may have their own object defined with the same name, but this is a questionable programming practice and should be avoided.

The value of the object will be preserved between function calls. Any value placed in an object with static storage duration will remain unchanged until changed by a function within the same module. It is also possible for a pointer to the object to be passed to a function outside the module in which the object is defined. This pointer could be used to modify the value of the object.

8.2.2 The extern Storage Class

If an object is declared with the keyword `extern` inside a function, then the object has *external linkage,* meaning that its value is available to all modules, and to the function(s) containing the definition in the current module. No initializer list may be specified in this case, which implies that the space for the object is allocated in some other module.

If an object is declared outside of the definition of a function, and the declaration does not contain either of the keywords `static` or `extern`, then the space for the object is created at this point. The object has *external linkage,* meaning that it is available to other modules in the program.

The following examples illustrate the creation of external objects, provided the declarations occur outside any function:

```
int   X;
float F;
```

If the declaration for an object, outside of the definition of a function, contains the keyword `extern` and has an initializer list, then space for the object is created at this point, and the object has external linkage. If, however, the declaration does not include an initializer list, then the compiler assumes that the object is declared elsewhere. If, during the remainder of the compilation of the module, no further declarations of the object are found, or more declarations with `extern` and no initializer list are found, then the object must have space allocated for it in another module. If a subsequent declaration in the same module does have an initializer list or omits the `extern` keyword, then the space for the object is created at that point.

The following examples also illustrate the creation of external objects:

```
extern LIST * ListHead = 0;
       int    StartVal = 77;
```

However, the next examples illustrate the *tentative definition* of external objects. If no further definition of the object of a form shown above is found, then the object is found outside of the module.

```
extern LIST * ListEl;
extern int    Z;
```

Another module may define its own object with the same name (provided it has *static* storage class), but it will not be able to access the external one. However, this can be confusing and is a questionable programming practice.

Any value placed in an object declared with the `extern` keyword will remain unchanged until changed by a function within the same or another module.

A function that is declared without the keyword `static` has external linkage.

Suppose a module declares an object (outside of any function definition) as follows:

```
struct list_el * ListTop;
```

where the structure `list_el` is defined elsewhere. This declaration allocates space for and declares the object `ListTop` to be a pointer to a structure `list_el`, with external linkage. Another module with the declaration,

```
extern struct list_el * ListTop;
```

refers to the same object `ListTop`, and states that it is found outside of the module.

Within a program, possibly consisting of more than one module, each object or function with external linkage must be defined (have space allocated for it) exactly once.

8.3 Automatic Storage Duration

The most commonly used object in a C program is one that has meaning only within the function in which it is defined. The object is created when execution of the function is begun and destroyed when execution of the function is completed. Such an object is said to have *automatic storage duration.* The *scope* of the object is said to be the function in which it is defined.

If such an object has the same name as another object defined outside the function (using `static` or `extern`), then the outside object is hidden from the function.

Within a function, any object that does not have its declaration preceded by the keyword `static` or `extern` has automatic storage duration.

It is possible to declare an object as automatic within any block of a function. The *scope* of such an object is the block in which it is declared, including any blocks inside it. Any outside block is unable to access such an object.

Automatic objects may be initialized as described in the chapter "Initialization of Objects". Initialization of the object only occurs when the block in which the object is declared is entered normally. In particular, a jump into a block nested within the function will **not** initialize any objects declared in that block. This is a questionable programming practice, and should be avoided.

The following function checks a string to see if it contains nothing but digits:

```
extern int IsInt( const char * ptr )
/*********************************/
{
    if( *ptr == '\0' ) return( 0 );
    for( ;; ) {
        char  ch;

        ch = *(ptr++);
        if( ch == '\0' )     return( 1 );
        if( !isdigit( ch ) ) return( 0 );
    }
}
```

The object `ch` has a scope consisting only of the `for` loop. Any statements before or after the loop cannot access `ch`.

8.3.1 The auto Storage Class

The declaration of an object in a function that does not contain the keywords `static`, `extern` or `register` declares an object with automatic storage duration. Such an object may precede its declaration with the keyword `auto` for readability.

An object declared with no storage class specifier or with `auto` is "addressable", which means that the *address-of* operator may be applied to it.

The programmer should not assume any relationship between the storage locations of multiple `auto` objects declared in a function. If relative placement of objects is important, a structure should be used.

The following function illustrates a use for `auto` objects:

```
extern int FindSize( struct thing * thingptr )
/*********************************************/
{
    auto char * start;
    auto char * finish;

    FindEnds( thingptr, &start, &finish );
    return( finish - start + 1 );
}
```

The addresses of the automatic objects `start` and `finish` are passed to `FindEnds`, which, presumably, modifies them.

8.3.2 The register Storage Class

An object that is declared within a function, and whose declaration includes the keyword `register`, is considered to have automatic storage duration. The `register` keyword merely provides a hint to the compiler that this object is going to be heavily used, allowing the compiler to try to put it into a high-speed access part of the machine, such as a machine register. The compiler may, however, ignore such a directive for any number of reasons, such as,

- the compiler does not support objects in registers,
- there are no available registers, or,
- the compiler makes its own decisions about register usage.

Only certain types of objects may be placed in registers, although the set of such types is implementation-defined.

The Open Watcom C16 and C32 compilers may place any object that is sufficiently small, including a small structure, in one or more registers.

The compiler will decide which objects will be placed in registers. The `register` keyword is ignored, except to prevent taking the address of such an object.

Objects declared with or without `register` may generally be treated in the same way. An exception to this rule is that the *address-of* operator `(&)` may not be applied to a `register` object, since registers are generally not within the normal storage of the computer.

9 Initialization of Objects

Any definition of an object may include a value or list of values for initializing it, in which case the declaration is followed by an equal sign (=) and the initial value(s).

The initial value for an object with static storage duration may be any expression that evaluates to a constant value, including using the *address-of* operator to take the address of a function or object with static storage duration.

The initial value for an object with automatic storage duration may be any expression that would be valid as an assignment to that object, including references to other objects. The evaluations of the initializations occur in the order in which the definitions of the objects occur.

9.1 Initialization of Scalar Types

The initial value for a scalar type (pointers, integers and floating-point types) may be enclosed in braces, although braces are not required.

The following declarations might appear inside a function:

```
static   int   MaxRecLen  = 1000;
static   int   MaxMemSize = { 1000 * 8 + 10000 };
         float Pi         = 3.14159;
auto     int   x          = 3;
register int   y          = x * MaxRecLen;
```

9.2 Initialization of Arrays

For arrays of characters being initialized with a string literal, and for arrays of `wchar_t` being initialized with a wide string literal, the braces around initial values are optional. For other arrays, the braces are required.

If an array of unknown size is initialized, then the size of the array is determined by the number of initializing values provided. In particular, an array of characters of unknown size may be initialized using a string literal, in which case the size of the array is the number of characters in the string, plus one for the terminating null character. Each character of the string is placed in successive elements of the array. Consider the following array declarations:

```
char  StartPt[] = "Starting point...";
int   Tabs[]    = { 1, 9, 17, 25, 33, 41 };
float Roots[]   = { 1., 1.414, 1.732, 2., 2.236 };
```

The object `StartPt` is an array of 18 characters, `Tabs` is an array of 6 integers, and `Roots` is an array of 5 floating-point numbers.

If an array is declared to have a certain number of elements, then the maximum number of values in the initialization list is the number of elements in the array. An exception is made for arrays of characters, where the initializer may be a string with the same length as the number of characters in the array. Each character from the string is assigned to the corresponding element of the array. The null character at the end of the string literal is ignored.

If there are fewer initialization values than elements of the array, then any elements not receiving a value from the list are assigned the value zero (for arithmetic types), or the null pointer constant (for pointers). Consider the following examples:

```
char  Vowels1[6]  = "aeiouy";
char  Vowels2[6]  = { 'a', 'e', 'i', 'o', 'u', 'y' };
int   Numbers[10] = { 100, 10, 1 };
float Blort[5]    = { 5.6, -2.2 };
```

The objects `Vowels1` and `Vowels2` are both arrays of six characters, and both contain exactly the same values in each of their corresponding elements. The object `Numbers` is an array of 10 integers, the first three of which are initialized to `100`, `10` and `1`, and the remaining seven are set to zero. The object `Blort` is an array of 5 floating-point numbers. The first two elements are initialized to `5.6` and `-2.2`, and the remaining three are set to zero.

If an array of more than one dimension is initialized, then each subarray may be initialized using a brace-enclosed list of values. This form will work for an arbitrary number of dimensions. Consider the following two-dimensional case:

```
int Box[3][4] = { { 11, 12, 13, 14 },
                  { 21, 22, 23, 24 },
                  { 31, 32, 33, 34 } };
```

The object `Box` is an array of 3 arrays of 4 integers. There are three values in the initialization list, corresponding to the first dimension (3 rows). Each initialization value is itself a list of values corresponding to the second dimension (4 columns). In other words, the first list of values `{ 11, 12, 13, 14 }` is assigned to the first row of `Box`, the second list of values `{ 21, 22, 23, 24 }` is assigned to the second row of `Box`, and the third list of values `{ 31, 32, 33, 34 }` is assigned to the third row of `Box.`

If all values are supplied for initializing an array, or if only elements from the end of the array are omitted, then the sub-levels need not be within braces. For example, the following declaration of `Box` is the same as above:

```
int Box[3][4] = { 11, 12, 13, 14,
                  21, 22, 23, 24,
                  31, 32, 33, 34 };
```

The same rules about incomplete initialization lists apply to multi-dimensional arrays. The following example defines a mathematical 3-by-3 identity matrix:

```
int Identity[3][3] = { { 1 },
                       { 0, 1 },
                       { 0, 0, 1 } };
```

The missing values are replaced with zeroes. The initialization also could have been given as,

```
int Identity[3][3] = { { 1, 0, 0 },
                       { 0, 1, 0 },
                       { 0, 0, 1 } };
```

or as,

```
int Identity[3][3] = { 1, 0, 0,
                       0, 1, 0,
                       0, 0, 1 };
```

9.3 Initialization of Structures

Structures may be initialized in a manner similar to arrays. The initializer list must be specified within braces.

For example,

```
struct printformat {
    int     pagewid;
    char    carr_ctl;
    char *  buffer;
};

char PrBuffer[256];

struct printformat PrtFmt = { 80, ' ', PrBuffer };
```

Each value from the initializer list is assigned to each successive member of the structure. Any unnamed gaps between members or at the end of the structure (caused by alignment) are ignored during initialization. If there are more members of the structure than values specified by the initializer list, then the remaining members are initialized to zero (for arithmetic types) or the null pointer constant (for pointers).

If a structure member is itself an array, structure or union, then the sub-members may be initialized using a brace-enclosed initializer list. If braces are not specified, then for the purposes of initialization, the sub-members are treated as if they are members of the outer structure, as each subsequent initializer value initializes a sub-member, until no more sub-members are found, in which case the next member of the outer structure is initialized.

9.4 Initialization of Unions

Initializations of unions is the same as for structures, except that only the first member of the union may be initialized, using a brace-enclosed initializer.

Consider the following example:

```
struct first3 {
    char first, second, third;
};

union ustr {
    char           string[20];
    struct first3  firstthree;
};
union ustr Str = { "Hello there" };
```

The object `Str` is declared to be a union of two types, the first of which is an array of 20 characters, and the second of which is a structure that allows direct access to the first three characters of the string contained in the array. The array is initialized to the string `"Hello there".` The three characters of `struct first3` will have the characters `'H', 'e'` and `'l'.` Had the declaration of `ustr` been,

```
union ustr {
    struct first3  firstthree;
    char           string[20];
};
```

then the initialization could only set the first three characters.

9.5 Uninitialized Objects

An object with *static* storage duration, and no explicit initialization, will be initialized as if every member that has arithmetic type was assigned zero and every member that has a pointer type was assigned a null (zero) pointer.

An object with *automatic* storage duration, and no explicit initialization, is not initialized. Hence, a reference to such an automatic object that has not been assigned a value will yield undefined behavior. On most systems, the value of the object will be arbitrary and unpredictable.

10 Expressions

An *expression* is a sequence of operators and operands that describes how to,

- calculate a value (eg. addition)
- create side-effects (eg. assignment, increment)

or both.

The order of execution of the expression is usually determined by a mixture of,

1. parentheses `()`, which indicate to the compiler the desired grouping of operations,
2. the precedence of operators, which describes the relative priority of operators in the absence of parentheses,
3. the common algebraic ordering,
4. the associativity of operators.

In most other cases, the order of execution is determined by the compiler and may not be relied upon. Exceptions to this rule are described in the relevant section. Most users will find that the order of execution is well-defined and intuitive. However, when in doubt, use parentheses.

The table below summarizes the levels of precedence in expressions.

Operations at a higher level in the table will occur before those below. All operators involving more than one operand associate from left to right, except for the conditional and assignment operators, which associate from right to left. Operations at the same level, except where discussed in the relevant section, may be executed in any order that the compiler chooses (subject to the usual algebraic rules). In particular, the compiler may regroup sub-expressions that are both associative and commutative in order to improve the efficiency of the code, provided the meaning (i.e. types and results) of the operands and result are not affected by the regrouping.

The order of any side-effects (for example, assignment, or action taken by a function call) is also subject to alteration by the compiler.

An *exception* occurs when the operands for an operator are invalid. For example, division by zero may cause an exception. If an exception occurs, the behavior is undefined. If an exception is a possibility, the program should be prepared to handle it.

In the following sections, a formal syntax is used to describe each level in the precedence table. This syntax is used in order to completely describe the relationships between the various levels.

Expression Type	Operators
primary	identifier constant string (expression)
postfix	`a[b] f() a.b a->b a++ a--`
unary	`sizeof u sizeof( a ) ++a --a &a *a +a -a ~a !a`
cast	`(type) a`
multiplicative	`a * b a / b a % b`
additive	`a + b a - b`
shift	`a << b a >> b`
relational	`a < b a > b a <= b a >= b`
equality	`a == b a != b`
bitwise AND	`a & b`
bitwise exclusive OR	`a ^ b`
bitwise inclusive OR	`a | b`
logical AND	`a && b`
logical OR	`a || b`
conditional †	`a ? b : c`
assignment †	`a = b a += b a -= b a *= b a /= b a %= b a &= b a ^= b a |= b a <<= b a >>= b`
comma	`a,b`

† associates from right to left

10.1 Lvalues

In order to understand certain components of expressions, it is important to understand the term *lvalue*.

An *lvalue* is an expression that designates an object. The simplest form of *lvalue* is an identifier which is an object (for example, an integer).

The type of the expression may not be `void` or a function. The term *lvalue* is derived from *l*eft *value,* which refers to the fact that an *lvalue* is typically on the left side of an assignment expression.

If `ptr` is a pointer to a type other than `void` or a function, then both `ptr` and `*ptr` are *lvalues.*

A *modifiable lvalue* is an *lvalue* whose type is not an array or an incomplete type, whose declaration does not contain the keyword `const`, and, if it is a structure or union, then none of its members contains the keyword `const`.

10.2 Primary Expressions

primary-expression:
identifier
or
constant
or
string-literal
or
(*expression*)

A *primary expression* is the simplest part of an expression. It consists of one of the following:

identifier — An identifier that designates a function is called a *function designator.* An identifier that designates an object is an *lvalue.*

constant — A constant is a primary expression whose type depends on its form. See "Constants".

string-literal — A string literal is a primary expression whose type is "array of `char`". A string literal is also an *lvalue* (but is not modifiable).

expression inside parentheses — The type and value of a parenthesized expression are the same as for the expression without parentheses. It may be an *lvalue,* function designator or void expression.

Given these declarations,

```
int     count;
int *   ctrptr;
int     f( int );
int     g( int );
```

the following are all valid primary expressions:

```
count
3
3.2
'a'
"Hello there"
(count + 3)
(*(ctrptr+1))
(f( ++i ) * g( j++ ))
```

10.3 Postfix Operators

postfix-expression:
primary-expression
or
array-subscripting-expression
or
function-call-expression
or
member-designator-expression
or
post-increment-expression
or
post-decrement-expression

10.3.1 Array Subscripting

array-subscripting-expression:
postfix-expression `[` *expression* `]`

The general form for array subscripting is,

```
array[index]
```

where `array` must have the type "array of *type*" or "pointer to *type*", and `index` must have an integral type. The result has type "*type*".

`array[index]` is equivalent to `(*(array+index))`, or the `index`-th element of the array `array`, where the first element is numbered zero. Note that `index` is scaled automatically to account for the size of the elements of `array`.

An alternate form for array subscripting is,

```
index[array]
```

although this form is not commonly used.

10.3.2 Function Calls

function-call-expression:
postfix-expression `()`
or
postfix-expression `(`*argument-expression-list*`)`

argument-expression-list:
one or more *assignment-expression*s separated by commas

A *postfix-expression* followed by parentheses containing zero or more comma-separated expressions is a *function-call-expression*. The postfix-expression denotes the function to be called, and must evaluate to a pointer to a function. The simplest form of this expression is an identifier which is the name of a function. For example, `Fn()` calls the function `Fn`.

The expressions within the parentheses denote the arguments to the function. If a function prototype has been declared, then the number of arguments must match the parameter list in the prototype, and the arguments are converted to the types specified in the prototype.

If the postfix-expression is simply an identifier, and no function prototype declaration for that identifier is in scope, then an implicit,

```
extern int identifier();
```

declaration is placed in the innermost block containing the function call. This declares the function as having external linkage, no information about its parameters is available, and the function returns an integer.

The expressions are evaluated (in an undefined order) and the values assigned to the parameters for the function. All arguments are passed by value, allowing the function to modify its parameters without affecting the arguments used to create the parameters. However, an argument can be a pointer to an object, in which case the function may modify the object to which the pointer points.

If a function prototype is in scope at both a call to a function and its definition (and if the prototypes are the same), then the compiler will ensure that the required number and type of parameters are present.

If no function prototype is in scope at a call to a function, then the *default argument promotions* are performed. (Integral types such as `char` and `short int` are converted to `int`, while `float` values are converted to `double`.) When the function definition is encountered, if the parameter types do not match the default argument promotions, then the behavior is undefined. (Usually, the parameters to the function will receive incorrect values.)

If a function prototype has been declared at a call to a function, then each argument is converted, as if by assignment, to the type of the corresponding parameter. When the function definition is encountered, if the types of the parameters do not match the types of the parameters in the function prototype, the behavior is undefined.

If the ellipsis (`, ...`) notation is used in a function prototype, then those arguments in a function call that correspond to the ellipsis have only the default argument promotions performed on them. (See the chapter "Functions" for a complete description of the ellipsis notation.)

Function calls may be recursive. Functions may call themselves either directly, or via other functions.

The following are some examples of function calls:

```
putchar( 'x' );
chr = getchar();
valid = isdigit( chr );
printf( "chr = %c, valid = %2x\n", chr, valid );
fnptr = &MyFunction;
(*fnptr)( parm1, parm2 );
fnptr( parm1, parm2 );
```

10.3.3 Structure and Union Members

member-designator-expression:
postfix-expression . identifier
or
postfix-expression−>identifier

The first operand of the . operator must be an object with a structure or union type. The second operand must be the name of a member of that type. The result is the value of the member, and is an lvalue if the first operand is also an lvalue.

The first operand of the −> operator must be a pointer to an object with a structure or union type. The second operand must be the name of a member of that type. The result is the value of the member of the structure or union to which the first expression points, and is an lvalue.

10.3.4 Post-Increment and Post-Decrement

post-increment-expression:
postfix-expression++

post-decrement-expression:
postfix-expression−−

The operand of post-increment and post-decrement must be a modifiable lvalue, and a scalar (not a structure, union or array).

The effect of the operation is that the operand is incremented or decremented by 1, adjusted for the type of the operand. For example, if the operand is declared to be a "pointer to *type*", then the increment or decrement will be by the value `sizeof( type )`.

The result of both post-increment and post-decrement (if it is just a subexpression of a larger expression) is the original, unmodified value of the operand. In other words, the original value of the operand is used in the expression, and then it is incremented or decremented. Whether the operand is incremented immediately after use or after completion of execution of the expression is undefined. Consider the statements,

```
int i = 2;
int j;

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

Depending on the compiler, `j` may get the value 4 or 5. If the increments are delayed until after the expression is evaluated, `j` gets the value `2 + 2`. If the increment of `i` happens immediately after its value is retrieved, then `j` gets the value `2 + 3`.

To avoid ambiguity, the above expression could be written as:

```
j = i + i;
i += 2;
```

10.4 Unary Operators

unary-expression:
  postfix-expression
or
  pre-increment-expression
or
  pre-decrement-expression
or
  unary-operator cast-expression
or
  sizeof-expression

unary-operator: one of
`& * + - ~ !`

10.4.1 Pre-Increment and Pre-Decrement Operators

pre-increment-expression:
  `++` *unary-expression*

pre-decrement-expression:
  `--` *unary-expression*

The operand of the pre-increment and pre-decrement operators must be a modifiable lvalue, and a scalar (not a structure, union or array).

The operand is incremented or decremented by 1, adjusted for the type of the operand. For example, if the operand is declared to be a "pointer to *type*", then the increment or decrement will be by the value `sizeof(` *type* `).`

The expression `++obj` is equivalent to `(obj += 1)`, while `--obj` is equivalent to `(obj -= 1).`

10.4.2 Address-of and Indirection Operators

unary-expression:
  `&` *cast-expression*
or
  `*` *cast-expression*

The unary `&` symbol denotes the *address-of* operator. Its operand must designate a function or an array, or be an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier. If the type of the operand is "*type*", then the type of the result is "pointer to *type*" and the result is the address of the operand.

If the type of the operand is "array of *type*", then the type of the result is "pointer to *type*" and the result is the address of the first element of the array.

The `*` symbol, in its unary form, denotes the *indirection* or *pointer* operator. Its operand must be a pointer type, except that it may not be a pointer to `void`. If the operand is a "pointer to *type*", then the type of the result is "*type*", and the result is the object to which the operand points.

No checking is performed to ensure that the value of the pointer is valid. If an invalid pointer value is used, the behavior of `*` is undefined.

Examples:

```
int   counter;
int * ctrptr;
void  (*fnptr)( int, int * );

ctrptr  = &counter;
*ctrptr = 3;

fnptr = FnRetVoid;
fnptr( *ctrptr, &counter );
```

10.4.3 Unary Arithmetic Operators

unary-expression:
+ *cast-expression*
or
− *cast-expression*
or
~ *cast-expression*
or
! *cast-expression*

The + symbol, in its unary form, simply returns the value of its operand. The type of its operand must be an arithmetic type (character, integer or floating-point). Integral promotion is performed on the operand, and the result has the promoted type.

The − symbol, in its unary form, is the *negation* or *negative* operator. The type of its operand must be an arithmetic type (character, integer or floating-point). The result is the negative of the operand. Integral promotion is performed on the operand, and the result has the promoted type. The expression `-obj` is equivalent to `(0-obj)`.

The ~ symbol is the *bitwise complement, 1's complement* or *bitwise not* operator. The type of the operand must be an integral type, and integral promotion is performed on the operand. The type of the result is the type of the promoted operand. Each bit of the result is the complement of the corresponding bit in the operand, effectively turning 0 bits to 1, and 1 bits to 0.

The `!` symbol is the *logical not* operator. Its operand must be a scalar type (not a structure, union or array). The result type is `int`. If the operand has the value zero, then the result value is 1. If the operand has some other value, then the result is 0.

10.4.4 The sizeof Operator

sizeof-expression:
  `sizeof` *unary-expression*
or
`sizeof(` *type-name* `)`

The `sizeof` operator gives the size (in bytes) of its operand. The operand may be an expression, or a type in parentheses. In either case, the type must not be a function, bit-field or incomplete type (such as `void`, or an array that has not had its length declared).

Note that an expression operand to `sizeof` is not evaluated. The expression is examined to determine the result type, from which the size is determined.

If the operand has a character type, then the result is 1.

If the type is a structure or union, then the result is the total number of bytes in the structure or union, including any internal or trailing padding included by the compiler for alignment purposes. The size of a structure can be greater than the sum of the sizes of its members.

If the type is an array, then the result is the total number of bytes in the array, unless the operand is a parameter in the function definition enclosing the current block, in which case the result is the size of a pointer.

The type of the result of the `sizeof` operator is implementation-defined, but it is an unsigned integer type, and is represented by `size_t` in the `<stddef.h>` header.

For the Open Watcom C[16] and C[32] compilers, the macro `size_t` is `unsigned int`.

Example:

```
struct s {
    struct s * next;
    int        obj1;
    int        obj2;
};

static struct s * SAllocAndFill( const struct s * def_s )
/*******************************************************/
{
    struct s * sptr;

    sptr = malloc( sizeof( struct s ) );
    if( sptr != NULL ) {
        memcpy( sptr, def_s, sizeof( struct s ) );
    }
    return( sptr );
}
```

The function `SAllocAndFill` receives a pointer to a `struct s.` It allocates such a structure, and copies the contents of the structure pointed to by `def_s` into the allocated memory. A pointer to the allocated structure is returned.

The library function `malloc` takes the number of bytes to allocate as a parameter and `sizeof( struct s )` provides that value. The library function `memcpy` also takes, as the third parameter, the number of bytes to copy and again `sizeof( struct s )` provides that value.

10.5 Cast Operator

cast-expression:
  unary-expression
or
  (*type-name*) *cast-expression*

When an expression is preceded by a type name in parentheses, the value of the expression is converted to the named type. This is called a *cast*. Both the type name and the operand type must be scalar (not a structure, union or array), unless the type name is `void`. If the type name is `void`, the operand type must be a complete type (not an array of unknown size, or a structure or union that has not yet been defined).

A cast does not yield an lvalue.

Pointers may be freely converted from "pointer to `void`" to any other pointer type without using an explicit cast operator. Pointers also may be converted from any pointer type to "pointer to `void`".

A pointer may be converted to a pointer to another type. However, the pointer may be invalid if the resulting pointer is not properly aligned for the type. Converting a pointer to a pointer to a type with less strict alignment, and back again, will yield the same pointer. However, converting it to a pointer to a type with more strict alignment, and back again, may yield a different pointer. On many computers, where alignment is not required (but may improve performance), conversion of pointers may take place freely.

> With Open Watcom C16 and C32, alignment of integers, pointers and floating-point numbers is not required, so the compiler does not do any alignment. However, aligning these types may make a program run slightly faster.
>
> A command line switch may be used to force the compiler to do alignment on all structures.

A pointer to a function may be converted to a pointer to a different type of function, and back again. The resulting pointer will be the same as the original pointer.

If a pointer is converted to a pointer to a different type of function, and a call is made using that pointer, the behavior is undefined.

A pointer may be converted to an integral type. The type of integer required to hold the value of the pointer is implementation-defined. If the integer is not large enough to fully contain the value, then the behavior is undefined.

An integer may be converted to a pointer. The result is implementation-defined.

> With Open Watcom C16, for the purposes of conversion between pointers and integers, `__near` pointers are treated as `unsigned int`. `__far` and `__huge` pointers are treated as `unsigned long int`, with the pointer's segment value in the high-order (most significant) two bytes. All the usual integer conversion rules then apply. Note that huge pointers are not normalized in any way.

With Open Watcom C[32], for the purposes of conversion between pointers and integers, `__near` pointers are treated as `unsigned int`. `__far16` and `_Seg16` pointers are also treated as `unsigned int`, with the pointer's segment value in the high-order (most significant) two bytes. All the usual integer conversion rules then apply. Note that `__far` pointers may not be converted to an integer without losing the segment information.

10.6 Multiplicative Operators

multiplicative-expression:
cast-expression
or
multiplicative-expression `*` *cast-expression*
or
multiplicative-expression `/` *cast-expression*
or
multiplicative-expression `%` *cast-expression*

The `*` symbol, in its binary form, yields the *product* of its operands. The operands must have arithmetic type, and have the usual arithmetic conversions performed on them.

The `/` symbol yields the *quotient* from the division of the first operand by the second operand. The operands must have arithmetic type, and have the usual arithmetic conversions performed on them. Note that when a division by zero occurs, the behavior is undefined.

When both operands of / are of integer type and positive value, and the division is inexact, the result is the largest integer less than the algebraic (exact) quotient. (The result is rounded down.)

When one or both operands of / is negative and the division is inexact, whether the compiler rounds the value up or down is implementation-defined.

The Open Watcom C[16] and C[32] compilers always round the result of integer division toward zero. This action is also called truncation.

The `%` symbol yields the *remainder* from the division of the first operand by the second operand. The operands of `%` must have integral type.

When both operands of `%` are positive, the result is a positive value smaller than the second operand. When one or both operands is negative, whether the result is positive or negative is implementation-defined.

With the Open Watcom C[16] and C[32] compiler, the remainder has the same sign as the first operand.

For integral types `a` and `b`, if `b` is not zero, then `(a/b)*b + a%b` will equal `a`.

10.7 Additive Operators

additive-expression:
multiplicative-expression
or
additive-expression + *multiplicative-expression*
or
additive-expression − *multiplicative-expression*

The + symbol, in its binary form, denotes the *sum* of its operands.

If both operands have arithmetic type, then the usual arithmetic conversions are performed on them.

If one of the operands is a pointer, then the other operand must have an integral type. The pointer operand may not be a pointer to `void`. Before being added to the pointer value, the integral value is multiplied by the size of the object to which the pointer points. The result type is the same as the pointer operand type. If the pointer value is a pointer to a member of an array, then the resulting pointer will point to a member of the same array, provided the array is large enough. If the resulting pointer does not point to a member of the array, then its use with the unary `*` (indirection) or `->` (arrow) operator will yield undefined behavior.

The − symbol, in its binary form, denotes the *difference* resulting from the subtraction of the second operand from the first. If both operands have arithmetic type, then the usual arithmetic conversions are performed on them.

If the first operand is a pointer, then the second operand must either be a pointer to the same type or an integral type.

In the same manner as for adding a pointer and an integral value, the integral value is multiplied by the size of the object to which the pointer points. The pointer operand may not be a pointer to `void`. The result type is the same type as the pointer operand.

If both operands are pointers to the same type, the difference is divided by the size of the type, representing the difference of the subscripts of the two array members (assuming the type is "array of *type*"). The type of the result is implementation-defined, and is represented by `ptrdiff_t` (a signed integral type) defined in the `<stddef.h>` header.

With Open Watcom C16 and C32, `ptrdiff_t` is `int`, unless the huge memory model is being used, in which case `ptrdiff_t` is `long int`.

10.8 Bitwise Shift Operators

shift-expression:
additive-expression
or
shift-expression << *additive-expression*
or
shift-expression >> *additive-expression*

The << symbol denotes the *left-shift* operator. Both operands must have an integral type, and the integral promotions are performed on them. The type of the result is the type of the promoted left operand.

The result of `op << amt` is `op` left-shifted `amt` bit positions. Zero bits are filled on the right. Effectively, the high bits shifted out of `op` are discarded, and the resulting set of bits is re-interpreted as the result. Another interpretation is that `op` is multiplied by 2 raised to the power `amt`.

The >> symbol denotes the *right-shift* operator. Both operands must have an integral type, and the integral promotions are performed on them. The type of the result is the type of the promoted left operand.

The result of `op >> amt` is `op` right-shifted `amt` bit positions. If `op` has an unsigned type, or a signed type and a non-negative value, then `op` is divided by 2 raised to the power `amt`. Effectively, the low bits shifted out of `op` are discarded, zero bits are filled on the left, and the resulting set of bits is re-interpreted as the result.

If `op` has a signed type and negative value, then the behavior of `op >> amt` is implementation-defined. Usually, the high bits vacated by the right shift are filled with the sign bit from before the shift (arithmetic right shift), or with 0 (logical right shift).

> With Open Watcom C^{16} and C^{32}, a right shift of a negative value of a signed type causes the sign bit to be propogated throughout the bits vacated by the shift. Essentially, the vacated bits are filled with 1 bits.

For both bitwise shift operators, if the number of bits to shift exceeds the number of bits in the type, the result is undefined.

10.9 Relational Operators

relational-expression:
shift-expression
or
relational-expression < *shift-expression*
or
relational-expression > *shift-expression*
or
relational-expression <= *shift-expression*
or
relational-expression >= *shift-expression*

Each of the symbols < *(less than),* > *(greater than),* <= *(less than or equal to),* >= *(greater than or equal to),* yields the value 1 if the relation is true, and 0 if the relation is false. The result type is `int`.

If both operands have arithmetic type, then the usual arithmetic conversions are performed on them.

If one of the operands is a pointer, then the other operand must be a pointer to a compatible type. The result depends on where (in the address space of the computer) the pointers actually point.

If both pointers point to members of the same array object, then the pointer that points to the member with a higher subscript will be greater than the other pointer.

If both pointers point to different members within the same structure, then the pointer pointing to the member declared later in the structure will be greater than the other pointer.

If both pointers point to the same union object, then they will be equal.

All other comparisons yield undefined behavior. As discussed above, the relationship between pointers is determined by the locations in the machine storage that the pointers reference. Typically, the numeric values of the pointer operands are compared.

10.10 Equality Operators

equality-expression:
relational-expression
or
equality-expression == *relational-expression*
or
equality-expression `!=` *relational-expression*

The symbols == *(equal to)* and `!=` *(not equal to)* yield the value 1 if the relation is true, and 0 if the relation is false. The result type is `int`.

If both operands have arithmetic type, then the usual arithmetic conversions are performed on them.

If both operands are pointers to the same type and they compare equal, then they are pointers to the same object.

If both operands are pointers and one is a pointer to `void`, then the other is converted to a pointer to `void`.

If one of the operands is a pointer, the other may be a null pointer constant (zero).

No other combinations are valid.

10.11 Bitwise AND Operator

and-expression:
equality-expression
or
and-expression `&` *equality-expression*

The `&` symbol, in its binary form, denotes the *bitwise AND* operator. Each of the operands must have integral type, and the usual arithmetic conversions are performed.

The result is the bitwise AND of the two operands. That is, the bit in the result is set if and only if each of the corresponding bits in the operands are set.

The following table illustrates some bitwise AND operations:

Operation	Result
`0x0000 & 0x7A4C`	`0x0000`
`0xFFFF & 0x7A4C`	`0x7A4C`
`0x1001 & 0x0001`	`0x0001`
`0x29F4 & 0xE372`	`0x2170`

10.12 Bitwise Exclusive OR Operator

exclusive-or-expression:
and-expression
or
exclusive-or-expression ^ *and-expression*

The ^ symbol denotes the *bitwise exclusive OR* operator. Each of the operands must have integral type, and the usual arithmetic conversions are performed.

The result is the bitwise exclusive OR of the two operands. That is, the bit in the result is set if and only if exactly one of the corresponding bits in the operands is set.

Another interpretation is that, if one of the operands is treated as a mask, then every 1 bit in the mask causes the corresponding bit in the other operand to be complemented (0 becomes 1, 1 becomes 0) before being placed in the result, while every 0 bit in the mask causes the corresponding bit in the other operand to be placed unchanged in the result.

The following table illustrates some exclusive OR operations:

Operation	Result
`0x0000 ^ 0x7A4C`	`0x7A4C`
`0xFFFF ^ 0x7A4C`	`0x85B3`
`0xFFFF ^ 0x85B3`	`0x7A4C`
`0x1001 ^ 0x0001`	`0x1000`
`0x29F4 ^ 0xE372`	`0xCA86`

10.13 Bitwise Inclusive OR Operator

inclusive-or-expression:
exclusive-or-expression
or
inclusive-or-expression | *exclusive-or-expression*

The | symbol denotes the *bitwise inclusive OR* operator. Each of the operands must have integral type, and the usual arithmetic conversions are performed.

The result is the bitwise inclusive OR of the two operands. That is, the bit in the result is set if at least one of the corresponding bits in the operands is set.

The following table illustrates some inclusive OR operations:

Operation		Result
0x0000	0x7A4C	0x7A4C
0xFFFF	0x7A4C	0xFFFF
0x1100	0x0022	0x1122
0x29F4	0xE372	0xEBF6

10.14 Logical AND Operator

logical-and-expression:
inclusive-or-expression
or
logical-and-expression `&&` *inclusive-or-expression*

The `&&` symbol denotes the *logical AND* operator. Each of the operands must have scalar type.

If both of the operands are not equal to zero, then the result is 1. Otherwise, the result is zero. The result type is `int`.

If the first operand is zero, then the second operand is not evaluated. Any side effects that would have happened if the second operand had been executed do not happen. Any function calls encountered in the second operand do not take place.

10.15 Logical OR Operator

logical-or-expression:
logical-and-expression
or
logical-or-expression `||` *logical-and-expression*

The `||` symbol denotes the *logical OR* operator. Each of the operands must have scalar type.

If one or both of the operands is not equal to zero, then the result is 1. Otherwise, the result is zero (both operands are zero). The result type is `int`.

If the first operand is not zero, then the second operand is not evaluated. Any side effects that would have happened if the second operand had been executed do not happen. Any function calls encountered in the second operand do not take place.

10.16 Conditional Operator

conditional-expression:
logical-or-expression
or
logical-or-expression `?` *expression* `:` *conditional-expression*

The `?` symbol separates the first two parts of a *conditional* operator, and the `:` symbol separates the second and third parts. The first operand must have a scalar type (not a structure, union or array).

The first operand is evaluated. If its value is not equal to zero, then the second operand is evaluated and its value is the result. Otherwise, the third operand is evaluated and its value is the result.

Whichever operand is evaluated, the other is not evaluated. Any side effects that might have happened during the evaluation of the other operand do not happen.

If both the second and third operands have arithmetic type, then the usual arithmetic conversions are performed on them, and the type of the result is the same type as the converted operands.

If both operands have the same structure, union or pointer type, then the result has that type.

If both operands are pointers, and one is "pointer to `void`", then the result type is "pointer to `void`".

If one operand is a pointer, and the other is a null pointer constant (0), the result type is that of the pointer.

If both operands are void expressions, then the result is a void expression.

No other combinations of result types are permitted.

Note that, unlike most other operators, the conditional operator associates from right to left. For example, the expression,

```
a = b ? c : d ? e : f;
```

is translated as if it had been parenthesized as follows:

```
a = b ? c : (d ? e : f);
```

This construct is confusing, and so should probably be avoided.

10.17 Assignment Operators

assignment-expression:
  conditional-expression
or
  simple-assignment-expression
or
  compound-assignment-expression

An *assignment operator* stores a value in the object designated by the left operand. The left operand must be a modifiable lvalue.

The result type and value are those of the left operand after the assignment.

Whether the left or right operand is evaluated first is undefined.

Note that, unlike most other operators, the assignment operators associate from right to left. For example, the expression,

```
a += b = c;
```

is translated as if it had been bracketed as follows:

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

10.17.1 Simple Assignment

simple-assignment-operator:
unary-expression = *assignment-expression*

The = symbol denotes *simple assignment.* The value of the right operand is converted to the type of the left operand and replaces the value designated by the left operand.

The two operands must obey one of the following rules,

- both have arithmetic types,
- both have the same structure or union type, or the right operand differs only in the presence of the `const` or `volatile` keywords,
- both are pointers to the same type,
- both are pointers and one is a pointer to `void`,
- the left operand is a pointer, and the right is a null pointer constant (0).

10.17.2 Compound Assignment

compound-assignment-expression:
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`+= -=`
`*= /= %=`
`&= ^= |=`
`<<= >>=`

A *compound assignment* operator of the form `a` *op*`= b` is equivalent to the simple assignment expression `a = a` *op* `(b)`, except that the left operand `a` is evaluated only once.

The compound assignment operator must have operands consistent with those allowed by the corresponding binary operator.

10.18 Comma Operator

expression:
assignment-expression
or
expression`,` *assignment-expression*

At the lowest precedence, the *comma operator* evaluates the left operand as a void expression (it is evaluated and its result, if any, is discarded), and then evaluates the right operand. The result has the type and value of the second operand.

In contexts where the comma is also used as a separator (function argument lists and initializer lists), a comma expression must be placed in parentheses.

For example,

```
Fn( (pi=3.14159,two_pi=2*pi) );
```

the function `Fn` has one parameter, which has the value 2 times `pi`.

```
for( i = 0, j = 0, k = 0;; i++, j++, k++ )
  statement;
```

The `for` statement allows three expressions. In this example, the first expression initializes three objects and the third expression increments the three objects.

10.19 Constant Expressions

A constant expression may be specified in several places:

- the size of a bit-field member of a structure,
- the value of an enumeration constant,
- an initializer list,
- the number of elements in an array,
- the value of a `case` label constant,
- with the `#if` and `#elif` preprocessor directives.

In most cases, a constant expression consists of a series of constant values and operations that evaluate to a constant value. Certain operations may only appear within the operand of the `sizeof` operator. These include:

- a function call,
- pre- or post-increment or decrement,
- assignment,
- comma operator,
- array subscripting,
- the . and, −> operators (structure member access),
- the unary `&` (address-of) operator (see exception below),
- the unary `*` (indirection) operator,
- casts to a type other than an integer type.

In a constant expression that is an initializer, floating-point constants and casts may be specified. Objects that have static storage duration, and function designators (names), may be used to provide addresses, either explicitly using the unary `&` (address-of) operator, or implicitly by specifying the identifier only.

The following examples illustrate constant expressions that may be used anywhere:

```
3
256*3 + 27
OPSYS == OS_DOS /* These are macro names */
```

The next set of examples are constant expressions that are only valid in an initializer:

```
&SomeObject
SomeFunction
3.5 * 7.2 / 6.5
```

In a constant expression that is part of a `#if` or `#elif` preprocessor directive, only integral constants and operators are permitted (and macros that, when replaced, follow these same rules).

11 Statements

A *statement* describes what actions are to be performed. Statements may only be placed inside functions. Statements are executed in sequence, except where described below.

11.1 Labelled Statements

Any statement may be preceded by a *label.* Labelled statements are usually the target of a `goto` statement, and hence occur infrequently.

A label is an identifier followed by a colon. Labels do not affect the flow of execution of a program. A label that is encountered during execution is ignored.

The following example illustrates a statement with a label:

```
xyz: i = 0;
```

Labels can only precede statements. It follows that labels may only appear inside functions.

A label may be defined only once within any particular function.

The identifier used for a label may be the same as another identifier for an object, function or tag, or a label in another function. The *name space* for labels is separate from non-label identifiers, and each function has its own label name space.

11.2 Compound Statements

A *compound statement* is a set of statements grouped together inside braces. It may have its own declarations of objects, with or without initializations, and may or may not have any executable statements. A compound statement is also called a *block.*

The general form of a compound statement is:

{ *declaration-list statement-list* }

where *declaration-list* is a list of zero or more declarations of objects to be used in the block. *statement-list* is a list of zero or more statements to be executed when the block is entered.

Any declarations for objects that have automatic storage duration and initializers for them are evaluated in the order in which they occur.

An object declared with the keyword `extern` inside a block may not be initialized in the declaration, since the storage for that object is defined elsewhere.

An object declared in a block, without the keyword `extern`, may not be redeclared within the same block, except in a block contained within the current block.

11.3 Expression Statements

A statement that is an expression is evaluated as a void expression for its side effects, such as the assigning of a value with the assignment operator. The result of the expression is discarded. This discarding may be made explicit by casting the expression as a `void`.

For example, the statement,

```
count = 3;
```

consists of the expression `count = 3`, which has the side effect of assigning the value 3 to the object `count`. The result of the expression is 3, with the type the same as the type of `count`. The result is not used any further. As another example, the statement,

```
(void) memcpy( dest, src, len );
```

indicates that, regardless of the fact that `memcpy` returns a result, the result should be ignored. However, it is equally valid, and quite common, to write,

```
memcpy( dest, src, len );
```

As a matter of programming style, casting an expression as `void` should only be done when the result of the expression might normally be expected to be used further. In this case, casting to `void` indicates that the result was intentionally discarded and is not an error of omission.

11.4 Null Statements

A null statement, which is just a semi-colon, takes no action. It is useful for placing a label just before a block-closing brace, or for indicating an empty block, such as in an iteration statement. Consider the following examples of null statements:

```
{
    gets( buffer );
    while( *buffer++ != '\0' )
        ;
    /* ... */
    endblk: ;
}
```

The `while` iteration statement skips over characters in `buffer` until the null character is found. The body of the iteration is empty, since the controlling expression does all of the work. The `endblk:` declares a label just before the final }, which might be used by a `goto` to exit the block.

11.5 Selection Statements

A *selection statement* evaluates an expression, called the *controlling expression,* then based on the result selects from a set of statements. These statements are then executed.

11.5.1 The if Statement

`if(` *expression* `)` *statement*
or
`if(` *expression* `)` *statement* `else` *statement*

In both cases, the type of the controlling expression (inside the parentheses) is a scalar type (not a structure, union or array). If the controlling expression evaluates to a non-zero value, then the first statement is executed.

In the second form, the `else` is executed if the controlling expression evaluates to zero.

Each statement may be a compound statement. For example,

```
if( delay > 5 ) {
    printf( "Waited too long\n" );
    ok = FALSE;
} else {
    ok = TRUE;
}
```

In the classic case of the dangling `else`, the `else` is bound to the nearest `if` that does not yet have an `else`. For example,

```
if( x > 0 )
    if( y > 0 )
        printf( "x > 0 && y > 0\n" );
else
    printf( "x <= 0\n" );
```

will print `x <= 0` when `x > 0` is true and `y > 0` is false, because the `else` is bound to the second `if`, not the first. To correct this example, it would have to be changed to,

```
if( x > 0 ) {
    if( y > 0 )
        printf( "x > 0 && y > 0\n" );
} else
    printf( "x <= 0\n" );
```

This example illustrates why it is a good idea to always use braces to explicitly state the subject of the control structures, rather than relying on the fact that a single statement is also a compound statement. A better way of writing the above example is,

```
if( x > 0 ) {
    if( y > 0 ) {
        printf( "x > 0 && y > 0\n" );
    }
} else {
    printf( "x <= 0\n" );
}
```

where all subjects of the control structures are contained within braces, leaving no doubt about the meaning. A dangling `else` cannot occur if braces are always used.

If the statements between the `if` and the `else` are reached via a label, the statements following the `else` will not be executed. However, jumping into a block is poor programming practice, since it makes the program difficult to follow.

11.5.2 The switch Statement

`switch(` *expression* `)` *statement*

Usually, *statement* is a compound statement or block. Embedded within the statement are `case` labels and possibly a `default` label, of the following form:

`case` *constant-expression : statement*
`default :` *statement*

The controlling expression and the constant-expressions on each `case` label all must have integral type. No two of the `case` constant-expressions may be the same value. The `default` label may appear at most once in any `switch` block.

The controlling statement is evaluated, and the integral promotion is performed on the result. If the promoted value of the expression matches any of the case labels promoted to the same type, control is given to the statement following that case label. Otherwise, control is given to the statement following the `default` label (if present). If no default label is present, then no statements in the `switch` block are executed.

When statements within a `switch` block are being executed and another `case` or `default` is encountered, it is ignored and execution continues with the statement following the label. The `break` statement may be used to terminate execution of the switch block.

In the following example,

```
int i;

for( i = 1; i <= 8; i++ ) {
    printf( "%d ", i );
    switch( i ) {
      case 2:
      case 4:
        printf( "less than 5 " );
      case 6:
      case 8:
        printf( "even\n" );
        break;
      default:
        printf( "odd\n" );
    }
}
```

the following output is produced:

```
1 odd
2 less than 5 even
3 odd
4 less than 5 even
5 odd
6 even
7 odd
8 even
```

11.6 Iteration Statements

Iteration statements control looping. There are three forms of iteration statements: `while`, `do`/`while` and `for`.

The controlling expression must have a scalar type. The *loop body* (often a compound statement or block) is executed repeatedly until the controlling expression is equal to zero.

11.6.1 The while Statement

`while (` *expression* `)` *statement*

The evaluation of the controlling expression takes place before each execution of the loop body (*statement*). If the expression evaluates to zero the first time, the loop body is not executed at all.

The *statement* may be a compound statement.

For example,

```
char * ptr;
/* ... */
while( *ptr != '\0' ) {
    if( *ptr == '.' )break;
    ++ptr;
}
```

The loop will scan characters pointed at by `ptr` until either a null character or a dot is found. If the initial value of `ptr` points at a null character, then no part of the loop body will be executed, leaving `ptr` pointing at the null character.

11.6.2 The do Statement

`do` *statement* `while (` *expression* `);`

The evaluation of the controlling expression takes place after each execution of the loop body (*statement*). If the expression evaluates to zero the first time, the loop body is executed exactly once.

The *statement* may be a compound statement.

For example,

```
char * ptr;
char * endptr;
/* ... */
endptr = ptr + strlen( ptr );
do {
    --endptr;
} while( endptr >= ptr  &&  *endptr == ' ' );
```

In this example, the loop will terminate when `endptr` finds a non-blank character starting from the right, or when `endptr` goes past the beginning of the string. If a non-blank character is found, `endptr` will be left pointing at that character.

11.6.3 The for Statement

The statement,

`for (` *expr1*`;` *expr2*`;` *expr3* `)` *statement*

is almost equivalent to,

expr1`;`
`while (` *expr2* `) {`
statement
expr3`;`
`}`

The difference is that the `continue` statement will pass control to the statement *expr3* rather than to the end of the loop body.

expr1 is an initialization expression and may be omitted.

expr2 is the controlling expression, and specifies an evaluation to be made before each iteration of the loop body. If the expression evaluates to zero, the loop body is not executed, and control is passed to the statement following the loop body. If *expr2* is omitted, then a non-zero (true) value is substituted in its place. In this case, the statements in the loop must cause an explicit break from the loop.

expr3 specifies an operation to be performed after each iteration. A common operation would be the incrementing of a counter. *expr3* may be omitted.

The *statement* may be a compound statement.

For example,

```
char charvec[256];
int  count;

for( count = 0; count <= 255; count++ ) {
    charvec[count] = count;
}
```

This example will initialize the character array `charvec` to the values from 0 to 255.

The following are examples of `for` statements:

```
for( ;; )
statement;
```

All statements in the body of the loop will be executed until a `break` or `goto` statement is executed which passes control outside of the loop, or a `return` statement is executed which exits the function. This is sometimes called *loop forever*.

```
for( i = 0; i <= 100; ++i )
statement;
```

The object `i` is given the initial value zero, and after each iteration of the loop is incremented by one. The loop is executed 101 times, with `i` having the successive values `0`, `1`, `2 ... 99`, `100`, and having the value `101` after termination of the loop.

```
for( ; *bufptr != '\0'; ++bufptr )
statement;
```

The object `bufptr` is already initialized, and the loop will continue until `bufptr` points at a null character. After each iteration of the loop, `bufptr` will be incremented to point at the next character.

11.7 Jump Statements

A jump statement causes execution to continue at a specific place in a program, without executing any other intervening statements. There are four jump statements: `goto`, `continue`, `break` and `return`.

11.7.1 The goto Statement

`goto` *identifier*`;`

identifier is a label somewhere in the current function (including any block within the function). The next statement executed will be the one following that label.

Note: it can be confusing to use the `goto` statement excessively. It is easy to create *spaghetti code,* which is very difficult to understand, even by the person who wrote it. It is recommended that the `goto` statement be used, at most, to jump *out of* blocks, never into them.

11.7.2 The continue Statement

```
continue;
```

A `continue` statement may only appear within a loop body, and causes a jump to the inner-most loop's loop-continuation statement (the end of the loop body).

In a `while` statement, the jump is effectively back to the `while`.

In a `do` statement, the jump is effectively down to the `while`.

In a `for` statement, the jump is effectively to the closing brace of the compound-statement that is the subject of the `for` loop. The third expression in the `for` statement, which is often an increment or decrement, is then executed before control is returned to the top of the loop.

11.7.3 The break Statement

```
break;
```

A `break` statement may only appear in an iteration (loop) body or a `switch` statement.

In a loop, a `break` will cause execution to continue at the statement following the loop body.

In a `switch` statement, a `break` will cause execution to continue at the statement following the switch. If the loop or `switch` that contains the `break` is enclosed inside another loop or `switch`, only the inner-most loop or `switch` is terminated. The `goto` statement may be used to terminate more than one loop or `switch`.

11.7.4 The return Statement

`return;`
or
`return` *expression*`;`

A popular variation of the second form is,

```
return( expression );
```

The `return` statement causes execution of the current function to be terminated, and control is passed to the caller. A function may contain any number of `return` statements.

If the function is declared with a return type of `void` (no value is returned), then no `return` statement within that function may return a value.

If the function is declared as having a return type of other than `void`, then any `return` statement with an expression will evaluate the expression and convert it to the return type. That value will be the value returned by the function. If a `return` is executed without an expression, and the caller uses the value returned by the function, the behavior is undefined since no value was returned. An arbitrary value will probably be used.

Reaching the closing brace } that terminates the function is equivalent to executing a `return` statement without an expression.

12 Functions

There are two forms for defining a function. The first form is,

storage-class return-type identifier (*parameter-type-list*)
{
declaration-list

statement-list
}

The *storage-class* may be one of `extern` or `static`. If *storage-class* is omitted, `extern` is assumed.

The *return-type* may be any valid type except an *array*. If *return-type* is omitted, `int` is assumed.

The *identifier* is the name of the function.

The *parameter-type-list* is either `void` or empty, meaning the function takes no parameters, or a comma-separated list of declarations of the objects, including both type and parameter name (identifier). If multiple arguments of the same type are specified, the type of each argument must be given individually. The form,

type id1, id2

is not permitted within the parameter list.

If the *parameter-type-list* ends with `,...` then the function will accept a variable number of arguments.

Any parameter declared as "array of *type*" is changed to "pointer to *type*". Any parameter declared as "*function*" is changed to "pointer to *function*".

The following examples illustrate several function definitions:

```
int F( void )
```

The function `F` has no parameters, and returns an integer.

```
void G( int x )
```

The function `G` has one parameter, an integer object named `x`, and does not return a value.

```
void * H( long int len, long int wid )
```

The function `H` has two parameters, long integer objects named `len` and `wid`, and returns a pointer which does not point to any particular type of object.

```
void I( char * format, ... )
```

The function `I` has one known parameter, an object named `format` that is a pointer to a character (string). The function also accepts a variable number of parameters following `format`. The function does not return a result.

This form of function definition also serves as a prototype declaration for any calls to the function that occur later in the same module. With the function prototype in scope at the time of a call to the function, the arguments are converted to the type of the corresponding parameter prior to the value being assigned. If a call to the function is to be made prior to its definition, or from another module, a function prototype should be specified for it in order to ensure proper conversion of argument types. Failure to do this will result in the default argument promotions being performed, with undefined behavior if the function parameter types do not match the promoted argument types.

The second form of function definition is,

storage-class return-type identifier `(` *identifier-list* `)`
declaration-list
`{`
declaration-list

statement-list
`}`

The *storage-class*, *return-type* and *identifier* parts are all the same as for the first form of definition. In this form, the *identifier-list* is a (possibly empty) comma-separated list of identifiers (object names) without any type information. Following the closing parenthesis, and before the opening brace of the body of the function, the declarations for the objects are given, using the normal rules. Any object of type `int` need not be explicitly declared.

In the declarations of the parameter identifiers, `register` is the only storage-class specifier that may be used.

A function prototype is created from the definition after the default argument promotions have been performed on each parameter. All arguments to a function declared in this manner will have the default argument promotions performed on them. The resulting types must match the types of the declared parameters, after promotion. Otherwise, the behavior is undefined.

Note that it is impossible to pass an object of type `float` to a function declared in this manner. The argument of type `float` will automatically be promoted to `double`, and the parameter will also be promoted to `double` (assuming that it was declared as `float`). For similar reasons, it is not possible to pass an object of type `char` or `short int` without promotion taking place.

According to the ISO standard for the C language, **this form of function definition is obsolete** and should not be used. It is provided for historical reasons, in particular, for compatibility with older C compilers. Using the first form of function definition often allows the compiler to generate better code.

The following examples are the same as those given with the first form above, with the appropriate modifications:

```
int F()
```

The function `F` has no parameters, and returns an integer.

```
void G( x )
```

The function `G` has one parameter, an integer object named `x`, and does not return a value. This example could have also been written as,

```
void G( x )
    int x;
```

which explicitly declares `x` to be an integer.

```
void * H( len, wid )
    long int len;
    long int wid;
```

The function `H` has two parameters, both integer objects named `len` and `wid`, and returns a pointer which does not point to any particular type of object. Any call to this function must ensure that the arguments are long integers, either by using an object so declared, or by explicitly casting the object to the type.

The last example using the ellipsis (`,...`) notation is not directly representable using the second form of function definition. With most compilers it is possible to handle variable argument lists in this form, but knowledge of the mechanism used to pass arguments to functions is required, and this mechanism may vary between different compilers.

12.1 The Body of the Function

Following the declaration of the function and the opening brace is the *body* of the function. It consists of two portions, both of which are optional.

The first portion is the declaration list for any objects needed within the function. These objects may have any type and any storage class. Objects with storage class `register` or `auto` have *automatic storage duration*, meaning they are created when the function is called, and destroyed when the function returns to the caller. (The value of the object is not preserved between calls to the function.) Objects with storage class `extern` or `static` have *static storage duration*, meaning they are created once, before the function is ever called, and destroyed only when the program terminates. Any value placed in such an object will remain even after the function has returned, so that the next time the function is called the value will still be present (unless some other action is taken to change it, such as using another object containing a pointer to the static object to modify the value).

Unless an explicit `return` statement is executed, the function will not return to the caller until the brace at the end of the function definition is encountered. The return will be as if a `return` statement with no expression was executed. If the function is declared as returning a value, and the caller attempts to use the value returned in this manner, the behavior is undefined. The value used will be arbitrary.

A function may call itself (recursion) directly, or it may call another function or functions which in turn call it. Any objects declared with *automatic storage duration* are created as a new instance of the object upon

each recursion, while objects declared with *static storage duration* only have one instance shared between the recursive instances of the function.

12.2 Function Prototypes

A function prototype is like a definition of a function, but without the body. A semi-colon is specified immediately following the closing right parenthesis of the function's declaration. The prototype describes the name of the function, the types of parameters it expects (names are optional) and the type of the return value. This information can be used by the C compiler to do proper argument type checking and conversion for calls to the function, and to properly handle the return value.

If no function prototype has been found by the time a call to a function is made, all arguments have the default argument promotions performed on them, and the return type is assumed to be `int`. If the actual definition of the function does not have parameters that match the promoted types, the behavior is undefined. If the return type is not `int` and a return value is required, the behavior is undefined.

The prototype for a function must match the function definition. Each parameter type and the type of the return value must be the same, otherwise the behavior is undefined.

All library functions have prototypes in one of several header files. That header file should be included whenever a function described therein is used. Refer to the Open Watcom C Library Reference manual for details.

12.2.1 Variable Argument Lists

If the prototype (and definition) for a function has a parameter list that ends with `,...` then the function has a *variable argument list* or *variable parameter list* meaning that the number of arguments to the function can vary. (The library function `printf` is an example.) At least one argument must be provided before the variable portion. This argument usually describes, in some fashion, how many other arguments to expect. It may be a simple count, or may involve (as with `printf`) an encoding of the number and types of arguments.

All arguments that correspond to a variable argument list have the default argument promotions performed on them, since it is not possible to determine, at compilation time, what types will be required by the function.

Since the parameters represented by the `,...` don't have names, special handling is required. The C language provides a special type and three macros for handling variable argument lists. To be able to use these, the header `<stdarg.h>` must be included.

The type `va_list` is an implementation-specific type used to store information about the variable list. Within the function, an object must be declared with type `va_list`. This object is used by the macros and functions for processing the list.

The macro `va_start` has the form,

```
void va_start( va_list parminfo
,
lastparm
 );
```

The object *parminfo* is set up by the macro with information describing the variable list. The argument *lastparm* is the name (identifier) of the last parameter before the `,...` and must not have been declared with the storage class `register`.

The macro `va_start` must be executed before any processing of the variable portion of the parameter list is performed.

`va_start` may be executed more than once, but only if an intervening `va_end` is executed.

The macro `va_arg` has the form,

```
type
 va_arg( va_list
parminfo
,
type
 );
```

parminfo is the same object named in the call to `va_start`. *type* is the type of argument expected. The types expected should only be those that result from the default argument promotions (`int`, `long int` and `long long int` and their unsigned varieties, `double` and `long double`), and those that are not subject to promotion (pointers, structures and unions). The type must be determined by the program. The `va_arg` macro expands to an expression that has the type and value of the next parameter in the variable list.

In the case of `printf`, the parameter type expected is determined by the "conversion specifications" such as `%s`, `%d` and so on.

The first invocation of the `va_arg` macro (after executing a `va_start`) returns the value of the parameter following *lastparm* (as specified in `va_start`). Each subsequent invocation of `va_arg` returns the next parameter in the list. At each invocation, the value of *parminfo* is modified (in some implementation-specific manner) to reflect the processing of the parameter list.

If the type of the next parameter does not match *type*, or if no parameter was specified, the behavior is undefined.

The macro `va_end` has the form,

```
void va_end( va_list parminfo
 );
```

parminfo is the same object named in the corresponding call to `va_start`. The function `va_end` closes off processing of the variable argument list, which must be done prior to returning from the function. If `va_end` is not called before returning, the behavior is undefined.

If `va_end` is called without a corresponding call to `va_start` having been done, the behavior is undefined.

After calling `va_end` and prior to returning, it is possible to call `va_start` again and reprocess the variable list. It will be necessary to call `va_end` again before returning.

The following function takes an arbitrary number of floating-point numbers as parameters along with a count, and returns the average of the numbers:

```
#include <stdarg.h>

extern double Average( int count, ... )
/*************************************/
{
    double  sum = 0;
    int     i;
    va_list parminfo;

    if( count == 0 ) {
        return( 0.0 );
    }
    va_start( parminfo, count );
    for( i = 0; i < count; i++ ) {
        sum += va_arg( parminfo, double );
    }
    va_end( parminfo );
    return( sum / count );
}
```

12.3 The Parameters to the Function main

The function `main` has a special meaning in C. It is the function that receives control when a program is started. The function `main` has the following definition:

```
extern int main( int argc, char * argv[] )
{
    statements
}
```

The objects `argc` and `argv` have the following properties:

- `argc` is the "argument count", or the number of parameters (including program name) supplied to the program, and its value is greater than zero,
- `argv` is an array of pointers to strings containing the parameters,
- `argv[0]` is the program name, if available, otherwise it is a pointer to a string containing only the null character,
- `argv[argc]` is a null pointer, representing the end of the argument list,
- `argv[1]` through `argv[argc-1]` are pointers to strings representing the arguments to the program. These strings are modifiable by the program, and exist throughout the execution of the program. The strings will generally be in mixed (upper and lower) case, although a system that cannot provide mixed case argument strings will provide them in lower case.

The translation of the arguments to the program, as provided by the operating system (often from the command-line used to invoke the program), into the strings contained in `argv`, is implementation-defined.

With Open Watcom C16 and C32, each unquoted, blank-separated token on the command line is made into a string that is an element of `argv`. Quoted strings are maintained as one element without the quotes.

For example, the command line,

```
pgm 2+ 1 tokens "one token"
```

will result in `argc` having the value 5, and the elements of `argv` being the strings `"pgm"`, `"2+"`, `"1"`, `"tokens"` and `"one token"`.

The function `main` may also be declared without any parameters, as,

```
extern int main( void )
{
    statements
}
```

The return value of `main` is an integer, usually representing a termination status. If no return value is specified (by using a `return` statement with no expression or encountering the closing brace in the function), then the value returned is undefined.

The `exit` library function may be used to terminate the program at any point. The value of the argument to `exit` is returned as if `main` had returned the value.

13 The Preprocessor

The *preprocessor*, as its name suggests, is that part of the C compiler which processes certain directives embedded in the source file(s) in advance of the actual compilation of the program. Specifically, the preprocessor allows a source file to,

- include other files (perhaps referencing externally-defined objects, or containing the definitions of structures or other types which are needed by more than one source file),
- compile certain portions of the code depending on some condition (such as the kind of computer for which the code is being generated), and,
- replace *macros* with other text which is then compiled.

The preprocessing phase occurs after trigraphs have been converted and physical lines ending with `\` have been concatenated to create longer logical lines, but before escape sequences in character constants have been converted, or adjacent string literals are concatenated.

Any line whose first non-blank character is a `#` marks the beginning of a *preprocessing directive.* Spaces may appear between the `#` and the identifier for the directive. The `#include` and `#define` directives are each contained on one line (after concatenation of lines ending with `\`), while the conditional compilation directives span multiple lines.

A preprocessor directive is not terminated by a semi-colon.

13.1 The Null Directive

A preprocessing directive of the form,

ix #

ix null

```
#
```

(with no other tokens on the same line) has no effect and is discarded.

13.2 Including Headers and Source Files

A directive of the form,

ix #include

```
#include <name>
```

will search a sequence of places defined by the implementation for the *header* identified by `name.` A header declares a set of library functions and any necessary types or macros needed for their use. Headers are usually provided by the compiler, or by a library provided for use with the compiler.

`name` may not contain a > character. If the header is found, the entire directive is replaced by the contents of the header. If the header is not found, an error will occur.

A directive of the form,

```
#include "name"
```

will search for the source file identified by `name. name` may not contain a `"` (double-quote) character. If the source file identified by `name` is found, then the entire directive is replaced by the contents of the file. Otherwise, the directive is processed as if the,

```
#include <name>
```

form had been used.

A third form of `#include` directive is also supported. A directive of the form,

```
#include tokens
```

causes all macro substitutions (described below) to take place on `tokens.` After substitution, the directive must match either the `<name>` or `"name"` forms described above (including < and >, or quotes), in which case the `#include` is processed in the corresponding manner.

See the User's Guide for details about how the compiler searches for included files.

`#include` directives may be nested. Each implementation may allow different depths of nesting, but all must allow at least 8 levels. (In other words, a source file may include another file, which includes another file, and so on, up to a depth of eight files.)

> The operating system may further limit the number of files that may be open at one time. See the appropriate operating system manual for details.

13.3 Conditionally Including Source Lines

A directive of the form,

ix #if

`#if` *constant-expression*
body of `#if`
`#endif`

evaluates the *constant-expression,* and if it evaluates to a non-zero value, then the body of the `#if` is processed by the preprocessor. Processing of the body ends when a corresponding `#elif`, `#else`, or the terminating `#endif` is encountered.

The `#if` directive allows source and preprocessor lines to be conditionally processed by the compiler.

If the *constant-expression* evaluates to zero, then the body of the `#if` is not processed, and the corresponding `#elif` or `#else` (if present) is processed. If neither of these directives are present, then the preprocessor skips to the `#endif`. Any preprocessing directives within the body of the `#if` are not processed, but they are examined in order to determine any nested directives, in order to find the matching `#elif`, `#else` or `#endif`.

The *constant-expression* is of the same form as used in the `if` statement, except that the values used must be integer values (including character constants). No cast or `sizeof` operators or enumeration constants may be used. Each identifier that is a macro name is replaced (as described below), and remaining identifiers are replaced with `0L`. All values are converted to long integers using the usual arithmetic conversions. After each item has been converted, the evaluation of the expression takes place using the arithmetic of the translation environment. Any character constants are evaluated as members of the *source* character set.

> With Open Watcom C[16] and C[32], character constants have the same value in both the source and execution character sets.

The unary expression,

`defined` *identifier*
or
`defined(` *identifier* `)`

may be used to determine if an identifier is currently defined as a macro. Any macro name that is part of this unary expression is not expanded. The above expressions evaluate to 1 if the named identifier is currently a macro, otherwise they evaluate to 0.

As discussed above, if the *constant-expression* of the `#if` evaluates to zero, the preprocessor looks for a corresponding `#elif`. This directive means "else if", and has a similar form as `#if`:

ix #elif

`#elif` *constant-expression*
body of `#elif`

An `#elif` may only be placed inside the body of an `#if`. The body of the `#elif` is processed only if the *constant-expression* evaluates to a non-zero value and the constant-expressions of the corresponding `#if` and (preceding) `#elif` statements evaluated to zero. Otherwise the body is not processed, and the preprocessor skips to the next corresponding `#elif` or `#else`, or to the `#endif` if neither of these directives is present.

The `#else` directive has the form,

ix #else

`#else`
body of `#else`

The body of the `#else` is processed only if the constant expressions of the corresponding `#if` and `#elif` statements evaluated to zero. The body of the `#else` is processed until the corresponding `#endif` is encountered.

The form of the `#endif` directive is,

```
#endif
```

and marks the end of the `#if`.

The following are examples of conditional inclusion of source lines:

```
#if OPSYS == OS_CMS
    fn_syntax = "filename filetype fm";
#elif OPSYS == OS_MVS
    fn_syntax = "'userid.library.type(membername)'";
#elif OPSYS == OS_DOS  ||  OPSYS == OS_OS2
    fn_syntax = "filename.ext";
#else
    fn_syntax = "filename";
#endif
```

The object `fn_syntax` is set to the appropriate filename syntax string depending on the value of the macro `OPSYS`. If `OPSYS` does not match any of the stated values, then `fn_syntax` is set to the default string `"filename"`.

```
#if HARDWARE == HW_IBM370
    #if OPSYS == OS_CMS
        escape_cmd = "CMS";
    #elif OPSYS == OS_MVS
        escape_cmd = "TSO";
    #else
        escape_cmd = "SYSTEM";
    #endif
#else
    escape_cmd = "SYSTEM";
#endif
```

The object `escape_cmd` is set to an appropriate string depending on the values of the macros `HARDWARE` and `OPSYS`. The indentation of the directives clearly illustrates the flow between various conditions and levels of directives.

13.3.1 The #ifdef and #ifndef Directives

The

ix #ifdef `#ifdef` directive is used to check if an identifier is currently defined as a macro. For example, the directive,

```
#ifdef xyz
```

processes the body of the `#ifdef` only if the identifier `xyz` is currently a macro. This example is equivalent to,

```
#if defined xyz
```

or

```
#if defined( xyz )
```

In a similar manner, the directive,

ix #ifndef

```
#ifndef xyz
```

is equivalent to,

```
    #if !defined xyz
```

or

```
    #if !defined( xyz )
```

13.4 Macro Replacement

A directive of the form,

ix #define

`#define` *identifier replacement-list*

defines a *macro* with the name *identifier.* This particular form of macro is called an *object-like* macro, because it is used like an object (as opposed to a function). Any source line that contains a token matching the macro name has that token replaced by the *replacement-list.* The tokens of the replacement-list are then rescanned for more macro replacements.

For example, the macro,

```
#define TABLE_LIMIT 256
```

defines the macro `TABLE_LIMIT` to be equivalent to the token `256.` This is sometimes called a *manifest constant,* because it provides a descriptive term for a value that makes programs easier to read. It is a very good idea to use descriptive names wherever appropriate to improve the readability of a program. It may also save time if the same value is used many different places, and the value must be changed at some point.

Care must be exercised when using more complicated object-like macros. Consider the following example:

```
#define COUNT1 10
#define COUNT2 20
#define TOTAL_COUNT COUNT1+COUNT2
/* ... */
memptr = malloc( TOTAL_COUNT * sizeof( int ) );
```

If `int` is 2 bytes in size, this call to `malloc` will allocate 50 bytes of memory, instead of the expected 60. This occurs because `TOTAL_COUNT * sizeof( int )` becomes `10+20 * 2` after macro replacement, and the precedence rules for expression evaluation cause the multiply to be done first. To solve this problem, the macro for `TOTAL_COUNT` should be defined as:

```
#define TOTAL_COUNT (COUNT1+COUNT2)
```

A directive of the form,

ix #define

`#define` *identifier* `(` *identifier-list* `)` *replacement-list*

is called a *function-like* macro, because it is used like a function call. No space may appear between *identifier* and the left parenthesis in the macro definition. Any source line(s) that contains what looks like a function call, where the name of the function matches a function-like macro name, and the number of parameters matches the number of identifiers in the *identifier-list,* has the entire function call replaced by the *replacement-list,* substituting the actual arguments of the function call for the occurrences of the identifiers in the replacement-list. If the left parenthesis following the macro name was created as the result of a macro substitution, no further substitution will take place. If the macro name appears but is not followed by a left parenthesis, no further substitution will take place.

Consider this example:

```
#define endof( string ) \
    (string + strlen( string ))
```

The \ causes the two lines to be joined together into one logical line, making this equivalent to,

```
#define endof( string )    (string + strlen( string ))
```

The function-like macro `endof` can be used to find a pointer to the null character terminating a string. The statement,

```
endptr = endof( ptr );
```

will have the macro replaced, so it will then be parsed as,

```
endptr = (ptr + strlen( ptr ));
```

Note that, in this case, the argument is evaluated twice. If `StrFn( ptr )` was specified instead of `ptr`, then the function would get called twice, because the substitution would yield,

```
endptr = (StrFn( ptr ) + strlen( StrFn( ptr ) ));
```

In gathering up the tokens used to identify the arguments, each sequence of tokens separated by a comma constitutes an argument, unless that comma happens to be within a matched pair of left and right parentheses. When a right parenthesis is found that matches the beginning left parenthesis, and the number of arguments matches the number of identifiers in the macro definition, then the gathering of the arguments is complete and the substitution takes place.

For example,

```
#define mymemcpy( dest, src, len ) \
    memcpy( dest, src, len )
/* ... */
mymemcpy( destptr, srcptr, (t=0, t=strlen(srcptr)) );
```

will, for the parameters `dest`, `src` and `len`, use the arguments `destptr`, `srcptr` and `(t=0, t=strlen(srcptr))` respectively.

This form of macro is also useful for "commenting out" a function call that is used for debugging the program. For example,

```
#define alive( where ) printf( "Alive at" where "\n" )
```

could later be replaced by,

```
#define alive( where ) /* */
```

Alternatively, the definition,

```
#define alive( where )
```

may be used. When the module or program is recompiled using this new definition for `alive`, all of the calls to `printf` made as a result of the macro replacement will disappear, without the necessity of deleting the appropriate lines in each module.

A directive of the form,

ix #undef

`#undef` *identifier*

causes the macro definition for *identifier* to be thrown away. No error is reported if no macro definition for *identifier* exists.

13.5 Argument Substitution

The argument substitution capabilities of the C preprocessor are very powerful, but can be tricky. The following sections illustrate the capabilities, and try to shed light on the problems that might be encountered.

13.5.1 Converting An Argument to a String

In the replacement-string for a function-like macro, each occurrence of `#` must be followed by a parameter to the macro. If so, both the `#` and the parameter are replaced by a string created from the characters of the argument itself, with no further substitutions performed on the argument. Each white space within the argument is converted to a single blank character. If the argument contains a character constant or string literal, any occurrences of `"` (double-quote) are replaced by `\"`, and any occurrences of `\` (backslash) are replaced by `\\`.

The following table gives a number of examples of the result of the application of the macro,

```
#define string( parm ) # parm
```

as shown in the first column:

Argument	After Substitution
`string( abc )`	`"abc"`
`string( "abc" )`	`"\"abc\""`
`string( "abc" "def" )`	`"\"abc\" \"def\""`
`string( \'/ )`	`"\\'/"`
`string( f(x) )`	`"f(x)"`

13.5.2 Concatenating Tokens

In the replacement-list, if a parameter is preceded or followed by `##`, then the parameter is replaced by the argument itself, without examining the argument for any further replacements. After all such substitutions, each `##` is removed and the tokens on either side are concatenated together. The newly formed token is then examined for further macro replacement.

`##` may not be either the first or last token in the replacement-list.

Assuming that the following macros are defined,

```
#define first     "Piece"
#define last      "of Earth"
#define firstlast "Peace on Earth"
#define first1    "Peas"
```

the following table gives a number of examples of the result of the application of the macro,

```
#define glue( x, y ) x ## y
```

as shown in the first column. For the examples that span several lines, each successive line of the "Result" column indicates successive expansions of the macros.

Argument	After Substitution
`glue( 12, 34 )`	`1234`
`glue( first, 1 )`	`first1` `"Peas"`
`glue( first, 2 )`	`first2`
`glue( first, last )`	`firstlast` `"Peace on Earth"`

13.5.3 Simple Argument Substitution

In the absence of either the `#` or `##` operators, a parameter is replaced by its argument. Before this happens, however, the argument is scanned again to see if there are any further macro substitutions to be made, applying all of the above rules. The rescanning applies *only* to the argument, not to any other tokens that might be adjacent to the argument when it replaces the parameter. In other words, if the last token of

the argument and the first token following in the replacement list together form a valid macro, no substitution of that macro will take place.

Consider the following examples, with these macro definitions in place:

```
#define f(a)   a
#define g(x)   (1+x)
#define h(s,t) s t
#define i(y)   2-y
#define xyz    printf
#define rcrs   rcrs+2
```

Invocation	After Substitution
`f(c)`	`c`
`f(f(c))`	`f(c)` `c`
`f(g(c))`	`f((1+c))` `(1+c)`
`h("hello",f("there"))`	`h("hello","there")` `"hello" "there"`
`f(xyz)("Hello\n")`	`f(printf)("Hello\n")` `printf("Hello\n")`

13.5.4 Variable Argument Macros

Macros may be defined to take optional additional parameters. This is accomplished using the `...` (ellipsis) keyword as the last parameter in the macro declaration. There may be no further parameters past the variable argument, and errors will be generated if the preprocessor finds anything other than a closing parenthesis after the ellipsis. The variable arguments may be referenced as a whole using the `__VA_ARGS__` keyword. Special behavior of pasting this parameter with a comma can result in the comma being removed (this is an extension to the standard). The only token to which this applies is a comma. Any other token which `__VA_ARGS__` is pasted with is not removed. The `__VA_ARGS__` parameter may be converted to a string using the `#` operator. Consider the following examples of macros with variable number of arguments:

```
#define shuffle1( a, b, ... )  b,__VA_ARGS__##,a
#define shuffle2( a, b, ... )  b,## __VA_ARGS__,a
#define shuffle3( a, b, ... )  b,## __VA_ARGS__##,a
#define showlist( ... )        #__VA_ARGS__
#define args( f, ... )         __VA_ARGS__
```

It is safe to assume that any time a comma is used near `__VA_ARGS__` the `##` operator should be used to paste them together. Both `shuffle1` and `shuffle2` macros are valid examples of pasting `__VA_ARGS__` together with a comma; either the leading or trailing comma may be concatenated, and if `__VA_ARGS__` is empty, the comma is removed. The macro `shuffle3` works as well; the sequence of concantenations happens from left to right, hence first the comma and empty `__VA_ARGS__` are concantenated and both are removed, afterwards the trailing comma is concatentated with `b`. Several example usages of the above macros follow:

Invocation	After Substitution
`shuffle(x,y,z)`	`y,z,x`
`shuffle(x,y)`	`y,x`
`shuffle(a,b,c,d,e)`	`b,c,d,e,a`
`showlist(x,y,z)`	`"x,y,z"`
`args("%d+%d=%d",a,b,c)`	`a,b,c`
`args("none")`	

13.5.5 Rescanning for Further Replacement

After all parameters in the replacement-list have been replaced, the resulting set of tokens is re-examined for any further replacement. If, during this scan, an apparent invocation of the macro currently being replaced is found, it is *not* replaced. Further invocations of the macro currently being replaced are not eligible for replacement until a new set of tokens from the source file, unrelated to the tokens resulting from the current substitution, are being processed.

Consider these examples, using the above macro definitions:

Invocation	After Rescanning
`f(g)(r)`	`g(r)` `(1+r)`
`f(f)(r)`	`f(r)`
`h(f,(b))`	`f (b)` `b`
`i(h(i,(b)))`	`i(i (b))` `2-i (b)`
`i(i (b))`	`i(2-b)` `2-2-b`
`rcrs`	`rcrs+2`

In other words, if an apparent invocation of a macro appears, and its name matches the macro currently being replaced, and the apparent invocation was *manufactured* by other replacements, it is *not* replaced. If, however, the apparent invocation comes directly from an argument to the macro replacement, then it *is* replaced.

After all replacements have been done, the resulting set of tokens replaces the invocation of the macro in the source file, and the file is then rescanned starting at the replacement-list. Any further macro invocations are then replaced. However, if as a result of scanning the replacement-list with following

tokens another apparent invocation of the macro just replaced is found, then that macro name is *not* replaced. An invocation of the macro will again be replaced only when a new invocation of the macro is found, unrelated to the just-replaced macro.

If the replacement-list of tokens resembles a preprocessor directive, the preprocessor will not process it.

A macro definition lasts until it is undefined (with `#undef`) or until the end of the module.

13.6 More Examples of Macro Replacement

The following examples are given in the ISO C standard, and are presented here as a complete guide to the way in which macros are replaced. The expansions are shown in stages to better illustrate the process.

The first set of examples illustrates the rules for creating string literals (using the # operator) and concatenating tokens (using the ## operator). The following definitions are used:

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", x ## s, x ## t )
#define INCFILE(n)  vers ## n  /* comment */
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"
```

The following replacements are made. The final result shows adjacent string literals joined together to form a single string. This step is not actually part of the preprocessor stage, but is given for clarity.

```
debug( 1, 2 );
   printf( "x" "1" "= %d, x" "2" "= %s", x1, x2 );
   printf( "x1= %d, x2= %s", x1, x2 );

fputs(str(strncmp("abc\0d", "abc", '\4') /* this goes away */
      == 0) str(: @\n), s);
   fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
   fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);

#include xstr(INCFILE(2).h)
   #include xstr(vers2.h)
   #include str(vers2.h)
   #include "vers2.h"
   (and then the directive is replaced by the file contents)

glue(HIGH, LOW)
   HIGHLOW
   "hello"

xglue(HIGH, LOW)
   xglue(HIGH, LOW ", world")
   glue( HIGH, LOW ", world")
   HIGHLOW ", world"
   "hello" ", world"
   "hello, world"
```

The following examples illustrate the rules for redefinition and re-examination of macros. The following definitions are used:

```
#define x    3
#define f(a) f(x * (a))
#undef  x
#define x    2
#define g    f
#define z    z[0]
#define h    g(~
#define m(a) a(w)
#define w    0,1
#define t(a) a
```

The following substitutions are made:

```
f(y+1) + f(f(z)) % t(t(g)(0) + t)(1)
f(x * (y+1)) + ...
f(2 * (y+1)) + f(f(z)) % t(t(g)(0) + t)(1)
...          + f(f(x * (z))) % ...
...          + f(f(2 * (z))) % ...
...          + f(x * (f(2 * (z)))) % ...
...          + f(2 * (f(2 * (z)))) % ...
...          + f(2 * (f(2 * (z[0])))) % t(t(g)(0) + t)(1)
...                                   % t(g(0) + t)(1)
...                                   % t(f(0) + t)(1)
...                                   % t(f(x * (0)) + t)(1)
...                                   % t(f(2 * (0)) + t)(1)
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1)
```

Another example:

```
g(2+(3,4)-w) | h 5) & m(f)^m(m)
f(2+(3,4)-w) | ...
f(2+(3,4)-0,1) | ...
f(x * (2+(3,4)-0,1)) | ...
f(2 * (2+(3,4)-0,1)) | h 5) & ...
...                  | g(~ 5) & ...
...                  | f(~ 5) & ...
...                  | f(x * (~ 5)) & ...
...                  | f(2 * (~ 5)) & m(f)^...
...                                  & f(w)^...
...                                  & f(0,1)^...
...                                  & f(x * (0,1))^...
...                                  & f(2 * (0,1))^m(m)
...                                                 ^m(w)
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1)
```

13.7 Redefining a Macro

Once a macro has been defined, its definition remains until it is explicitly undefined (using the `#undef` directive), or until the compilation of the source file is finished. If a macro is undefined, then it may be redefined in some other (or the same) way. If, during a macro replacement, the name of a macro that has been defined, undefined and then defined again is encountered, the current (most recent) definition of the macro is used, *not* the one that was in effect when the macro being replaced was defined.

Consider this example:

```
#define MAXVAL 1000
#define g(x)   CheckLimit( x, MAXVAL )

#undef  MAXVAL
#define MAXVAL 200

g( 10 );
```

This macro invocation expands to,

```
CheckLimit( 10, 200 );
```

A macro that has been defined may be redefined (without undefining it first) only if the new definition has a replacement-list that is identical to the original definition. Each preprocessing token in both the original and new replacement lists must have the same ordering and spelling, and there must be the same number of tokens. The number of spaces between tokens does not matter, unless one definition has no spaces, and the other has spaces. Comments count as one space.

The following examples illustrate valid redefinitions of macros:

```
#define OBJ_LIKE     (1-1)
#define OBJ_LIKE     /******/ (1-1) /****/
#define FN_LIKE(a)   ( a )
#define FN_LIKE( a ) (    /******/ \
                        a  /******* \
                           */ )
```

The next examples illustrate invalid redefinitions of the same macros:

```
#define OBJ_LIKE     (0)
```

The token sequence is different.

```
#define OBJ_LIKE     (1 - 1)
```

The spacing is different (none versus one).

```
#define FN_LIKE(b)   ( a )
```

The parameter is a different name, and is used differently.

```
#define FN_LIKE(b)   ( b )
```

The parameter is a different name.

13.8 Changing the Line Numbering and File Name

A directive of the form,

ix #line

`#line` *number*

sets the line number that the compiler associates with the current line in the source file to the specified number.

A directive of the form,

`#line` *number string*

sets the line number as above and also sets the name that the compiler associates with the source file that is being read to the name contained in the string.

If the directive is not recognized as one of the two forms described above, then macro substitution is performed (if possible) on the tokens on the line, and another attempt is made. If the directive still does not match one of the two forms, an error is reported.

13.9 Displaying a Diagnostic Message

A directive of the form,

ix #error

`#error` *tokens*

causes the compiler to display a diagnostic message containing the tokens from the directive.

13.10 Providing Other Information to the Compiler

A directive of the form,

ix #pragma

`#pragma` *tokens*

informs the compiler about some aspect of the compilation, in an implementation-defined manner.

A pragma operator of the form,

ix _Pragma

`_Pragma` *(string-literal)*

is an alternative method of specifying #pragma directives. For example, the following two statements are equivalent.

```
_Pragma( "library (\"kernel32.lib\")" )
#pragma library ("kernel32.lib")
```

The _Pragma operator can be used in macro definition.

```
# define LIBRARY(X) PRAGMA(library (#X))
# define PRAGMA(X) _Pragma(#X)
LIBRARY(kernel32.lib) // same as #pragma library ("kernel32.lib")
```

See the User's Guide for full details of the `#pragma` directive.

13.11 Standard Predefined Macros

The following macro names are reserved by the compiler:

`__DATE__`
: The date of translation of the source file (a string literal). The form of the date is "Mmm dd yyyy" where:

  Mmm — represents the month and is one of:

  ```
  Jan  Feb  Mar  Apr  May  Jun
  Jul  Aug  Sep  Oct  Nov  Dec
  ```

  dd — is the day of the month. The first character is a blank if the day is less than 10.

  yyyy — is the year.

  If the compiler cannot determine the current date, another date is provided.

  With Open Watcom C[16] and C[32], the current date is always available.

`__FILE__`
: The name of the current source file (a string literal). The name may be changed using the `#line` directive.

`__LINE__`
: The line number of the current source line (a decimal constant). The line number may be changed using the `#line` directive.

`__STDC__`
: The integer constant 1, indicating that the compiler is a standard-conforming implementation.

`__STDC_HOSTED__`
: The integer constant 1, indicating that the compiler is a hosted (not freestanding) implementation.

`__STDC_LIB_EXT1__`
: The long integer constant 200509L, indicating conformance to the ISO/IEC Technical Report 24731, Extensions to the C Library, Part I: Bounds-checking interfaces.

`__STDC_VERSION__`
: A decimal constant indicating the version of ISO C language standard that the compiler adheres to. Depending on compile time switches, this will be either 199901L (to indicate conformance with ISO/IEC 9899:1999) or 199409L (to indicate conformance with ISO/IEC 9899/AMD1:1995).

__TIME__
The time of translation of the source file (a string literal). The form of the time is "hh:mm:ss", with leading zeros provided for values less than 10.

If the compiler cannot determine the current time, another time is provided.

> With Open Watcom C16 and C32, the current time is always available.

__func__
The name of the current function (a string literal).

Any other macros predefined by the compiler will begin with an underscore (_) character. None of the predefined macros, nor the identifier `defined`, may be undefined (with `#undef`) or redefined (with `#define`).

13.12 Open Watcom C16 and C32 Predefined Macros

The Open Watcom C16 and C32 compilers also provide the following predefined macros for describing the memory model being used:

__COMPACT__
The compact memory model is being used.

__FLAT__
The "flat" memory model is being used for the 80386 processor. All segment registers refer to the same segment.

__FUNCTION__
The name of the current function (a string literal).

__HUGE__
The huge memory model is being used.

__LARGE__
The large memory model is being used.

__MEDIUM__
The medium memory model is being used.

__SMALL__
The small memory model is being used.

The Open Watcom C16 and C32 compilers also provide the following macros for describing the target operating system:

__DOS__
The program is being compiled for use on a DOS operating system.

__NETWARE_386__
The program is being compiled for use on the Novell Netware 386 operating system.

__NT__

The program is being compiled for use on the Windows NT operating system.

__OS2__
The program is being compiled for use on the OS/2 operating system.

__QNX__
The program is being compiled for use on the QNX operating system.

__WINDOWS__
The program is being compiled for use with Microsoft Windows.

__WINDOWS_386__
The program is being compiled for use with Microsoft Windows, using the Open Watcom 32-bit Windows interface.

The Open Watcom C16 compiler also provides the following miscellaneous macro:

__CHEAP_WINDOWS__
The program is being compiled for use with Microsoft Windows using the "zW" compiler option.

The Open Watcom C16 and C32 compilers also provide the following miscellaneous macros:

__CHAR_SIGNED__
The program is being compiled using the "j" compiler option. The default `char` type is treated as a signed quantity.

__FPI__
The program is being compiled using in-line floating point instructions.

__INLINE_FUNCTIONS__
The program is being compiled using the "oi" compiler option.

__WATCOMC__
The compiler being used is the Open Watcom C16 or Open Watcom C32 compiler. The value of the macro is the version number of the compiler times 100.

__386__
The program is being compiled for the 80386 processor, using the Open Watcom C32 compiler.

The Open Watcom C16 and C32 compilers also provide the following predefined macros for compatibility with the Microsoft C compiler, even though most of these macros do not begin with an underscore (_) character:

MSDOS
The program is being compiled for use on a DOS operating system.

_M_IX86
The program is being compiled for a specific target architecture. The macro is identically equal to 100 times the architecture compiler option value (-0, -1, -2, -3, -4, -5, etc.). If "-5" (Pentium instruction timings) was specified as a compiler option, then the value of `_M_IX86` would be 500.

M_I86
The program is being compiled for use on the Intel 80x86 processor.

M_I386

The program is being compiled for use on the Intel 80386 processor.

`M_I86CM`
The compact memory model is being used.

`M_I86HM`
The huge memory model is being used.

`M_I86LM`
The large memory model is being used.

`M_I86MM`
The medium memory model is being used.

`M_I86SM`
The small memory model is being used.

`NO_EXT_KEYS`
The program is being compiled for ISO/ANSI conformance using the "za" (no extended keywords) compiler option.

13.13 The offsetof Macro

The macro,

```
offsetof( type, member );
```

expands to a constant expression with type `size_t`. The value of the expression is the offset in bytes of *member* from the start of the structure *type*. *member* should not be a bit-field.

To use this macro, include the `<stddef.h>` header.

13.14 The NULL Macro

The `NULL` macro expands to a *null pointer constant,* which is a value that indicates a pointer does not currently point to anything.

It is recommended that `NULL`, instead of 0, be used for null pointer constants.

To use this macro, include the `<stddef.h>` header.

14 The Order of Translation

This chapter describes the sequence of steps that the C compiler takes in order to translate a set of source files. Most programmers do not need to thoroughly understand these steps, as they are intuitive. However, occasionally it will be necessary to examine the sequence to solve a problem in the translation process.

Even though the steps of translation are listed as separate phases, the compiler may combine them together. However, this should be transparent to the user.

The following are the phases of translation:

1. The characters of the source file(s) are mapped to the source character set. Any end-of-line markers used in the file system are translated, as necessary, to new-line characters. Any trigraphs are replaced by the appropriate single character.

2. Physical source lines are joined together wherever a line is terminated by a backslash (\) character. Effectively, the \ and the new-line character are deleted, creating a longer line from that record and the one following.

3. The source is broken down into preprocessing tokens and sequences of "white-space" (space and tab) characters (including comments). Each token is the longest sequence of characters that can be a token. Each comment is replaced by one white-space character. The new-line characters are retained at this point.

4. Preprocessing directives are executed and macro invocations are substituted. A header named in a `#include` directive is processed according to rules 1 to 4.

5. Members of the source character set and escape sequences in character constants and string literals are converted to single characters in the execution character set.

6. Adjacent character string literal tokens and adjacent wide string literal tokens are concatenated.

7. White-space characters separating tokens are discarded. Each preprocessing token is converted to a token. The tokens are translated according to the syntactic and semantic rules.

The final phase usually occurs outside of the compilation phase. In this phase, often called the *linking* phase, all external object definitions are resolved, and an executable program image is created. The completed image contains all the information necessary to run the program in the appropriate execution environment.

Programmer's Guide

15 Modularity

For many small programs, it is possible to write a single module which contains all of the C source for the program. This module can then be compiled, linked and run.

However, for larger applications it is not possible to maintain one module with everything in it. Or, if it is technically possible, compiling such a large module every time a change is made to the source carries too great a time penalty with it. At this point, it becomes necessary to break the program into pieces, or modules.

Dividing a program can be done quite easily. If the only issue is to reduce the size of the modules that need to be compiled, then arbitrary divisions of the code into modules will accomplish the goal.

There are other advantages, however, to planning program modularity. Some of these advantages are:

- recompilation time is reduced,
- code can be grouped into logically-connected areas, making it easier to find things,
- data structures can be hidden in one module, avoiding the temptation of letting an outside piece of code "peek" into a structure it really should not access directly,
- whole modules can be rewritten or redesigned without affecting other modules,
- areas of the code that depend on the hardware or operating system can be isolated for easy replacement when the program is ported. This may extend to replacing the module with an assembly language equivalent for increased performance.

The following sections discuss each of these points in more detail.

15.1 Reducing Recompilation Time

As discussed above, merely breaking a program into pieces will reduce the amount of time spent recompiling the source. A bug is often a simple coding error, requiring only a one or two line change. Recompiling only a small percentage of the code and relinking will be faster than recompiling everything.

Occasionally, recompiling all of the modules will be required. This usually arises when a data structure, constant, macro or other item that is used by several modules is changed. With good program design, such a change would occur in a header file, and all modules that include that header would be recompiled.

15.2 Grouping Code With Related Functionality

The best way to break programs into modules is to designate each module as having some overall purpose. For example, one module may deal exclusively with interacting with the user. Another module may manage a table of names, while yet another may process some small subset of the set of actions that may be performed by the program.

Many of the modules then become *resource managers,* and every part of the code that needs to do something significant with that resource must act through that resource manager.

Using the example of the names table manager, it is likely that the manager will need to do things like create and delete a name entry in the table. These actions would translate directly to two functions with external linkage.

By dividing up a program along lines of related functionality, it is usually easy to know where to look when a problem is being tracked.

Module names that clearly state the purpose of the module also help to locate things.

15.3 Data Hiding

Sometimes a module is written that has exclusive ownership of a data structure, such as a linked list. All other modules that wish to access the structure must call a function in the module that owns it. This technique is known as *data hiding.* The actual data is hidden in the structure, and only the *functional interface* (also called the *procedural interface)* may be used to access it. The functional interface is just the set of functions provided for accessing the structure.

The main advantage of data hiding is that the data structure may be changed with little or no impact on other modules. Also, access to the structure is controlled, leading to fewer errors because of misuse of the structure.

It is possible to have different levels of data hiding. Complete data hiding occurs when no outside module has access to the structure at all. Partial data hiding occurs when elements of the structure can be accessed, but the overall structure may not be manipulated.

Note that these rules work only if the programmer respects them. The rules are not enforced by the compiler. If a module includes a header that describes the data structures being used by another module that wants exclusive access to the structures, a rule is being broken. Whether this is good or bad depends entirely on the judgement of the programmer.

15.3.1 Complete Data Hiding

With complete data hiding, having a pointer to an element of the structure has no intrinsic value except as a parameter to the functional interface. Getting or setting a value in the structure requires a function call.

The advantage of this technique is that the complete data structure may be totally redesigned without affecting other modules. The definitions of the individual structures (`struct`'s, `union`'s, arrays) may be changed and no other module will have to be changed, or even recompiled.

The main disadvantage of complete data hiding is that even simple accesses require a function call, which is less efficient than just referencing a storage location.

Function-like macros may also be used to implement complete data hiding, avoiding the function call but hiding the true structure of the data. Recompilation of all modules may be required if the data structures change.

15.3.2 Partial Data Hiding

Partial data hiding occurs when the structure itself (for example, a linked list) is not accessible in its entirety, but elements of the structure (an element of the linked list) are accessible.

Using the names table manager as an example, it may be necessary to call the names table manager to create a name entry, but once the name is created, a pointer to the name is returned as the return value of the create function. This pointer points to a structure which is defined in a header that any module can include. Therefore, the contents of an element of the data structure can be manipulated directly.

This method is more efficient than the complete data hiding technique. However, when the structure used for the names table is changed, all modules that refer to that structure must be recompiled.

15.4 Rewriting and Redesigning Modules

With modular program design and data hiding, it is often possible to completely replace a module without affecting others. This is usually only possible when the functional interface does not change. With partial data hiding, the actual types used to implement the structure would have to remain unchanged, otherwise at least a recompilation would be required. Changing a `struct`, for example, would probably require a recompilation if only the types changed, or new members were added. If, however, the names of the members changed, or some other fundamental change occurred, then source code changes in these other modules would be necessary.

15.5 Isolating System Dependent Code in Modules

System dependencies are only relevant if the program being developed is to be run on different computers or operating systems. Isolating system dependent code is discussed more thoroughly in the chapter "Writing Portable Programs".

It is quite difficult, sometimes, to identify what constitutes system dependent code. The first time a program is ported to a new system, a number of problem areas usually arise. These areas should be carefully examined, and the code that is dependent on the host environment should be isolated. Isolation may be accomplished by placing the code in a separate module marked as system dependent, or by placing macros in the code to compile differently for the different systems.

16 Writing Portable Programs

Portable software is software that is written in such a way that it is relatively easy to get the software running on a new and different computer. By choosing the C language, the first step has been taken to reduce the effort involved in porting, but there are many other things that must be done. Some of these things include:

- isolating the portions of the code that depend on the hardware or operating system being used,
- being aware of what features of the C language are implementation-defined and avoiding them, or taking them into account,
- being aware of the various ranges of values that may be stored in certain types, and declaring objects appropriately,
- being aware of special features available on some systems that might be useful.

No programmer can seriously expect to write a large portable program the first time. The first port of the program will take a significant period of time, but the final result will be a program which is much more portable than before. Generally, each subsequent port will be easier and take less time. Of course, if the new target system has a new concept that was not considered in the original program design (such as a totally different user-interface), then porting will necessarily take longer.

16.1 Isolating System Dependent Code

The biggest problem when trying to port a program is to uncover all the places in the code where an assumption about the underlying hardware or operating system was made, and which proves to be incorrect on the new system. Many of these differences are hidden in library routines, but they can still cause problems.

Consider, for example, the issue of distinguishing between alphabetic and non-alphabetic characters. The library provides the function `isalpha` which takes a character argument and returns a non-zero value if the character is alphabetic, and 0 otherwise. Suppose a programmer, writing a FORTRAN compiler, wanted to know if a variable name started with the letters 'I' through 'N', in order to determine if it should be an integer variable. The programmer might write,

```
upletter = toupper( name[0] );
if( upletter >= 'I'  &&  upletter <= 'N' ) {
    /* ... */
}
```

If the program was being developed on a machine using the ASCII character set, this code would work fine, since the upper case letters have 26 consecutive values. However, porting the program to a machine using the EBCDIC character set, problems may arise because between the letters 'I' and 'J' are 7 other characters, including '}'. Thus, the name "}VAR" might be considered a valid integer variable name, which it is not. To solve this problem, the programmer could write,

```
if( isalpha( name[0] ) ) {
    upletter = toupper( name[0] );
    if( upletter >= 'I'  &&  upletter <= 'N' ) {
        /* ... */
    }
}
```

In this case, it is not necessary to isolate the code because a relatively simple coding change covers both cases. But there are cases where each system will require a new set of functions for some aspect of the program.

Consider the user interface of a program. If the program just displays lines of output to a scrolling terminal, and accepts lines of input in the same way, the user interface probably won't need to change between systems. But suppose the program has a sophisticated user interface involving full-screen presentation of data, windows, and menus, and uses a mouse and the keyboard for input. In the absence of standards for such interfaces, it is quite likely that each system will require a customized set of functions. Here is where program portability can become an art.

An approach to this problem is to completely isolate the user interface code of the program. The processing of data occurs independently of what appears on the screen. At the completion of processing, a function is called which updates the screen. This code may or may not be portable, depending on how many layers of functions are built between the physical screen and the generic program. At a level fairly close to the screen hardware, a set of functions should be defined which perform the set of actions that the program needs. The full set of functions will depend extensively on the requirements of the program, but they should be functions that can reasonably be expected to work on any system to which the program will eventually be ported.

Other areas that may be system dependent include:

- The behavior and capabilities of devices, including printers. Some printers support multiple fonts, expanded and compressed characters, underlining, graphics, and so on. Others support only relatively simple text output.

- Accessing memory regions outside of normally addressable storage. A good example is the Intel 80x86 family of processors. With the Open Watcom C^{16} 16-bit compiler, the addressable storage is 1024 kilobytes, but a 16-bit address can only address 64 kilobytes. Special steps must be taken when compiling in order to address the full storage space. Many compilers for the 8086, including Open Watcom C^{16} and C^{32}, introduce new keywords that describe pointer types beyond the 16-bit pointer.

- Code that has been written in assembly language for speed. As code generation technology advances, assembly language code should become less necessary.

- Code that accesses some special feature of the system. As an example, many systems provide the ability to temporarily exit to the operating system level, and later return to the program. The method of doing this varies between systems, and the requirements of the program often change as well.

- Handling the command line parameters. While C breaks the list of parameters down into strings, the interpretation of those strings may vary between systems. A program probably should attempt to conform to any conventions of the system on which it is being run.

- Handling other startup requirements. Allocation of memory, initializing devices, and so on, may be done at this point.

16.2 Beware of Long External Names

According the C Language standard, a compiler may limit external names (functions and global objects) to 6 significant characters. This limitation is often imposed by the "linking" stage of the development process.

In practice, most systems allow many more significant characters. However, the developer of a portable program should be aware of the potential for porting the program to a system that has a small limit, and name external objects accordingly.

If the developer must port a program with many names that are not unique within the limitations imposed by the target development system, the preprocessor may be used to provide shorter unique names for all objects. Note that this method may seriously impair any symbolic debugging facilities provided by the development system.

16.3 Avoiding Implementation-Defined Behavior

Several aspects of the code generated by the C compiler depend on the behavior of the particular C compiler being used. A portable program should avoid these where possible, and take them into consideration where they can't be avoided. It may be possible to use macros to avoid some of these issues.

An important behavior that varies between systems is the number of characters of external objects and functions that the system recognizes. The standard states that a system must recognize a minimum of 6 characters, although future standards may remove or extend this limit. Most systems allow more than 6 characters, but several recognize only 8 characters. For true portability, a function or object that has external linkage should be kept unique in the first 6 characters. Sometimes this requires ingenuity when thinking of names, but developing a system for naming objects goes a long way towards fitting within this restriction. The goal, of course, is to still have meaningful object names. If all systems that will eventually be used have a higher limit, then the programmer may decide to go past the 6 character limit. If a port is done to a system with the 6 character limit, a lot of source changes may be required.

To solve this problem, macros could be used to map the actual function names into more cryptic names that fit within the 6 character limit. This technique may have the adverse affect of making debugging very difficult because many of the function and object names will not be the same as contained in the source code.

Another implementation-defined behavior occurs with the type `char`. The standard does not impose a `signed` or `unsigned` interpretation on the type. A program that uses an object of type `char` that requires the values to be interpreted as signed or unsigned should explicitly declare the object with that type.

16.4 Ranges of Types

The range of an object of type `int` is not specified by the standard, except to say that the minimum range is −32767 to 32767. If an object is to contain an integer value, then thought should be given as to whether or not this range of values is acceptable on all systems. If the object is a counter that will never go outside the range 0 to 255, then the range will be adequate. However, if the object is to contain values that may exceed this range, then a `long int` may be required.

The same argument applies to objects with type `float`. It may make more sense to declare them with type `double`.

When converting floating-point numbers to integers, the rounding behavior can also vary between compilers and systems. If it is important to know how the rounding behaves, then the program should refer to the macro `FLT_ROUNDS` (defined in the header `<float.h>`), which is a value describing the type of rounding performed.

16.5 Special Features

Some systems provide special features that may or may not exist on other systems. For example, many provide the ability to exit to the operating system, run some other programs, then return to the program that was running. Other systems may not provide this ability. In an interactive program, this feature may be very useful. By isolating the code that deals with this feature, a program may remain easily portable. On the systems that don't support this feature, it may be necessary to provide a *stub* function which does nothing, or displays a message.

16.6 Using the Preprocessor to Aid Portability

The preprocessor is particularly useful for providing alternate code sequences to deal with portability issues. Conditional compilation provided by the `#if` directive allows the insertion of differing code sequences depending on some criteria. Defining a set of macros which describe the various systems, and another macro that selects a particular system, makes it easy to add system-dependent code.

For example, consider the macros,

```
#define OS_DOS    0
#define OS_CMS    1
#define OS_MVS    2
#define OS_OS2    3
#define OS_QNX    4

#define HW_IBMPC  0
#define HW_IBM370 1

#define PR_i8086  0
#define PR_370    1
```

They describe a set of operating systems `(OS)`, hardware `(HW)` and processors `(PR)`, which together can completely describe a computer and its operating system. If the program was being ported to a IBM 370 running the MVS operating system, then it could include a header defining the macros above, and declare the macros,

```
#define OPSYS     OS_MVS
#define HARDWARE  HW_IBM370
#define PROCESSOR PR_370
```

The following code sequence would include the call only if the program was being compiled for a 370 running MVS:

```
#if HARDWARE == HW_IBM370  &&  OPSYS == OS_MVS
    DoMVSStuff( x, y );
#endif
```

In other cases, code may be conditionally compiled based only on the hardware regardless of the operating system, or based only on the operating system regardless of the hardware or processor.

This technique may work well if used in moderation. However, a module that is filled with these directives becomes difficult to read, and that module becomes a candidate for being rewritten entirely for each system.

17 Avoiding Common Pitfalls

Even though a C program is much easier to write than the corresponding assembly language program, there are a few areas where most programmers make mistakes, and spend a great deal of time staring at the code trying to figure out why the program doesn't work.

The bugs that are the most difficult to find often occur when the compiler doesn't give an error or warning, but the code generated is not what the programmer expected. After a great deal of looking, the programmer spots the error and realizes that the compiler generated the correct code, but it wasn't the code that was wanted.

Some compilers, including Open Watcom C^{16} and C^{32}, have optional checking for common errors built into them, providing warnings when these conditions arise. It is probably better to eliminate the code that causes the warning than to turn off the checking done by the compiler.

The following sections illustrate several common pitfalls, and discuss how to avoid them.

17.1 Assignment Instead of Comparison

The code fragment,

```
chr = getc();
if( chr = 'a' ) {
    printf( "letter is 'a'\n" );
} else {
    printf( "letter is not 'a'\n" );
}
```

will never print the message `letter is not 'a'`, regardless of the value of `chr`.

The problem occurs in the second line of the example. The statement,

```
if( chr = 'a' ) {
```

assigns the character constant `'a'` to the object `chr`. If the value of `chr` is not zero, then the statement that is the subject of the `if` is executed.

The value of the constant `'a'` is never zero, so the first part of the `if` will always be executed. The second part might as well not even be there!

Of course, the correct way to code the second line is,

```
if( chr == 'a' ) {
```

changing the = to ==. This statement says to compare the value of `chr` against the constant `'a'` and to execute the subject of the `if` only if the values are the same.

Using one equal sign (assignment) instead of two (comparison for equality) is a common errors made by programmers, often by those who are familiar with languages such as Pascal, where the single = means "comparison for equality".

17.2 Unexpected Operator Precedence

The code fragment,

```
if( chr = getc() != EOF ) {
    printf( "The value of chr is %d\n", chr );
}
```

will always print `1`, as long as end-of-file is not detected in `getc`. The intention was to assign the value from `getc` to `chr`, then to test the value against `EOF`.

The problem occurs in the first line, which says to call the library function `getc`. The return value from `getc` (an integer value representing a character, or `EOF` if end-of-file is detected), is compared against `EOF`, and if they are not equal (it's not end-of-file), then 1 is assigned to the object `chr.` Otherwise, they are equal and 0 is assigned to `chr.` The value of `chr` is, therefore, always 0 or 1.

The correct way to write this code fragment is,

```
if( (chr = getc()) != EOF ) {
    printf( "The value of chr is %d\n", chr );
}
```

The extra parentheses force the assignment to occur first, and then the comparison for equality is done.

Note: doing assignment inside the controlling expression of loop or selection statements is not a good programming practice. These expressions tend to be difficult to read, and problems such as using = instead of == are more difficult to detect when, in some cases, = is desired.

17.3 Delayed Error From Included File

Suppose the source file `mytypes.h` contained the line,

```
typedef int COUNTER
```

and the main source file being compiled started with,

```
#include "mytypes.h"

extern int main( void )
/*********************/
{
    COUNTER x;
/* ... */
}
```

Attempting to compile the main source file would report a message such as,

```
Error! Expecting ';' but found 'extern' on line 3
```

Examining the main source file does not show any problem. The problem actually occurs in the included source file, since the `typedef` statement does not end with a semi-colon. It is this semi-colon that the compiler is expecting to find. The next token found is the `extern` keyword, so the error is reported in the main source file.

When an error occurs shortly after an `#include` directive, and the error is not readily apparent, the error may actually be caused by something in the included file.

17.4 Extra Semi-colon in Macros

The next code fragment illustrates a common error when using the preprocessor to define constants:

```
#define MAXVAL 10;

/* ... */

if( value >= MAXVAL ) break;
```

The compiler will report an error message like,

```
Error! Expecting ')' but found ';' on line 372
```

The problem is easily spotted when the macro substitution is performed on line 372. Using the definition for `MAXVAL`, the substituted version of line 372 reads,

```
if( value >= 10; ) break;
```

The semi-colon (`;`) in the definition was not treated as an end-of-statement indicator as expected, but was included in the definition of the macro (*manifest constant*) `MAXVAL`. The substitution then results in a semi-colon being placed in the middle of the controlling expression, which yields the syntax error.

17.5 The Dangling else

In the code fragment,

```
if( value1 > 0 )
    if( value2 > 0 )
        printf( "Both values greater than zero\n" );
else
    printf( "value1 is not greater than zero\n" );
```

suppose `value1` has the value 3, while `value2` has the value −7. This code fragment will cause the message,

```
value1 is not greater than zero
```

to be displayed.

The problem occurs because of the `else`. The program is indented incorrectly according to the syntax that the compiler will determine from the statements. The correct indentation should clearly show where the error lies:

```
if( value1 > 0 )
    if( value2 > 0 )
        printf( "Both values greater than zero\n" );
    else
        printf( "value1 is not greater than zero\n" );
```

The `else` belongs to the *second* `if`, not the first. Whenever there is more than one `if` statement without braces and without an `else` statement, the next `else` will be matched to the most recent `if` statement.

This code fragment clearly illustrates the usefulness of using braces to state program structure. The above example would be (correctly) written as,

```
if( value1 > 0 ) {
    if( value2 > 0 ) {
        printf( "Both values greater than zero\n" );
    }
} else {
    printf( "value1 is not greater than zero\n" );
}
```

17.6 Missing break in switch Statement

In the code fragment,

```
switch( value ) {
  case 1:
    printf( "value is 1\n" );
  default:
    printf( "value is not 1\n" );
}
```

if `value` is 1, the following output will appear:

```
value is 1
value is not 1
```

This unexpected behavior occurs because, when `value` is 1, the `switch` causes control to be passed to the `case 1:` label, where the first `printf` occurs. Then the `default` label is encountered. Labels are ignored in execution, so the next statement executed is the second `printf`.

To correct this example, it should be changed to,

```
switch( value ) {
  case 1:
    printf( "value is 1\n" );
    break;
  default:
    printf( "value is not 1\n" );
}
```

The `break` statement causes control to be passed to the statement following the closing brace of the `switch` statement.

17.7 Side-effects in Macros

In the code fragment,

```
#define endof( ptr ) ptr + strlen( ptr )
/* ... */
endptr = endof( ptr++ );
```

the statement gets expanded to,

```
endptr = ptr++ + strlen( ptr++ );
```

The parameter `ptr` gets incremented twice, rather than once as expected.

The only way to avoid this pitfall is to be aware of what macros are being used, and to be careful when using them. Several library functions may be implemented as macros on some systems. These functions include,

```
getc        putc
getchar     putchar
```

The ISO standard requires that documentation states which library functions evaluate their arguments more than once.

18 Programming Style

Programming style is as individual as a person's preference in clothing. Unfortunately, just as some programmers wouldn't win a fashion contest, some code has poor style. This code is usually easy to spot, because it is difficult to understand.

Good programming style can make the difference between programs that are easy to debug and modify, and those that you just want to avoid.

There are a number of aspects to programming style. There is no perfect style that is altogether superior to all others. Each programmer must find a style that makes him or her comfortable. The intention is to write code that is easy to read and understand, not to try to stump the next person who has to fix a problem in the code.

Good programming style will also lead to less time spent writing a program, and certainly less time spent debugging or modifying it.

The following sections discuss various aspects of programming style. They reflect the author's own biases, but they are biases based on years of hacking his way through code, mostly good and some bad, and much of it his own!

18.1 Consistency

Perhaps the most important aspect of style is *consistency*. Try, as much as possible, to use the same rules throughout the entire program. Having a mixed bag of styles within one program will confuse even the best of programmers trying to decipher the code.

If more than one programmer is involved in the project, it may be appropriate, before the first line of code is written, to discuss general rules of style. Some rules are more important than others. Make sure everyone understands the rules, and are encouraged to follow them.

18.2 Case Rules for Object and Function Names

When examining a piece of code, the scope of an object is sometimes difficult to determine. One needs to examine the declarations of objects within the function, then those declared outside of any functions, then those declared included from other source files. If no strict rules of naming objects are followed, each place will need to be laboriously searched each time.

Using mixed case object names, with strict rules, can make the job much easier. It does not matter what rules are established, as long as the rules are consistently applied throughout the program.

Consider the following sample set of rules, used throughout this book:

1. objects declared within a function with automatic storage duration are entirely in lower case,

```
int         x, counter, limit;
float       save_global;
struct s *  sptr;
```

2. objects with static storage duration (global objects) start with an upper case letter, and words or word fragments also start with upper case,

```
static int      TotalCount;
extern float    GlobalAverage;
static struct s SepStruct;
```

3. function names start with an upper case letter, and words or word fragments also start with upper case, (distinguishable from global objects by the left parenthesis),

```
extern int     TrimLength( char * ptr, int len );
static field * CreateField( char * name );
```

4. all constants are entirely in upper case.

```
#define FIELD_LIMIT 500
#define BUFSIZE     32

enum { INVALID, HELP, ADD, DELETE, REPLACE };
```

5. all `typedef` tags are in upper case.

```
typedef struct {
    float real;
    float imaginary;
} COMPLEX;
```

Thus, the storage duration and scope of each identifier can be determined without regard to context. Consider this program fragment:

```
chr = ReadChar();
if( chr != EOF ) {
    GlbChr = chr;
}
```

Using the above rules,

1. `ReadChar` is a function,

2. `chr` is an object with automatic storage duration defined within the current function,

3. `EOF` is a constant,

4. `GlbChr` is an object with static storage duration.

Note: the library functions do not use mixed case names. Also, the function `main` does not begin with an upper case `M`. Using the above coding style, library functions would stand out from other functions because of the letter-case difference.

18.3 Choose Appropriate Names

The naming of objects can be critical to the ease with which bugs can be found, or changes can be made. Using object names such as `linecount, columns` and `rownumber` will make the program more readable. Of course, short forms will creep into the code (few programmers like to type more than is really necessary), but they should be used judiciously.

Consistency of naming also helps to make the code more readable. If a structure is used throughout the program, and many different routines need a pointer to that structure, then the name of each object that points to it could be made the same. Using the example of a symbol table, the object name `symptr` might be used everywhere to mean "pointer to a symbol structure". A programmer seeing that object will automatically know what it is declared to be.

Appropriate function names are also very important. Names such as `DoIt`, while saving the original programmer from trying to think of a good name, make it more difficult for the next programmer to figure out what is going on.

18.4 Indent to Emphasize Structure

The following is a valid function:

```
static void BubbleSort( int list[], int n )
/*********************************/ { int index1
= 0; int index2; int temp; if( n < 2 )return; do {
index2 = index1 + 1; do { if( list[ index1 ] >
list[ index2 ] ) { temp = list[ index1 ]; list[
index1 ] = list[ index2 ]; list[ index2 ] = temp;
} } while( ++index2 < n ); } while( ++index1 < n-1
); }
```

(The compiler will know that it's valid, but the programmer would find it difficult to validate.) Here is the same function, but using indenting to clearly illustrate the function structure:

```
static void BubbleSort( int list[], int n )
/*************************************/
  {
    int index1 = 0;
    int index2;
    int temp;

    if( n < 2 )return;
    do {
        index2 = index1 + 1;
        do {
            if( list[ index1 ] > list[ index2 ] ) {
                temp = list[ index1 ];
                list[ index1 ] = list[ index2 ];
                list[ index2 ] = temp;
            }
        } while( ++index2 < n );
    } while( ++index1 < n-1 );
  }
```

Generally, it is good practice to indent each level of code by a consistent amount, for example 4 spaces. Thus, the subject of an `if` statement is always indented 4 spaces inside the `if`. In this manner, all loop and selection statements will stand out, making it easier to determine when the statements end.

The following are some recommended patterns to use when indenting statements. These patterns have been used throughout the book.

```
int Fn( void )
/************/
{
    /* indent 4 */
}

if( condition ) {
    /* indent 4 */
} else {
    /* indent 4 */
}

if( condition ) {
    /* indent 4 */
} else if( condition ) {
    /* indent 4 from first if */
    if( condition ) {
        /* indent 4 from nearest if */
    }
} else {
    /* indent 4 from first if */
}

switch( condition ) {
  case VALUE:
    /* indent 4 from switch */
  case VALUE:
  default:
}

do {
    /* indent 4 */
while( condition );

while( condition ) {
    /* indent 4 */
}

for( a; b; c ) {
    /* indent 4 */
}
```

Two other popular indenting styles are,

```
if( condition )
  {
    statement
  }
```

and,

```
if( condition )
{
    statements
}
```

It is not important which style is used. However, a consistent style is an asset.

18.5 Visually Align Object Declarations

A lengthy series of object declarations can be difficult to read if care is not taken to improve the readability. Consider the declarations,

```
struct flentry *flptr;
struct fldsym *sptr;
char *bufptr,*wsbuff;
int length;
```

Now, consider the same declarations, but with some visual alignment done:

```
struct flentry * flptr;
struct fldsym  * sptr;
char           * bufptr;
char           * wsbuff;
int              length;
```

It is easier to scan a list of objects when their names all begin in the same column.

18.6 Keep Functions Small

A function that is several hundred lines long can be difficult to comprehend, especially if it is being looked at on a terminal, which might only have 25 lines. Large functions also tend to have a lot of nesting of program structures, making it difficult to follow the logic.

A function that fits entirely within the terminal display can be studied and understood more easily. Program constructs don't get as complicated. Large functions often can be broken up into smaller functions which are easier to maintain.

18.7 Use static for Most Functions

Most functions do not need to be called from routines outside of the current module. Yet, if the keyword `static` is not used in the function declaration, then the function is automatically given *external linkage*. This can lead to a proliferation of external symbols, which may cause naming conflicts. Also, some linking programs may impose limitations.

Only those functions that must have external linkage should be made external. All other definitions of functions should start with the keyword `static`.

It also is a good idea to start definitions for external functions with the keyword `extern`, even though it is the default case.

18.8 Group Static Objects Together

Static objects that are declared outside of any function definition, and are used throughout the module, generally should be declared together, for example before the definition of the first function. Placing the declarations of these objects near the beginning of the module makes them easier to find.

18.9 Do Not Reuse the Names of Static Objects

If an object with static storage duration exists in one module, but has *internal linkage*, then another object with the same name should not be declared in another module. The programmer may confuse them.

Even more importantly, if an object exists with *external linkage*, a module should not declare another object with the same name with *internal linkage*. This second object will overshadow the first within the module, but the next programmer to look at the code will likely be confused.

18.10 Use Included Files to Organize Structures

Included source files can be used to organize data structures and related information. They should be used when the same structure is needed in different modules. They should even be considered when the structure is used only in one place.

Generally, each included source file should contain structures and related information for one aspect of the program. For example, a file that describes a symbol table might contain the actual structures or other types that are required, along with any manifest constants that are useful.

18.11 Use Function Prototypes

Function prototypes are very useful for eliminating common errors when calling functions. If every function in a program is prototyped (and the prototypes are used), then it is difficult to pass the wrong number or types of arguments, or to misinterpret the return value.

Using the symbol table example, the included source file that describes the symbol table structure and any related global objects or constant values could also contain the function prototypes for the functions used to

access the table. Another approach is to have separate source files containing the function prototypes, possibly using a different naming convention for the file. For example,

```
#include "symbols.h"
#include "symbols.fn"
```

would include the structures and related values from `symbols.h`, and the function prototypes from `symbols.fn`.

18.12 Do Not Do Too Much In One Statement

In the same manner that a big function that does too much can be confusing, so too can a long statement. Historically, a programmer might combine many operations into a single statement in order to get the compiler to produce better code. With current compilers, splitting the statement into two or more simpler statements will produce equivalent code, and will make the program easier to understand.

A common example of a statement that can be split is,

```
if( (c = getchar()) != EOF ) {
```

Historically, this statement might have allowed the compiler to avoid storing the value of `c` and then reloading it again to compare with `EOF`. However, the equivalent,

```
c = getchar();
if( c != EOF ) {
```

is more readable, and most compilers will produce the same code.

18.13 Do Not Use goto Too Much

The `goto` statement is a very powerful tool, but it is very easy to misuse. Here are some general rules for the use of `goto`'s:

- don't use them!

If that rule is not satisfactory, then these should be followed:

- Never `goto` a label that is above. That is the beginning of *spaghetti code*. Loop statements can always be used.
- Never `goto` the middle of a block (compound-statement). A block should always be entered by passing over the opening brace.
- Use `goto` to jump out of nested blocks, where the `break` statement is not appropriate.

Above all, keep the use of `goto`'s to a minimum.

18.14 Use Comments

Comments are crucial to good programming style. Regardless of how well the program is written, some code will be difficult to understand. Comments make it possible to give a full explanation for what the code is trying to do.

Each function definition should begin with a short comment describing what the function does.

Each module should begin with comments describing the purpose of the module. It is also a good idea to type in who wrote it, when it was written, who modified it and why, and when it was modified. This last collection of information is commonly called an *audit trail*, as it leaves a trail allowing a programmer to see the evolution of the module, along with who has been changing it.

The following audit trail is from one module in an actual product:

```
/* Modified: By:            Reason:
 * ========  ==             ======
 * 84/04/23  Dave McClurkin Initial implementation
 * 84/11/08  Jim Graham     Implemented TOTAL non-combinable;
 *                            added MAXIMUM,MINIMUM,AVERAGE
 * 84/12/12  Steve McDowell Added call to CheckBreak
 * 85/01/12  ...            Fixed overflow problems
 * 85/01/29  Alex Kachura   Saves value of TYP_ field
 * 86/01/31  Steve McDowell Switched to use of numeric accumulator
 * 86/12/10  ...            Removed some commented code
 * 87/02/24  ...            Made all commands combinable
 */
```

Appendices

A. Compiler Keywords

The following topics are discussed:

- Standard Keywords
- Open Watcom C^{16} and C^{32} Keywords

A.1 Standard Keywords

The following is the list of keywords reserved by the C language:

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

A.2 Open Watcom Extended Keywords

The Open Watcom compilers also reserve the following extended keywords:

Microsoft compilers compatible

```
__asm           __finally       __pascal
__based         __fortran       __saveregs
__cdecl         __huge          __segment
__declspec      __inline        __segname
__except        __int64         __self
__export        __interrupt     __stdcall
__far           __leave         __syscall
__far16         __loadds        __try
__fastcall      __near          __unaligned
```

IBM compilers compatible

```
_Cdecl          _Finally        _Seg16
_Except         _Leave          _Syscall
_Export         _Packed         _System
_Far16          _Pascal         _Try
_Fastcall
```

Open Watcom specific

```
__builtin_isfloat       __ow_imaginary_unit     __watcall
```

The keywords __based, __segment, __segname and __self are described in the section "Based Pointers for Open Watcom C16 and C32". Open Watcom C16 and C32 provide the predefined macro _based for convenience and compatibility with the Microsoft C compiler. It may be used in place of __based. Open Watcom C16 and C32 provide the predefined macro _segment for convenience and compatibility with the Microsoft C compiler. It may be used in place of __segment. Open Watcom C16 and C32 provide the predefined macro _segname for convenience and compatibility with the Microsoft C compiler. It may be used in place of __segname. Open Watcom C16 and C32 provide the predefined macro _self for convenience and compatibility with the Microsoft C compiler. It may be used in place of __self.

The keywords __far, __huge and __near are described in the sections "Special Pointer Types for Open Watcom C16" and "Special Pointer Types for Open Watcom C32". Open Watcom C16 and C32 provide the predefined macros far and _far for convenience and compatibility with the Microsoft C compiler. They may be used in place of __far. Open Watcom C16 and C32 provide the predefined macros huge and _huge for convenience and compatibility with the Microsoft C compiler. They may be used in place of __huge. Open Watcom C16 and C32 provide the predefined macros near and _near for convenience and compatibility with the Microsoft C compiler. They may be used in place of __near.

The keywords __far16 , _Far16 and _Seg16 are described in the section "Special Pointer Types for Open Watcom C32". Open Watcom C16 and C32 provide the predefined macro _far16 for convenience and compatibility with the Microsoft C compiler. It may be used in place of __far16.

The _Packed keyword is described in the section "Structures".

The __cdecl and _Cdecl keywords may be used with function definitions, and indicates that the calling convention for the function is the same as that used by Microsoft C. All parameters are pushed onto the stack, instead of being passed in registers. This calling convention may be controlled by a #pragma directive. See the User's Guide. Open Watcom C16 and C32 provide the predefined macros cdecl and _cdecl for convenience and compatibility with the Microsoft C compiler. They may be used in place of __cdecl.

The __fastcall and _Fastcall keywords may be used with function definitions, and indicates that the calling convention used is compatible with Microsoft C compiler. This calling convention may be controlled by a #pragma directive. Open Watcom C16 and C32 provide the predefined macro _fastcall, for convenience and compatibility with the Microsoft C compiler. It may be used in place of __fastcall See the User's Guide.

The __fortran keyword may be used with function definitions, and indicates that the calling convention for the function is suitable for calling a function written in FORTRAN. By default, this keyword has no effect. This calling convention may be controlled by a #pragma directive. See the User's Guide. Open Watcom C16 and C32 provide the predefined macros fortran and _fortran for convenience and compatibility with the Microsoft C compiler. They may be used in place of __fortran.

The __pascal and _Pascal keywords may be used with function definitions, and indicates that the calling convention for the function is suitable for calling a function written in Pascal. All parameters are pushed onto the stack, but in reverse order to the order specified by __cdecl. This calling convention may be controlled by a #pragma directive. See the User's Guide. Open Watcom C16 and C32 provide the predefined macros pascal and _pascal for convenience and compatibility with the Microsoft C compiler. They may be used in place of __pascal.

The __syscall , _Syscall and _System keywords may be used with function definitions, and indicates that the calling convention used is compatible with OS/2 (version 2.0 or higher). This calling

convention may be controlled by a #pragma directive. See the User's Guide. Open Watcom C16 and C32 provide the predefined macro _syscall for convenience and compatibility with the Microsoft C compiler. It may be used in place of __syscall.

The __stdcall keyword may be used with function definitions, and indicates that the calling convention used is compatible with Win32. This calling convention may be controlled by a #pragma directive. Open Watcom C16 and C32 provide the predefined macro _stdcall, for convenience and compatibility with the Microsoft C compiler. It may be used in place of __stdcall See the User's Guide.

The __watcall keyword may be used with function definitions, and indicates that the Open Watcom default calling convention is used. This calling convention may be controlled by a #pragma directive. See the User's Guide.

The __export and _Export keywords may be used with objects with static storage duration (global objects) and with functions, and describes that object or function as being a known object or entry point within a Dynamic Link Library in OS/2 or Microsoft Windows. The object or function must also be declared as having external linkage (using the extern keyword). In addition, any *call back* function whose address is passed to Windows (and which Windows will "call back") must be defined with the __export keyword, otherwise the call will fail and cause unpredictable results. The __export keyword may be omitted if the object or function is exported by an option specified using the linker. See the Open Watcom Linker User's Guide. Open Watcom C16 and C32 provide the predefined macro _export for convenience and compatibility with the Microsoft C compiler. It may be used in place of __export.

The __interrupt keyword may be used with function definitions for functions that handle computer interrupts. All registers are saved before the function begins execution and restored prior to returning from the interrupt. The machine language return instruction for the function is changed to iret (interrupt return). Functions written using __interrupt are suitable for attaching to the interrupt vector using the library function _dos_setvect. Open Watcom C16 and C32 provide the predefined macros interrupt and _interrupt for convenience and compatibility with the Microsoft C compiler. They may be used in place of __interrupt.

The __loadds keyword may be used with functions, and causes the compiler to generate code that will force the DS register to be set to the default data segment (DGROUP) so that near pointers will refer to that segment. This keyword is normally used with functions written for Dynamic Link Libraries in Windows and OS/2. Open Watcom C16 and C32 provide the predefined macro _loadds for convenience and compatibility with the Microsoft C compiler. It may be used in place of __loadds.

The __saveregs keyword may be used with functions. It is provided for compatibility with Microsoft C, and is used to save and restore all segment registers in Open Watcom C16 and C32. Open Watcom C16 and C32 provide the predefined macro _saveregs for convenience and compatibility with the Microsoft C compiler. It may be used in place of __saveregs.

The __try , _Try , __except , _Except , __finally , _Finally , __leave and _Leave keywords may be used for exception handling, See the "Structured Exception Handling" in User's Guide. Open Watcom C16 and C32 provide the predefined macro _try for convenience and compatibility with the Microsoft C compiler. It may be used in place of __try. Open Watcom C16 and C32 provide the predefined macro _except for convenience and compatibility with the Microsoft C compiler. It may be used in place of __except. Open Watcom C16 and C32 provide the predefined macro _finally for convenience and compatibility with the Microsoft C compiler. It may be used in place of __finally. Open Watcom C16 and C32 provide the predefined macro _leave for convenience and compatibility with the Microsoft C compiler. It may be used in place of __leave.

The __ow_imaginary_unit keyword may be used as _Imaginary constant 1.0.

The `__builtin_isfloat` keyword may be used as function for testing symbol type.

B. Trigraphs

The following is the list of trigraphs. In a C source file, all occurrences (including inside quoted strings and character constants) of any of the trigraph sequences below are replaced by the corresponding single character.

Character	Trigraph Sequence
[	??(
]	??)
{	??<
}	??>
|	??!
#	??=
\	??/
^	??'
~	??-

No other trigraphs exist. Any question mark (?) that does not belong to one of the trigraphs is not changed.

To get a sequence of characters that would otherwise be a trigraph, place a \ before the second question mark. This will cause the trigraph to be broken up so that it is not recognized, but later in the translation process, the \? will be converted to ?. For example, ?\?= will be translated to ??=.

C. Escape Sequences

The following are the escape sequences and their meanings:

Escape Sequence	Meaning
\a	Causes an audible or visual alert
\b	Back up one character
\f	Move to the start of the next page
\n	Move to the start of the next line
\r	Move to the start of the current line
\t	Move to the next horizontal tab
\v	Move to the next vertical tab

Each escape sequence maps to a single character. When such a character is sent to a display device, the action corresponding to that character is performed.

D. Operator Precedence

The table below summarizes the levels of precedence in expressions.

Expression Type	Operators
primary	identifier string constant (expression)
postfix	`a[b]` `f()` `a.b` `a->b` `a++` `a--`

unary	`sizeof u` `sizeof( a )`
	`++a` `--a` `&a` `*a`
	`+a` `-a` `~a` `!a`

cast	`(type) a`
multiplicative	`a * b` `a / b` `a % b`
additive	`a + b` `a - b`
shift	`a << b` `a >> b`
relational	`a < b` `a > b` `a <= b` `a >= b`
equality	`a == b` `a != b`
bitwise AND	`a & b`
bitwise exclusive OR	`a ^ b`
bitwise inclusive OR	`a | b`
logical AND	`a && b`
logical OR	`a || b`
conditional †	`a ? b : c`

assignment †	`a = b` `a += b` `a -= b` `a *= b`
	`a /= b` `a %= b` `a &= b` `a ^= b`
	`a |= b` `a <<= b` `a >>= b`

comma	`a,b`

† associates from right to left

Operations at a higher level in the table will occur before those below. All operators involving more than one operand associate from left to right, except for the conditional and assignment operators, which associate from right to left. Operations at the same level, except where discussed in the relevant section, may be executed in any order that the compiler chooses (subject to the usual algebraic rules). In particular, the compiler may regroup sub-expressions that are both associative and commutative in order to improve the efficiency of the code, provided the meaning (i.e. types and results) of the operands and result are not affected by the regrouping.

The order of any side-effects (for example, assignment, or action taken by a function call) is also subject to alteration by the compiler.

E. Formal C Grammar

This appendix presents the formal grammar of the C programming language. The following notation is used:

{digit}`(0)`
: Zero or more occurrences of *digit* are allowed.

{digit}`(1)`
: One or more occurrences of *digit* are allowed.

⟨integer-suffix ⟩
: *integer-suffix* is optional, with only one occurrence being allowed if present.

`A | B | C`
: Choose one of `A`, `B` or `C`.

E.1 Lexical Grammar

The following topics are discussed:

- Tokens
- Keywords
- Identifiers
- Constants
- String Literals
- Operators
- Punctuators

E.1.1 Tokens

token

	keyword
or	identifier
or	constant
or	string-literal
or	operator
or	punctuator

E.1.2 Keywords

keyword

standard-keyword
or Open Watcom-extended-keyword

standard-keyword

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

Open Watcom-extended-keyword

Microsoft compilers compatible

```
__asm                   __finally               __pascal
__based                 __fortran               __saveregs
__cdecl                 __huge                  __segment
__declspec              __inline                __segname
__except                __int64                 __self
__export                __interrupt             __stdcall
__far                   __leave                 __syscall
__far16                 __loadds                __try
__fastcall              __near                  __unaligned
```

IBM compilers compatible

```
_Cdecl                  _Finally                _Seg16
_Except                 _Leave                  _Syscall
_Export                 _Packed                 _System
_Far16                  _Pascal                 _Try
_Fastcall
```

Open Watcom specific

```
__builtin_isfloat       __ow_imaginary_unit     __watcall
```

E.1.3 Identifiers

identifier

nondigit {nondigit | digit}(0)

nondigit

a | b | ... | z | A | B | ... | Z | _

digit

`0 | 1 | ... | 9`

E.1.4 Constants

constant

floating-constant
or integer-constant
or enumeration-constant
or character-constant

floating-constant

fractional-constant ⟨exponent-part ⟩ ⟨floating-suffix ⟩
or digit-sequence exponent-part ⟨floating-suffix ⟩

exponent-part

`e | E` ⟨`+ | -` ⟩ digit-sequence

floating-suffix

`f | F | l | L`

fractional-constant

⟨digit-sequence ⟩ . digit-sequence
or digit-sequence .

digit-sequence

{digit}`(1)`

integer-constant

decimal-constant ⟨integer-suffix ⟩
or octal-constant ⟨integer-suffix ⟩
or hexadecimal-constant ⟨integer-suffix ⟩

integer-suffix

`u | U` ⟨`l | L` ⟩
or `l | L` ⟨`u | U` ⟩

decimal-constant

nonzero-digit{digit}`(0)`

nonzero-digit

`1 | 2 | ... | 9`

octal-constant

`0`{octal-digit}`(0)`

octal-digit
0 | 1 | ... | 7

hexadecimal-constant
0x|0X{hexadecimal-digit}(1)

hexadecimal-digit
0 | 1 | ... | 9 |
a | b | ... | f | A | B | ... | F

enumeration-constant
identifier

character-constant
'{c-char}(1)'
or L'{c-char}(1)'

c-char
any character in the source character set except
the single-quote ', backslash \, or new-line character
or escape-sequence

escape-sequence is one of
\' \" \\
\o \oo \ooo
\x{hexadecimal-digit}(1)
\a \b \f \n \r \t \v

E.1.5 String Literals

string-literal
"{s-char}(0)"
or L"{s-char}(0)"

s-char
any character in the source character set except
the double-quote ", backslash \, or new-line character
or escape-sequence

E.1.6 Operators

operator is one of
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
:>

E.1.7 Punctuators

punctuator
[ ] ( ) { } * , : = ; ... #

E.2 Phrase Structure Grammar

The following topics are discussed:

- Expressions
- Declarations
- Statements
- External Definitions

E.2.1 Expressions

constant-expression
conditional-expression

expression
assignment-expression{ , assignment-expression}(0)

assignment-expression
conditional-expression
or unary-expression assignment-operator assignment-expression

assignment-operator is one of
= *= /= %= += -= <<= >>= &= ^= |=

conditional-expression
logical-OR-expression ⟨? expression : conditional-expression ⟩

logical-OR-expression
logical-AND-expression{ | | logical-AND-expression}(0)

logical-AND-expression
inclusive-OR-expression {&& inclusive-OR-expression}(0)

inclusive-OR-expression
exclusive-OR-expression { | exclusive-OR-expression}(0)

exclusive-OR-expression
AND-expression {^ AND-expression}(0)

AND-expression
equality-expression {& equality-expression}(0)

equality-expression
relational-expression {==|!= relational-expression}(0)

relational-expression
shift-expression {<|>|<=|>= shift-expression}(0)

shift-expression
additive-expression {<<|>> additive-expression}(0)

additive-expression
multiplicative-expression {+|− multiplicative-expression}(0)

multiplicative-expression
cast-expression {*|/|% cast-expression}(0)

cast-expression
unary-expression
or (type-name) cast-expression

unary-expression
postfix-expression
or ++ | -- | sizeof unary-expression
or sizeof (type-name)
or unary-operator cast-expression

unary-operator is one of
& * + − ~ !

postfix-expression
primary-expression
or postfix-expression [expression]
or postfix-expression (⟨argument-expression-list ⟩)
or postfix-expression . identifier
or postfix-expression −> identifier
or postfix-expression ++
or postfix-expression −−

argument-expression-list
assignment-expression {, assignment-expression}(0)

primary-expression
identifier
or constant
or string-literal
or (expression)

E.2.2 Declarations

declaration
declaration-specifiers ⟨init-declarator-list ⟩;

declaration-specifiers
storage-class-specifier ⟨declaration-specifiers ⟩
or type-specifier ⟨declaration-specifiers ⟩

init-declarator-list
init-declarator {, init-declarator}(0)

init-declarator
declarator ⟨= initializer ⟩

storage-class-specifier
`typedef` | `extern` | `static` | `auto` | `register`

type-specifier
`void` | `char` | `short` | `int` | `long` | `float` |
`double` | `signed` | `unsigned`
or struct-or-union-specifier
or enum-specifier
or typedef-name
or type-qualifier

type-qualifier
`const` | `volatile`
or Open Watcom-type-qualifier

Open Watcom-type-qualifier

`__based`	`__fortran`	`_Seg16`
`_Cdecl`	`__huge`	`__segment`
`__cdecl`	`__inline`	`__segname`
`__declspec`	`__int64`	`__self`
`_Export`	`__interrupt`	`__stdcall`
`__export`	`__loadds`	`_Syscall`
`__far`	`__near`	`__syscall`
`_Far16`	`_Packed`	`_System`
`__far16`	`_Pascal`	`__unaligned`
`_Fastcall`	`__pascal`	`__watcall`
`__fastcall`	`__saveregs`	

struct-or-union-specifier
struct-or-union ⟨identifier ⟩ { struct-declaration-list }
or struct-or-union identifier

struct-or-union
`struct | union`

struct-declaration-list
{struct-declaration}(1)

struct-declaration
type-specifier-list struct-declarator-list;

type-specifier-list
{type-specifier}(1)

struct-declarator-list
struct-declarator {, struct-declarator}(0)

struct-declarator
declarator
or ⟨declarator ⟩ : constant-expression

enum-specifier
enum ⟨identifier ⟩ { enumerator-list }
or enum identifier

enumerator-list
enumerator {, enumerator}(0)

enumerator
enumeration-constant ⟨= constant-expression ⟩

declarator
⟨pointer ⟩ direct-declarator

direct-declarator
identifier
or (declarator)
or direct-declarator [⟨constant-expression ⟩]
or direct-declarator (parameter-type-list)
or direct-declarator (⟨identifier-list ⟩)

pointer
{* ⟨type-specifier-list ⟩}(1)

parameter-type-list
parameter-list ⟨`, ...` ⟩

parameter-list
parameter-declaration {`,` parameter-declaration}`(0)`

parameter-declaration
declaration-specifiers declarator
or declaration-specifiers ⟨abstract-declarator ⟩

identifier-list
identifier {`,` identifier}`(0)`

type-name
type-specifier-list ⟨abstract-declarator ⟩

abstract-declarator
pointer
or ⟨pointer ⟩ direct-abstract-declarator

direct-abstract-declarator
`(` abstract-declarator `)`
or ⟨direct-abstract-declarator ⟩ `[` ⟨constant-expression ⟩ `]`
or ⟨direct-abstract-declarator ⟩ `(` ⟨parameter-type-list ⟩ `)`

typedef-name
identifier

initializer
assignment-expression
or `{`initializer-list ⟨`,` ⟩`}`

initializer-list
initializer {`,` initializer}`(0)`

E.2.3 Statements

statement
labelled-statement
or compound-statement
or expression-statement
or selection-statement
or iteration-statement
or jump-statement

labelled-statement
identifier : statement
or `case` constant-expression : statement
or `default` : statement

compound-statement
{ ⟨declaration-list⟩ ⟨statement-list⟩ }

declaration-list
{declaration}(1)

statement-list
{statement}(1)

expression-statement
⟨expression⟩;

selection-statement
`if` (expression) statement
or `if` (expression) statement `else` statement
or `switch` (expression) statement

iteration-statement
`while` (expression) statement
or `do` statement `while` (expression);
or `for` (⟨expression⟩; ⟨expression⟩; ⟨expression⟩) statement

jump-statement
`goto` identifier;
or `continue;`
or `break;`
or `return` ⟨expression⟩;

E.2.4 External Definitions

file
{external-definition}(1)

external-definition
function-definition
or declaration

function-definition
⟨declaration-specifiers⟩ declarator ⟨declaration-list⟩
compound-statement

E.3 Preprocessing Directives Grammar

preprocessing-file
group

group
{group-part}(1)

group-part
⟨pp-token⟩ new-line
or if-section
or control-line

if-section
if-group {elif-group}(0) ⟨else-group⟩ endif-line

if-group
`#if` const-expression new-line ⟨group⟩
`#ifdef` identifier new-line ⟨group⟩
`#ifndef` identifier new-line ⟨group⟩

elif-group
`#elif` constant-expression new-line ⟨group⟩

else-group
`#else` new-line ⟨group⟩

endif-line
`#endif` new-line

control-line
`#include` pp-tokens new-line
`#define` identifier ⟨pp-tokens⟩ new-line
`#define` identifier `(` ⟨identifier-list⟩ `)` ⟨pp-tokens⟩ new-line
`#undef` identifier new-line
`#line` pp-tokens new-line
`#error` ⟨pp-tokens⟩ new-line
`#pragma` ⟨pp-tokens⟩ new-line
`#` new-line

pp-tokens
{preprocessing-token}(1)

preprocessing-token
header-name (only within a `#include` directive)
or identifier (no keyword distinction)
or constant
or string-literal
or operator
or punctuator
or each non-white-space character that cannot be one of the above

header-name
<{h-char}(0)>

h-char
any character in the source character set except *new-line* and >

new-line
the new-line character

F. Translation Limits

All standard-conforming C compilers must be able to translate and execute a program that contains one instance of every one of the following limits. Each limit is the minimum limit (the smallest maximum) that the compiler may impose.

The Open Watcom C^{16} and C^{32} compilers do not impose any arbitrary restrictions in any of these areas. Restrictions arise solely because of memory limitations.

- 15 nesting levels of compound statements, iteration control structures (`for`, `do/while`, `while`), and selection control structures (`if`, `switch`),
- 8 nesting levels of conditional inclusion (`#if`),
- 12 pointer, array and function declarators (in any order) modifying an arithmetic, structure, union or incomplete type in a declaration,
- 31 nesting levels of parenthesized declarators within a full declarator,
- 32 nesting levels of parenthesized expressions within a full expression,
- 31 significant initial characters in an internal identifier or a macro name,
- 6 significant initial characters in an external identifier,
- 511 external identifiers in one translation unit (module),
- 127 identifiers with block scope declared in one block,
- 1024 macro identifiers simultaneously defined in one translation unit (module),
- 31 parameters in one function definition,
- 31 arguments in one function call,
- 31 parameters in one macro definition,
- 31 parameters in one macro invocation,
- 509 characters in a logical (continued) source line,
- 509 characters in a character string literal or wide string literal (after concatenation),
- 32767 bytes in an object,
- 8 nesting levels for `#included` files,

- 257 `case` labels for a `switch` statement (excluding those for any nested `switch` statements),
- 127 members in a single structure or union,
- 127 enumeration constants in a single enumeration,
- 15 levels of nested structure or union definitions in a single struct-declaration-list (structure or union definition).

G. Macros for Numerical Limits

Although the various numerical types may have different ranges depending on the implementation of the C compiler, it is still possible to write programs that can adapt to these changing ranges. In most circumstances, it is clear whether an integer object is sufficiently large to contain all necessary values for it, regardless of whether or not the integer is only 16 bits.

However, a programmer may want to be able to conditionally compile code based on information about the range of certain types. The header `<limits.h>` defines a set of macros that describe the range of the various integer types. The header `<float.h>` defines another set of macros that describe the range and other characteristics of the various floating-point types.

G.1 Numerical Limits for Integer Types

The following macros are replaced by constant expressions that may be used in `#if` preprocessing directives. For a compiler to conform to the C language standard, the magnitude of the value of the expression provided by the compiler must equal or exceed the ISO value given below, and have the same sign. (Positive values must be greater than or equal to the ISO value. Negative values must be less than or equal to the ISO value.) The values for the actual compilers are shown following the ISO value.

• the number of bits in the smallest object that is not a bit-field (byte)

Macro: `CHAR_BIT`	Value
ISO	`>= 8`
Open Watcom C16 and C32	`8`

• the minimum value for an object of type `signed char`

Macro: `SCHAR_MIN`	Value
ISO	`<= -127`
Open Watcom C16 and C32	`-128`

• the maximum value for an object of type `signed char`

Macro: SCHAR_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 127 127

• the maximum value for an object of type unsigned char

Macro: UCHAR_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 255 255

• the minimum value for an object of type char

If char is unsigned (the default case)

Macro: CHAR_MIN	Value
ISO Open Watcom C[16] and C[32]	0 0

If char is signed (by using the command-line switch to force it to be signed), then CHAR_MIN is equivalent to SCHAR_MIN

Macro: CHAR_MIN	Value
ISO Open Watcom C[16] and C[32]	<= -127 -128

• the maximum value for an object of type char

If char is unsigned (the default case), then CHAR_MAX is equivalent to UCHAR_MAX

Macro: CHAR_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 255 255

If char is signed (by using the command-line switch to force it to be signed), then CHAR_MAX is equivalent to SCHAR_MAX

Macro: CHAR_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 127 127

• the maximum number of bytes in a multibyte character, for any supported locale

Macro: MB_LEN_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 1 2

• the minimum value for an object of type `short int`

Macro: SHRT_MIN	Value
ISO Open Watcom C[16] and C[32]	<= -32767 -32768

• the maximum value for an object of type `short int`

Macro: SHRT_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 32767 32767

• the maximum value for an object of type `unsigned short int`

Macro: USHRT_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 65535 65535

• the minimum value for an object of type `int`

Macro: INT_MIN	Value
ISO Open Watcom C[16] Open Watcom C[32]	<= -32767 -32768 -2147483648

• the maximum value for an object of type `int`

Macro: `INT_MAX`	Value
ISO	`>= 32767`
Open Watcom C[16]	`32767`
Open Watcom C[32]	`2147483647`

• the maximum value for an object of type `unsigned int`

Macro: `UINT_MAX`	Value
ISO	`>= 65535`
Open Watcom C[16]	`65535`
Open Watcom C[32]	`4294967295`

• the minimum value for an object of type `long int`

Macro: `LONG_MIN`	Value
ISO	`<= -2147483647`
Open Watcom C[16] and C[32]	`-2147483648`

• the maximum value for an object of type `long int`

Macro: `LONG_MAX`	Value
ISO	`>= 2147483647`
Open Watcom C[16] and C[32]	`2147483647`

• the maximum value for an object of type `unsigned long int`

Macro: `ULONG_MAX`	Value
ISO	`>= 4294967295`
Open Watcom C[16] and C[32]	`4294967295`

• the minimum value for an object of type `long long int`

Macro: `LLONG_MIN`	Value
ISO	`<= -9223372036854775807`
Open Watcom C[16] and C[32]	`-9223372036854775808`

• the maximum value for an object of type `long long int`

Macro: `LLONG_MAX`	Value
ISO Open Watcom C[16] and C[32]	`>= 9223372036854775807` `9223372036854775807`

• the maximum value for an object of type `unsigned long long int`

Macro: `ULLONG_MAX`	Value
ISO Open Watcom C[16] and C[32]	`>= 18446744073709551615` `18446744073709551615`

G.2 Numerical Limits for Floating-Point Types

The following macros are replaced by expressions which are not necessarily constant. For a compiler to conform to the C language standard, the magnitude of the value of the expression provided by the compiler must equal or exceed the ISO value given below, and have the same sign. (Positive values must be greater than or equal to the ISO value. Negative values must be less than or equal to the ISO value.) The values for the actual compilers are shown following the ISO value. Most compilers will exceed some of these values.

For those characteristics that have three different macros, the macros that start with `FLT_` refer to type `float`, `DBL_` refer to type `double` and `LDBL_` refer to type `long double`.

• the radix (base) of representation for the exponent

Macro: `FLT_RADIX`	Value
ISO Open Watcom C[16] and C[32]	`>= 2` `2`

• the precision, or number of digits in the floating-point mantissa, expressed in terms of the `FLT_RADIX`

Macro: `FLT_MANT_DIG`	Value
ISO Open Watcom C[16] and C[32]	no value specified `23`

Macro: DBL_MANT_DIG	Value
ISO Open Watcom C16 and C32	no value specified 52

Macro: LDBL_MANT_DIG	Value
ISO Open Watcom C16 and C32	no value specified 52

• the number of decimal digits of precision

Macro: FLT_DIG	Value
ISO Open Watcom C16 and C32	>= 6 6

Macro: DBL_DIG	Value
ISO Open Watcom C16 and C32	>= 10 15

Macro: LDBL_DIG	Value
ISO Open Watcom C16 and C32	>= 10 15

• the minimum negative integer *n* such that FLT_RADIX raised to the power *n*, minus 1, is a normalized floating-point number, or,

  • the minimum exponent value in terms of FLT_RADIX, or,
  • the base FLT_RADIX exponent for the floating-point value that is closest, but not equal, to zero

Macro: FLT_MIN_EXP	Value
ISO Open Watcom C16 and C32	no value specified -127

Macro: DBL_MIN_EXP	Value
ISO	no value specified
Open Watcom C16 and C32	-1023

Macro: LDBL_MIN_EXP	Value
ISO	no value specified
Open Watcom C16 and C32	-1023

- the minimum negative integer *n* such that 10 raised to the power *n* is in the range of normalized floating-point numbers, or,
    - the base 10 exponent for the floating-point value that is closest, but not equal, to zero

Macro: FLT_MIN_10_EXP	Value
ISO	<= -37
Open Watcom C16 and C32	-38

Macro: DBL_MIN_10_EXP	Value
ISO	<= -37
Open Watcom C16 and C32	-307

Macro: LDBL_MIN_10_EXP	Value
ISO	<= -37
Open Watcom C16 and C32	-307

- the maximum integer *n* such that FLT_RADIX raised to the power *n*, minus 1, is a representable finite floating-point number, or,
    - the maximum exponent value in terms of FLT_RADIX, or,
    - the base FLT_RADIX exponent for the largest valid floating-point value

Macro: FLT_MAX_EXP	Value
ISO	no value specified
Open Watcom C16 and C32	127

Macro: DBL_MAX_EXP	Value
ISO Open Watcom C16 and C32	no value specified 1023

Macro: LDBL_MAX_EXP	Value
ISO Open Watcom C16 and C32	no value specified 1023

- the maximum integer *n* such that 10 raised to the power *n* is a representable finite floating-point number, or,
    - the base 10 exponent for the largest valid floating-point value

Macro: FLT_MAX_10_EXP	Value
ISO Open Watcom C16 and C32	>= 37 38

Macro: DBL_MAX_10_EXP	Value
ISO Open Watcom C16 and C32	>= 37 308

Macro: LDBL_MAX_10_EXP	Value
ISO Open Watcom C16 and C32	>= 37 308

- the maximum representable finite floating-point number

Macro: FLT_MAX	Value
ISO Open Watcom C16 and C32	>= 1E+37 3.402823466E+38

Macro: DBL_MAX	Value
ISO Open Watcom C16 and C32	>= 1E+37 1.79769313486231560E+308

Macro: LDBL_MAX	Value
ISO Open Watcom C[16] and C[32]	>= 1E+37 1.79769313486231560E+308

• the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type, or,

  • the smallest number eps such that (1.0 + eps) != 1.0

Macro: FLT_EPSILON	Value
ISO Open Watcom C[16] and C[32]	<= 1E-5 1.192092896E-15

Macro: DBL_EPSILON	Value
ISO Open Watcom C[16] and C[32]	<= 1E-9 2.2204460492503131E-16

Macro: LDBL_EPSILON	Value
ISO Open Watcom C[16] and C[32]	<= 1E-9 2.2204460492503131E-16

• the minimum positive normalized floating-point number

Macro: FLT_MIN	Value
ISO Open Watcom C[16] and C[32]	<= 1E-37 1.175494351E-38

Macro: DBL_MIN	Value
ISO Open Watcom C[16] and C[32]	<= 1E-37 2.2250738585072016E-308

Macro: LDBL_MIN	Value
ISO Open Watcom C[16] and C[32]	<= 1E-37 2.2250738585072016E-308

As discussed in the section "Integer to Floating-Point Conversion", the macro FLT_ROUNDS is replaced by a constant expression whose value indicates what kind of rounding occurs following a floating-point operation. The following table gives the value of FLT_ROUNDS and its meaning:

FLT_ROUNDS	Technique
-1	indeterminable
0	toward zero
1	to nearest number
2	toward positive infinity
3	toward negative infinity

If FLT_ROUNDS has any other value, the rounding mechanism is implementation-defined.

For the Open Watcom C[16] and C[32] compiler, the value of FLT_ROUNDS is 1, meaning that floating-point values are rounded to the nearest representable number.

H. Implementation-Defined Behavior

This appendix describes the behavior of Open Watcom C^{16} and C^{32} when the standard describes the behavior as *implementation-defined.* The term describing each behavior is taken directly from the ISO/ANSI C Language standard. The numbers in parentheses at the end of each term refers to the section of the standard that discusses the behavior.

H.1 Translation

How a diagnostic is identified (5.1.1.3).

A diagnostic message appears as:

filename(*line-number*): *error-type*! *msg-number*: *msg_text*

where:

filename is the name of the source file where the error was detected. If the error was found in a file included from the source file specified on the compiler command line, then the name of the included file will appear.

line-number is the source line number in the named file where the error was detected.

error-type is either the word `Error` for errors that prevent the compile from completing successfully (no code will be generated), or `Warning` for conditions detected by the compiler that may not do what the programmer expected, but are otherwise valid. Warnings will not prevent the compiler from generating code. The issuance of warnings may be controlled by a command-line switch. See the User's Guide for details.

msg-number is the letter `E` (for errors) followed by a four digit error number, or the letter `W` (for warnings) followed by a three digit warning number. Each message has its own unique message number.

msg-text is a descriptive message indicating the problem.

Example:

```
test.c(35): Warning! W301: No prototype found for 'GetItem'
test.c(57): Error! E1009: Expecting '}' but found ','
```

H.2 Environment

The semantics of the arguments to main (5.1.2.2.1).

Each blank-separated token, except within quoted strings, on the command line is made into a string that is an element of `argv.` Quoted strings are maintained as one element.

For example, for the command line,

```
pgm 2+ 1 tokens "one token"
```

`argc` would have the value `5`, and the five elements of `argv` would be,

```
pgm
2+
1
tokens
one token
```

What constitutes an interactive device (5.1.2.3).

For Open Watcom C^{16} and C^{32}, the keyboard and the video display are considered interactive devices.

H.3 Identifiers

The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2).

Unlimited.

The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2).

The Open Watcom C^{16} and C^{32} compilers do not impose a limit. The Open Watcom Linker limits significant characters to 40.

Whether case distinctions are significant in an identifier with external linkage (6.1.2).

The Open Watcom C^{16} and C^{32} compilers produce object names in mixed case. The Open Watcom Linker provides an option to respect or ignore case when resolving linkages. By default, the linker respects case. See the Open Watcom Linker User's Guide for details.

H.4 Characters

The members of the source and execution character sets, except as explicitly specified in the standard (5.2.1).

The full IBM PC character set is available in both the source and execution character sets. The set of values between 0x20 and 0x7F are the ASCII character set.

The shift states used for the encoding of multibyte characters (5.2.1.2).

There are no shift states in the support for multibyte characters.

The number of bits in a character in the execution character set (5.2.4.2.1).

8

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4).

Both the source and execution character sets are the full IBM PC character set for whichever code page is in effect. In addition, the following table shows escape sequences available in the source character set, and what they translate to in the execution character set.

Escape Sequence	Hex Value	Meaning
\a	07	Bell or alert
\b	08	Backspace
\f	0C	Form feed
\n	0A	New-line
\r	0D	Carriage return
\t	09	Horizontal tab
\v	0B	Vertical tab
\'	27	Apostrophe or single quote
\"	22	Double quote
\?	3F	Question mark
\\	5C	Backslash
ddd		Octal value
\x*ddd*		Hexadecimal value

The value of an integer character constant that contains a character or escape sequence that is not represented in the execution character set or the extended character set for a wide character constant (6.1.3.4).

Not possible. Both the source and execution character sets are the IBM PC character set. Thus, all characters in the source character set map directly to the execution character set.

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4).

A multi-character constant is stored with the right-most character in the lowest-order (least significant) byte, and subsequent characters (moving to the left) being placed in higher-order (more significant) bytes. Up to four characters may be placed in a character constant.

The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4).

The Open Watcom C^{16} and C^{32} compilers currently support only the `"C"` locale, using North American English, and translates code page 437 to UNICODE.

To support multibyte characters, a command line switch can be used to indicate which multibyte character set to use. See the User's Guide for details.

Whether a plain `char` has the same range of values as `signed char` or `unsigned char` (6.2.1.1).

Open Watcom C[16] and C[32] treat `char` as `unsigned`, although a compiler command line switch can be used to make it `signed`.

H.5 Integers

The representations and sets of values of the various types of integers (6.1.2.5).

Integers are stored using 2's complement form. The high bit of each signed integer is a sign bit. If the sign bit is 1, the value is negative.

The ranges of the various integer types are described in the section "Integer Types".

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

When converting to a shorter type, the high-order bits of the longer value are discarded, and the remaining bits are interpreted according to the new type.

For example, converting the signed long integer `-15584170` (hexadecimal `0xFF123456`) to a signed short integer yields the result `13398` (hexadecimal `0x3456`).

When converting an unsigned integer to a signed integer of equal length, the bits are simply re-interpreted according to the new type.

For example, converting the unsigned short integer `65535` (hexadecimal `0xFFFF`) to a signed short integer yields the result `-1` (hexadecimal `0xFFFF`).

The results of bitwise operations on signed integers (6.3).

The sign bit is treated as any other bit during bitwise operations. At the completion of the operation, the new bit pattern is interpreted according to the result type.

The sign of the remainder on integer division (6.3.5).

The remainder has the same sign as the numerator (left operand).

The result of a right shift of a negative-valued signed integral type (6.3.7).

A right shift of a signed integer will leave the higher, vacated bits with the original value of the high bit. In other words, the sign bit is propogated to fill bits vacated by the shift.

For example, the result of `((short) 0x0123) >> 4` would be `0x0012`. The result of `((short) 0xFEFE) >> 4` will be `0xFFEF`.

H.6 Floating Point

The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

These are discussed in the section "Floating-Point Types". The floating-point format used is the IEEE Standard for Binary Floating-Point Arithmetic as defined in the ANSI/IEEE Standard 754-1985.

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3).

Truncation is only possible when converting a `long int` (signed or unsigned) to `float`. The 24 most-significant bits (including sign bit) are used. The 25th is examined, and if it is 1, the value is rounded up by adding one to the 24-bit value. The remaining bits are ignored.

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4).

The value is rounded to the nearest value in the smaller type.

H.7 Arrays and Pointers

The type of integer required to hold the maximum size of an array — that is, the type of the `sizeof` operator, `size_t` (6.3.3.4, 7.1.1).

```
unsigned int
```

The result of casting an integer to a pointer or vice versa (6.3.4).

Open Watcom C[16] conversion of pointer to integer:

Pointer Type	`short int` `int`	`long int`
near	result is pointer value	result is DS register in high-order 2 bytes, pointer value in low-order 2 bytes
far huge	segment is discarded, result is pointer offset (low-order 2 bytes of pointer)	result is segment in high-order 2 bytes, offset in low-order 2 bytes

Open Watcom C[16] conversion of integer to pointer:

Integer Type	near pointer	far pointer huge pointer
`short int` int	result is integer value	result segment is DS register, offset is integer value
`long int`	result is low-order 2 bytes of integer value	result segment is high-order 2 bytes, offset is low-order 2 bytes

Open Watcom C[32] conversion of pointer to integer:

Pointer Type	`short`	`int` `long int`
near	result is low-order 2 bytes of pointer value	result is pointer value
far huge	segment is discarded, result is low-order 2 bytes of pointer value	segment is discarded, result is pointer offset

Open Watcom C[32] conversion of integer to pointer:

Integer Type	near pointer	far pointer huge pointer
`short int`	result is integer value, with zeroes for high-order 2 bytes	result segment is DS register, offset is integer value, with zeroes for high-order 2 bytes
`int` long int	result is integer value	result segment is DS register, offset is integer value

The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

If the huge memory model is being used, `ptrdiff_t` has type `long int`.

For all other memory models, `ptrdiff_t` has type `int`.

If two huge pointers are subtracted and the huge memory model is not being used, then the result type will be `long int` even though `ptrdiff_t` is `int`.

H.8 Registers

The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier (6.5.1).

The Open Watcom C16 and C32 compilers may place any object that is sufficiently small, including a small structure, in one or more registers.

The number of objects that can be placed in registers varies, and is decided by the compiler. The keyword `register` does not control the placement of objects in registers.

H.9 Structures, Unions, Enumerations and Bit-Fields

A member of a union object is accessed using a member of a different type (6.3.2.3).

The behavior is undefined. Whatever bit values are present as were stored via one member will be extracted via another.

The padding and alignment of members of structures (6.5.2.1).

The Open Watcom C16 and C32 compilers align structure members by default. A command line switch, or the `pack` pragma, may be used to override the default. See the User's Guide for default values and other details.

Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.5.2.1).

`signed int`

The order of allocation of bit-fields within a unit (6.5.2.1).

Low-order (least significant) bit to high-order bit.

Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).

Bit-fields may not straddle storage-unit boundaries. If there is insufficient room to store a subsequent bit-field in a storage-unit, then it will be placed in the next storage-unit.

The integer type chosen to represent the values of an enumeration type (6.5.2.2).

By default, Open Watcom C16 and C32 will use the smallest integer type that can accommodate all values in the enumeration. The first appropriate type will be chosen according to the following table:

Type	Smallest Value	Largest Value
signed char	-128	127
unsigned char	0	255
signed short	-32768	32767
unsigned short	0	65535
signed long	-2147483648	2147483647
unsigned long	0	4294967295
signed long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

Both compilers have a command-line switch that force all enumerations to type `int`. See the User's Guide for details.

H.10 Qualifiers

What constitutes an access to an object that has volatile-qualified type (6.5.5.3).

Any reference to a volatile object is also an access to that object.

H.11 Declarators

The maximum number of declarators that may modify an arithmetic, structure or union type (6.5.4).

Limited only by available memory.

H.12 Statements

The maximum number of `case` values in a `switch` statement (6.6.4.2).

Limited only by available memory.

H.13 Preprocessing Directives

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

The character sets are the same so characters will match. Character constants are unsigned quantities, so no character will be negative.

The method for locating includable source files (6.8.2).

See the User's Guide for full details of how included files are located.

The support of quoted names for includable source files (6.8.2).

See the User's Guide for full details of how included files are located.

The mapping of source file character sequences (6.8.2).

The source and execution character sets are the same. Escape sequences are not supported in preprocessor directives.

The behavior of each recognized `#pragma` directive (6.8.6).

See the User's Guide.

The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.8.8).

The date and time are always available.

H.14 Library Functions

The null pointer constant to which the macro `NULL` expands (7.1.6).

For Open Watcom C[16], the `NULL` macro expands to `0` for the small and medium (small data) memory models, and to `0L` for the compact, large and huge (big data) memory models.

For Open Watcom C[32], the `NULL` macro expands to `0`.

The implementation-defined behavior of the library functions is described in the Open Watcom C Library Reference manual.

I. Examples of Declarations

This chapter presents a series of examples of declarations of objects and functions. Along with each example is a description that indicates how to read the declaration.

This chapter may be used as a "cookbook" for declarations. Some complicated but commonly required declarations are given here.

The first examples are very simple, and build in complexity. Some of the examples given near the end of each section are unlikely to ever be required in a real program, but hopefully they will provide an understanding of how to read and write C declarations.

To reduce the complexity and to better illustrate how a small difference in the declaration can mean a big difference in the meaning, the following rules are followed:

1. if an object is being declared, it is called `x` or `X`,
2. if a function is being declared, it is called `F`,
3. if an object is being declared, it usually has type `int`, although any other type may be substituted,
4. if a function is being declared, it usually returns type `int`, although any other type may be substituted.

Storage class specifiers (`extern`, `static`, `auto` or `register`) have purposely been omitted.

I.1 Object Declarations

Here are some examples of object (variable) declarations:

```
int x;
 2  1
 (1) x is an (2) integer.
```

```
int * x;
 3  2 1
 (1) x is a (2) pointer to an (3) integer.
```

```
int ** x;
 4  32 1
```
(1) x is a (2) pointer to a (3) pointer to an (4) integer.

```
const int x;
  2    3  1
```
(1) x is a (2) constant (3) integer.

```
int const x;
 3    2   1
```
(1) x is a (2) constant (3) integer (same as above).

```
const int * x;
  3    4  2 1
```
(1) x is a (2) pointer to a (3) constant (4) integer. The value of x may change, but the integer that it points to may not be changed. In other words, *x cannot be modified.

```
int * const x;
 4  3   2   1
```
(1) x is a (2) constant (3) pointer to an (4) integer. The value of x may not change, but the integer that it points to may change. In other words, x will always point at the same location, but the contents of that location may vary.

```
const int * const x;
  4    5  3   2   1
```
(1) x is a (2) constant (3) pointer to a (4) constant (5) integer. The value of x may not change, and the integer that it points to may not change. In other words, x will always point at the same location, which cannot be modified via x.

```
int x[];
 3  12
```
(1) x is an (2) array of (3) integers.

```
int x[53];
 4  123
```
(1) x is an (2) array of (3) 53 (4) integers.

```
int * x[];
 4  3 12
```
(1) x is an (2) array of (3) pointers to (4) integer.

```
int (*x)[];
 4   21 3
```
(1) x is a (2) pointer to an (3) array of (4) integers.

```
int * (*x)[];
 5  4  21 3
```
(1) x is a (2) pointer to an (3) array of (4) pointers to (5) integer.

```
int (*x)();
 4   21 3
```
(1)x is a (2)pointer to a (3)function returning an (4)integer.

```
int (*x[25])();
 6   4123   5
```
(1)x is an (2)array of (3)25 (4)pointers to (5)functions returning an (6)integer.

I.2 Function Declarations

Here are some examples of function declarations:

```
int F();
 3  12
```
(1)F is a (2)function returning an (3)integer.

```
int * F();
 4  3 12
```
(1)F is a (2)function returning a (3)pointer to an (4)integer.

```
int (*F())();
 5   312  4
```
(1)F is a (2)function returning a (3)pointer to a (4)function returning an (5)integer.

```
int * (*F())();
 6  5  312  4
```
(1)F is a (2)function returning a (3)pointer to a (4)function returning a (5)pointer to an (6)integer.

```
int (*F())[];
 5   312  4
```
(1)F is a (2)function returning a (3)pointer to an (4)array of (5)integers.

```
int (*(*F())[])();
 7   5 312  4  6
```
(1)F is a (2)function returning a (3)pointer to an (4)array of (5)pointers to (6)functions returning an (7)integer.

```
int * (*(*F())[])();
 8  7  5 312  4  6
```
(1)F is a (2)function returning a (3)pointer to an (4)array of (5)pointers to (6)functions returning a (7)pointer to an (8)integer.

I.3 __far, __near and __huge Declarations

The following examples illustrate the use of the `__far` and `__huge` keywords.

The use of the `__near` keyword is symmetrical with the use of the `__far` keyword, so no examples of `__near` are shown.

```
int __far X;
 3    2   1
```
(1) X is a (2) far (3) integer.

```
int * __far x;
 4  3   2   1
```
(1) x is (2) far, and is a (3) pointer to an (4) integer.

```
int __far * x;
 4    2   3 1
```
(1) x is a (2) far (3) pointer to an (4) integer.

```
int __far * __far x;
 5    3   4   2   1
```
(1) x is (2) far, and is a (3) far (4) pointer to an (5) integer.

```
int __far X[];
 4    2   13
```
(1) X is a (2) far (3) array of (4) integers.

```
int __huge X[];
 4    2    13
```
(1) x is a (2) huge (3) array of (4) integers (X is an array that can exceed 64K in size.)

```
int * __far X[];
 5  4   2   13
```
(1) X is a (2) far (3) array of (4) pointers to (5) integers.

```
int __far * X[];
 5    3   4 12
```
(1) X is an (2) array of (3) far (4) pointers to (5) integers.

```
int __far * __far X[];
 6    4   5   2   13
```
(1) X is a (2) far (3) array of (4) far (5) pointers to (6) integers.

```
int __far F();
 4    2   13
```
(1) F is a (2) far (3) function returning an (4) integer.

```
int * __far F();
 5  4   2   13
```
(1) F is a (2) far (3) function returning a (4) pointer to an (5) integer.

```
int __far * F();
 5    3   4 12
```
(1) F is a (2) function returning a (3) far (4) pointer to an (5) integer.

```
int __far * __far F();
 6    4   5   2   13
```
(1) F is a (2) far (3) function returning a (4) far (5) pointer to an (6) integer.

```
int (__far * x)();
 5     2   3 1 4
```
(1) x is a (2) far (3) pointer to a (4) function returning an (5) integer.

```
int __far * (* x)();
 6    4   5  2 1 3
```
(1) x is a (2) pointer to a (3) function returning a (4) far (5) pointer to an (6) integer.

```
int __far * (__far * x)();
 7    5   6    2   3 1 4
```
(1) x is a (2) far (3) pointer to a (4) function returning a (5) far (6) pointer to an (7) integer.

I.4 __interrupt Declarations

The following example illustrates the use of the `__interrupt` keyword.

```
void __interrupt __far F();
 5          3       2   14
```
(1) F is a (2) far (3) interrupt (4) function returning (5) nothing.

J. A Sample Program

This chapter presents an entire C program, to illustrate many of the features of the language, and to illustrate elements of programming style.

This program implements a memo system suitable for maintaining a set of memos, and displaying them on the screen. The program allows the user to display memos relevant to today's date, move through the memos adding new ones and replacing or deleting existing ones. The program displays help information whenever an invalid action is entered, or when the sole parameter to the program is a question mark.

The program is in complete conformance to the ISO C standard. It should be able to run, without modification, on any system that provides an ISO-conforming C compiler.

J.1 The memos.h File

The source file `memos.h` contains the structures used for storing the memos:

```
/* This structure is for an individual line in a memo.
 */
typedef struct text_line {
    struct text_line * next;
    char               text[1];
} TEXT_LINE;

/* This structure is the head of an individual memo.
 */
typedef struct memo_el {
    struct memo_el * prev;
    struct memo_el * next;
    TEXT_LINE *      text;
    char             date[9];
} MEMO_EL;
```

J.2 The memos.c File

The source for the program follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#include "memos.h"

/* This program implements a simple memo facility.
 * Memos may be added to a memo file, displayed
 * on the screen, and deleted.
 *
 * Modified     by              reason
 * ========     ==              ======
 * 87/10/02     Steve McDowell  Initial implementation.
 * 88/09/20     Steve McDowell  Fixed up some style issues,
 *                              introduced use of TRUE and
 *                              FALSE.
 */

/* Define some constants to make the code more readable.
 */
#define TRUE        1
#define FALSE       0
#define NULLCHAR    '\0'

static const char FileName[] = { "memos.db" };
static const char TempName[] = { "tempmemo.db" };

static MEMO_EL *  MemoHead      = NULL;
static int        MemosModified = FALSE;
static int        QuitFlag      = TRUE;

typedef enum {
    INVALID,
    HELP,
    ADD,
    DELETE,
    REPLACE,
    SHOW,
    UP,
    DOWN,
    TOP,
    TODAY,
    SAVE,
    QUIT
} ACTION;

/* This table maps action keywords onto the "actions" defined
 * above. The table also defines short forms for the keywords.
 */
typedef struct {
    ACTION act;
    char * keyword;
} ACTION_MAP;
```

```
static ACTION_MAP KeywordMap[] = {
    HELP,    "help",
    HELP,    "h",
    ADD,     "add",
    ADD,     "a",
    DELETE,  "delete",
    DELETE,  "del",
    REPLACE, "replace",
    REPLACE, "rep",
    SHOW,    "show",
    SHOW,    "sh",
    UP,      "up",
    UP,      "u",
    DOWN,    "down",
    DOWN,    "d",
    DOWN,    "",
    TOP,     "top",
    TODAY,   "today",
    TODAY,   "tod",
    SAVE,    "save",
    SAVE,    "sa",
    QUIT,    "quit",
    QUIT,    "q",

    INVALID, "" };

/* Maximum buffer length (maximum length of line of memo).
 */
#define MAXLEN 80

/* Function prototypes.
 */
static TEXT_LINE *  AddLine();
static MEMO_EL *    AddMemo();
static MEMO_EL *    DeleteMemo();
static MEMO_EL *    DoActions();
static MEMO_EL *    DoDownAction();
static MEMO_EL *    DoUpAction();
static MEMO_EL *    EnterAMemo();
static ACTION       GetAction();
static void *       MemoMAlloc();
static ACTION       PromptAction();
static ACTION       ReadAction();
static MEMO_EL *    ReadAMemo();
static MEMO_EL *    ShowTodaysMemos();

extern int main( int argc, char * argv[] )
/*************************************/
{
    int       index;
    MEMO_EL * el;

    printf( "Memo facility\n" );

/* Check for a single argument that is a question mark,
 * If found, then display the usage notes.
 */
    if( argc == 2  &&  strcmp( argv[1], "?" ) == 0 ) {
        Usage();
        exit( 0 );
    }
    ReadMemos();
    MemosModified = FALSE;
    QuitFlag      = FALSE;
```

```
/* Use the command line parameters, if any, as the first
 * actions to be performed on the memos.
 */
    el = NULL;
    for( index = 1; index < argc; ++index ) {
        el = DoActions( el, GetAction( argv[index] ) );
        if( QuitFlag ) {
            return( FALSE );
        }
    }
    HandleMemoActions( el );
    return( FALSE );
}

static void ReadMemos( void )
/***************************/

/* Read the memos file, building the structure to contain it.
 */
{
    FILE *    fid;
    MEMO_EL * new_el;
    MEMO_EL * prev_el;
    int       mcount;

    fid = fopen( FileName, "r" );
    if( fid == NULL ) {
        printf( "Memos file not found."
                " Starting with no memos.\n" );
        return;
    }

/* Loop reading entire memos.
 */
    prev_el = NULL;
    for( mcount = 0;; mcount++ ) {
        new_el = ReadAMemo( fid );
        if( new_el == NULL ) {
            printf( "%d memo(s) found.\n", mcount );
            fclose( fid );
            return;
        }
        if( prev_el == NULL ) {
            MemoHead = new_el;
            new_el->prev = NULL;
        } else {
            prev_el->next = new_el;
            new_el->prev = prev_el;
        }
        new_el->next = NULL;
        prev_el = new_el;
    }
}

static int ReadLine( char buffer[], int len, FILE * fid )
/*******************************************************/

/* Read a line from the memos file. Handle any I/O errors and
 * EOF. Return the length read, not counting the newline on
 * the end.
 */
{
    if( fgets( buffer, len, fid ) == NULL ) {
        if( feof( fid ) ) {
            return( EOF );
        }
        perror( "Error reading memos file" );
        abort();
    }
    return( strlen( buffer ) - 1 );
}
```

```
static MEMO_EL * ReadAMemo( FILE * fid )
/**************************************/

/* Read one memo, creating the memo structure and filling it
 * in. Return a pointer to the memo (NULL if none read).
 */
{
    MEMO_EL *   el;
    int         len;
    TEXT_LINE * line;
    char        buffer[MAXLEN];

    len = ReadLine( buffer, MAXLEN, fid );
    if( len == EOF ) {
        return( NULL );
    }

/* First line must be of the form "Date:" or "Date:YY/MM/DD":
 */
    if( (len != 5  &&  len != 13)
            ||  strncmp( buffer, "Date:", 5 ) != 0 ) {
        BadFormat();
    }
    buffer[len] = NULLCHAR;
    el = MemoMAlloc( sizeof( MEMO_EL ) );
    el->text = NULL;
    strcpy( el->date, buffer + 5 );
    line = NULL;
    for( ;; ) {
        len = ReadLine( buffer, MAXLEN, fid );
        if( len == EOF ) {
            BadFormat();
        }
        buffer[len] = NULLCHAR;
        if( strcmp( buffer, "====" ) == 0 ) {
            return( el );
        }
        line = AddLine( buffer, el, line );
    }
}

static TEXT_LINE * AddLine( char        buffer[],
                            MEMO_EL *   el,
                            TEXT_LINE * prevline )
/***********************************************/

/* Add a line of text to the memo, taking care of all the
 * details of modifying the structure.
 */
{
    TEXT_LINE * line;

    line = MemoMAlloc( sizeof( TEXT_LINE ) + strlen( buffer ) );
    strcpy( line->text, buffer );
    line->next = NULL;
    if( prevline == NULL ) {
        el->text = line;
    } else {
        prevline->next = line;
    }
    return( line );
}
```

```
static ACTION PromptAction( void )
/*********************************/

/* The user didn't specify an action on the command line,
 * so prompt for it.
 */
{
    ACTION act;

    for( ;; ) {
        printf( "\nEnter an action:\n" );
        act = ReadAction();
        if( act != INVALID ) {
            return( act );
        }
        printf( "\nThat selection was not valid.\n" );
        Help();
    }
}

static ACTION ReadAction( void )
/******************************/

/* Read an action from the terminal.
 * Return the action code.
 */
{
    char buffer[80];

    if( gets( buffer ) == NULL ) {
        perror( "Error reading action" );
        abort();
    }
    return( GetAction( buffer ) );
}

static ACTION GetAction( char buffer[] )
/**************************************/

/* Given the string in the buffer, return the action that
 * corresponds to it.
 * The string in the buffer is first zapped into lower case
 * so that mixed-case entries are recognized.
 */
{
    ACTION_MAP * actmap;
    char *       bufptr;

    for( bufptr = buffer; *bufptr != NULLCHAR; ++bufptr ) {
        *bufptr = tolower( *bufptr );
    }
    for( actmap = KeywordMap; actmap->act != INVALID; ++actmap ) {
        if( strcmp( buffer, actmap->keyword ) == 0 ) break;
    }
    return( actmap->act );
}

static void HandleMemoActions( MEMO_EL * el )
/*******************************************/

/* Handle all the actions entered from the keyboard.
 */
{
    for( ;; ) {
        el = DoActions( el, PromptAction() );
        if( QuitFlag ) break;
    }
}
```

```
static MEMO_EL * DoActions( MEMO_EL * el, ACTION act )
/*****************************************************/

/* Perform one action on the memos.
 */
{
    MEMO_EL * new_el;
    MEMO_EL * prev_el;

    switch( act ) {
      case HELP:
        Help();
        break;
      case ADD:
        new_el = AddMemo( el );
        if( new_el != NULL ) {
            el = new_el;
            MemosModified = TRUE;
        }
        break;
      case DELETE:
        el = DeleteMemo( el );
        MemosModified = TRUE;
        break;
      case REPLACE:
        prev_el = el;
        new_el = AddMemo( el );
        if( new_el != NULL ) {
            DeleteMemo( prev_el );
            MemosModified = TRUE;
        }
        break;
      case SHOW:
        DisplayMemo( el );
        break;
      case UP:
        el = DoUpAction( el );
        break;
      case DOWN:
        el = DoDownAction( el );
        break;
      case TOP:
        el = NULL;
        break;
      case TODAY:
        el = ShowTodaysMemos();
        break;
      case SAVE:
        if( SaveMemos() ) {
            MemosModified = FALSE;
        }
        break;
      case QUIT:
        if( WantToQuit() ) {
            QuitFlag = TRUE;
            el = NULL;
        }
    }
    return( el );
}
```

```
static MEMO_EL * AddMemo( MEMO_EL * el )
/**************************************/

/* Add a memo following the current one.
 */
{
    MEMO_EL * new_el;
    MEMO_EL * next;

    new_el = EnterAMemo();
    if( new_el == NULL ) {
        return( NULL );
    }
    if( el == NULL ) {
        next = MemoHead;
        MemoHead = new_el;
    } else {
        next = el->next;
        el->next = new_el;
    }
    new_el->prev = el;
    new_el->next = next;
    if( next != NULL ) {
        next->prev = new_el;
    }
    return( new_el );
}

static MEMO_EL * EnterAMemo( void )
/*********************************/

/* Read a memo from the keyboard, creating the memo structure
 * and filling it in. Return a pointer to the memo (NULL if
 * none read).
 */
{
    MEMO_EL *   el;
    int         len;
    TEXT_LINE * line;
    char        buffer[MAXLEN];

    printf( "What date do you want the memo displayed"
            " (YY/MM/DD)?\n" );
    if( gets( buffer ) == NULL ) {
        printf( "Error reading from terminal.\n" );
        return( NULL );
    }
    len = strlen( buffer );
    if( len != 0
            &&  (len != 8
                ||  buffer[2] != '/'
                || buffer[5] != '/') ) {
        printf( "Date is not valid.\n" );
        return( NULL );
    }
    el = MemoMAlloc( sizeof( MEMO_EL ) );
    el->text = NULL;
    strcpy( el->date, buffer );
    line = NULL;
    printf( "\nEnter the text of the memo.\n" );
    printf( "To terminate the memo,"
            " enter a line starting with =\n" );
    for( ;; ) {
        if( gets( buffer ) == NULL ) {
            printf( "Error reading from terminal.\n" );
            return( NULL );
        }
```

```
        if( buffer[0] == '=' ) {
            return( el );
        }
        line = AddLine( buffer, el, line );
    }
}

static MEMO_EL * DeleteMemo( MEMO_EL * el )
/*****************************************/

/* Delete the current memo.
 * Return a pointer to another memo, usually the following one.
 */
{
    MEMO_EL * prev;
    MEMO_EL * next;
    MEMO_EL * ret_el;

    if( el == NULL ) {
        return( MemoHead );
    }
    prev = el->prev;
    next = el->next;
    ret_el = next;
    if( ret_el == NULL ) {
        ret_el = prev;
    }

/* If it's the first memo, set a new MemoHead value.
 */
    if( prev == NULL ) {
        MemoHead = next;
        if( next != NULL ) {
            next->prev = NULL;
        }
    } else {
        prev->next = next;
        if( next != NULL ) {
            next->prev = prev;
        }
    }
    DisposeMemo( el );
    return( ret_el );
}

static MEMO_EL * DoUpAction( MEMO_EL * el )
/*****************************************/

/* Perform the UP action, including displaying the memo.
 */
{
    if( el == NULL ) {
        DisplayTop();
    } else {
        el = el->prev;
        DisplayMemo( el );
    }
    return( el );
}
```

```
static MEMO_EL * DoDownAction( MEMO_EL * el )
/*******************************************/

/* Perform the DOWN action, including displaying the memo.
 */
{
    MEMO_EL * next_el;

    next_el = (el == NULL) ? MemoHead : el->next;
    if( next_el == NULL ) {
        printf( "No more memos.\n" );
    } else {
        el = next_el;
        DisplayMemo( el );
    }
    return( el );
}

static MEMO_EL * ShowTodaysMemos( void )
/**************************************/

/* Show all memos that either:
 * (1) match today's date
 * (2) don't have a date stored.
 * Return a pointer to the last displayed memo.
 */
{
    MEMO_EL * el;
    MEMO_EL * last_el;
    time_t    timer;
    struct tm ltime;
    char      date[9];

/* Get today's time in YY/MM/DD format.
 */
    time( &timer );
    ltime = *localtime( &timer );
    strftime( date, 9, "%y/%m/%d", &ltime );
    last_el = NULL;
    for( el = MemoHead; el != NULL; el = el->next ) {
        if( el->date[0] == NULLCHAR
                ||  strcmp( date, el->date ) == 0 ) {
            DisplayMemo( el );
            last_el = el;
        }
    }
    return( last_el );
}

static void DisplayMemo( MEMO_EL * el )
/*************************************/

/* Display a memo on the screen.
 */
{
    TEXT_LINE * tline;

    if( el == NULL ) {
        DisplayTop();
        return;
    }
    if( el->date[0] == NULLCHAR ) {
        printf( "\nUndated memo\n" );
    } else {
        printf( "\nDated: %s\n", el->date );
    }
    for( tline = el->text; tline != NULL; tline = tline->next ) {
        printf( "    %s\n", tline->text );
    }
}
```

```
static int SaveMemos( void )
/**************************/

/* Save the memos to the memos file.
 */
{
    FILE *      fid;
    MEMO_EL *   el;
    TEXT_LINE * tline;
    char        buffer[20];

    if( MemoHead == NULL ) {
        printf( "No memos to save.\n" );
        return( FALSE );
    }

/* Open a temporary filename in case something goes wrong
 * during the save.
 */
    fid = fopen( TempName, "w" );
    if( fid == NULL ) {
        printf( "Unable to open \"%s\" for writing.\n", TempName );
        printf( "Save not performed.\n" );
        return( FALSE );
    }
    for( el = MemoHead; el != NULL; el = el->next ) {
        sprintf( buffer, "Date:%s", el->date );
        if( !WriteLine( buffer, fid ) ) {
            return( FALSE );
        }
        tline = el->text;
        for( ; tline != NULL; tline = tline->next ) {
            if( !WriteLine( tline->text, fid ) ) {
                return( FALSE );
            }
        }
        if( !WriteLine( "====", fid ) ) {
            return( FALSE );
        }
    }

/* Now get rid of the old file, if it's there, then rename
 * the new one.
 */
    fclose( fid );
    fid = fopen( FileName, "r" );
    if( fid != NULL ) {
        fclose( fid );
        if( remove( FileName ) != 0 ) {
            perror( "Can't remove old memos file" );
            return( FALSE );
        }
    }
    if( rename( TempName, FileName ) != 0 ) {
        perror( "Can't rename new memos file" );
        return( FALSE );
    }
    return( TRUE );
}

static int WriteLine( char * text, FILE * fid )
/*********************************************/
{
    if( fprintf( fid, "%s\n", text ) < 0 ) {
        perror( "Error writing memos file" );
        return( FALSE );
    }
    return( TRUE );
}
```

```
/* Routines for displaying HELP and other simple text.
 */

static void Usage( void )
/************************/
{
    printf( "Usage:\n" );
    printf( "    memos ?\n" );
    printf( "          displays this text\n" );
    printf( "  or\n" );
    printf( "    memos\n" );
    printf( "          prompts for all actions.\n" );
    printf( "  or\n" );
    printf( "    memos action\n" );
    printf( "          performs the action.\n" );
    printf( "          More than one action may be specified.\n" );
    printf( "          action is one of:\n" );
    ShowActions();
}

static void ShowActions( void )
/*****************************/
{
    printf( "            Help    (display this text)\n" );
    printf( "            Add     (add new memo here)\n" );
    printf( "            DELete  (delete current memo)\n" );
    printf( "            REPlace (replace current memo)\n" );
    printf( "            SHow    (show the current memo again)\n" );
    printf( "            Up      (move up one memo)\n" );
    printf( "            Down    (move down one memo)\n" );
    printf( "            TOP     (move to the top of the list\n" );
    printf( "            TODay   (display today's memos)\n" );
    printf( "            SAve    (write the memos to disk)\n" );
}

static void Help( void )
/**********************/
{
    printf( "Choose one of:\n" );
    ShowActions();
    printf( "            Quit\n" );
}

static void DisplayTop( void )
/****************************/
{
    printf( "Top of memos.\n" );
}

static int WantToQuit( void )
/***************************/

/* Check to see if the memos have been modified, but not saved.
 * If so, query the user to make sure that he/she wants to quit
 * without saving the memos.
 */
{
    char buffer[MAXLEN];

    if( !MemosModified  ||  MemoHead == NULL ) {
        return( TRUE );
    }
    printf( "\nThe memos have been modified but not saved.\n" );
    printf( "Do you want to leave without saving them?\n" );
    gets( buffer );
    return( tolower( buffer[0] ) == 'y' );
}
```

```
static void BadFormat( void )
/***************************/
{
    printf( "Invalid format for memos file\n" );
    abort();
}

static void * MemoMAlloc( int size )
/**********************************/

/* Allocate the specified size of memory, dealing with the
 * case of a failure by displaying a message and quitting.
 */
{
    register char * mem;

    mem = malloc( size );
    if( mem == NULL ) {
        printf( "Unable to allocate %d characters of memory\n",
                size );
        abort();
    }
    return( mem );

}

static void DisposeMemo( MEMO_EL * el )
/*************************************/

/* Dispose of a memo, including its lines.
 */
{
    TEXT_LINE * tline;
    TEXT_LINE * next;

    tline = el->text;
    while( tline != NULL ) {
        next = tline->next;
        free( tline );
        tline = next;
    }
    free( el );
}
```

K. Glossary

address An address is a location in a computer's memory. Each storage location (byte) has an address by which it is referenced. A *pointer* is an address.

aggregate An aggregate type is either an *array* or a *structure.* The term *aggregate* refers to the fact that arrays and structures are made up of other types.

alignment On some computers, objects such as integers, pointers and floating-point numbers may be stored only at certain addresses (for example, only at even addresses). An attempt to reference an object that is not properly aligned may cause the program to fail. Other computers may not require alignment, but may suggest it in order to increase the speed of execution of programs.

C compilers align all objects that require it, including putting padding characters within structures and arrays, if necessary. However, it is still possible for a program to attempt to reference an improperly-aligned object.

> The Open Watcom C[16] and C[32] compilers align structure members by default. A command line switch, or the `pack` pragma, may be used to control this behavior. Other objects may also be aligned by default.
>
> See the User's Guide for default values and other details.

argument An argument to a function call is an expression whose value is assigned to the parameter for the function. The function may modify the parameter, but the original argument is unaffected. This method of passing values to a function is often called *call by value.*

The argument may be a pointer to an object, in which case the function may modify the object to which the pointer points, while the argument value (the pointer) is unaffected.

array An array is a set of objects of the same type, grouped into adjacent memory locations. References to elements of the array are made by *subscripts* or *indices.*

assignment Assignment is the storing of a value into an object, which is usually done with the = operator.

automatic storage duration
An object with automatic storage duration is created when the *function* in which it is defined is invoked, and is destroyed when the function returns to the caller.

bit A bit is the smallest possible unit of information, representing one of two values, 0 or 1. If the bit is 0, it is said to be *off.* If the bit is 1, it is said to be *on.*

A bit is not representable by an *address,* but is part of a *byte,* which does have an address.

> Most processors, including the Intel 80x86 family of processors, have 8 bits in a byte.

bit-field A bit-field is a type that contains a specified number of bits.

block A block is a part of a function that begins with `{` and ends with `}` and contains declarations of objects and statements that perform some action. A block is also called a *compound statement.*

byte A byte is the smallest unit of storage representable by a unique *address,* usually capable of holding one character of information.

> Most processors, including the Intel 80x86 family of processors, have 8 *bits* in a byte.

cast To cast an object is to explicitly convert it to another *type.*

character constant
A character constant is usually one character (possibly a *trigraph* or *escape sequence)* contained within single-quotes (for example, `'a'`, `'??('` and `'\n'`).

> The Open Watcom C^{16} and C^{32} compilers allow character constants with one, two, three or four characters.

comment A comment is a sequence of characters, outside of a *string literal* or *character constant,* starting with `/*` and ending with `*/`. The comment is only examined to find the `*/` that terminates it. Hence, a comment may not contain another comment.

compiler A compiler is a program which reads a file containing programming language statements and translates it into instructions that the computer can understand.

For example, a C compiler translates statements described in this book.

compound statement
A compound statement is a part of a function that begins with `{` and ends with `}` and contains declarations of objects and statements that perform some action. A compound statement is also called a *block.*

declaration A declaration describes the attributes of an object or function, such as the storage duration, linkage, and type. The space for an object is reserved when its *definition* is found. The declaration of a function describes the function arguments and type and is also called a function prototype. The declaration of a function does not include the statements to be executed when the function is called.

decrement To decrement a number is to subtract (one) from it. To decrement a pointer is to decrease its value by the size of the object to which the pointer points.

definition A definition of an object is the same as a *declaration,* except that the storage for the object is reserved when its definition is found. A function definition includes the statements to be executed when the function is called.

exception An exception occurs when an operand to an operator has an invalid value. Division by zero is a common exception.

floating-point A floating-point number is a member of a subset of the mathematical set of real numbers, containing (possibly) a fraction and an exponent. The floating-point *type* is represented by one of the keywords `float`, `double` or `long double`.

function A function is a collection of declarations and statements, preceded by a declaration of the name of the function and the *parameters* to it, as well as a possible *return value.* The statements describe a series of steps to be taken after the function is called, and before it finishes.

header A header contains C source, usually function prototypes, structure and union definitions, linkages to externally-defined objects and macro definitions. A header is included using the `#include` preprocessor directive.

identifier An identifier is a sequence of characters, starting with a letter or underscore, and consisting of letters, digits and underscores. An identifier is used as the name of an object, a tag, function, typedef, label, macro or member of a structure or union.

implementation-defined behavior
Behavior that is implementation-defined depends on how a particular C compiler handles a certain case. All C compilers must document their behavior in these cases.

incomplete type
An incomplete type is one which has been declared, but its size or structure has not yet been stated. An example is an array of items that was declared without specifying how many items. The `void` type is also an incomplete type, but it can never be completed.

increment To increment a number is to add (one) to it. To increment a pointer is to increase its value by the size of the object to which the pointer points.

index An index (or *subscript*) is a number used to reference an element of an *array.* It is an integral value. The first element of an array has the index zero.

indirection Indirection occurs when an object that is a pointer to an object is actually used to point to it. The unary form of the `*` operator, or the `->` operator are used for indirection.

initialization The initialization of an object is the act of giving it its first (initial) value. This may be done by giving an initialization value when the object is declared, or by explicitly assigning it a value.

integer An integer is a *type* that is a subset of the mathematical set of integers. It is represented by the keyword `int`, and has a number of variations including `signed char`, `unsigned char`, `short signed int`, `short unsigned int`, `signed int`, `unsigned int`, `long signed int`, `long unsigned int`, `long long signed int` and `long long unsigned int`.

integral promotion
An object or constant that is a `char`, `short int`, `int` bit-field, or of `enum` type, that is used in an expression, is promoted to an `int` (if `int` is large enough to contain all possible values of the smaller type) or `unsigned int`.

keyword A keyword is an *identifier* that is reserved for use by the compiler. No object name or other use of an identifier may use a keyword.

label A label is an *identifier* that corresponds to a particular *statement* in a *function.* It may be used by the `goto` statement. `default` is a special label which is used with the `switch` statement.

library function A library function is a function provided with the C compiler that performs some commonly needed action. The C language standard describes a set of functions that all C compilers must provide. Whether or not the function actually generates a function call is implementation-defined.

line A line is conceptually similar to a line as seen in a text editor. The line in a text editor may be called a physical line. Several physical lines may be joined together into one logical line (or just "line") by ending all but the last line with a `\` symbol. C does not normally require statements to fit onto one line, so using the `\` symbol is usually only necessary when defining *macros.*

linkage An object with *external* linkage may be referenced by any *module* in the program. An object with *internal* linkage may be referenced only within the module in which it is defined. An object with *no* linkage may only be referenced within the *block* in which it is defined.

lint lint is a utility program, often provided with the compiler, which detects problems that the compiler will accept as syntactically valid, but likely are not what the programmer intended.

lvalue An lvalue is an expression that designates an object. The term originally comes from the assignment expression,

```
L = R
```

in which the left operand `L` to the assignment operator must be a modifiable value. The most common form of lvalue is the identifier of an object.

If an expression `E` evaluates to a pointer to an object, then `*E` is an lvalue that designates the object to which `E` points. In particular, if `E` is declared as a "pointer to `int`", then both `E` and `*E` are lvalues having the respective types "pointer to `int`" and `int`.

macro There are two kinds of macros. An object-like macro is an *identifier* that is replaced by a sequence of *tokens.* A function-like macro is an apparent function call which is replaced by a sequence of tokens.

module Referred to in the C language standard as a *translation unit,* a module is usually a file containing C source code. A module may include headers or other source files, and have conditional compilation (preprocessing directives), object declarations, and/or functions. A module is thus considered to be a C source file after the included files and conditional compilation have been processed.

name space A name space is a category of identifiers. The same identifier may appear in different name spaces. For example, the identifier `thing` may be a label, object name, tag and member of a structure or union, all at the same time, since each of these has its own name

space. The syntax of the use of the identifier resolves which category the identifier falls into.

nesting Nesting is placing something inside something else. For example, a `for` statement may, as part of its body, contain another `for` statement. The second `for` is said to be nested inside the first. Another form of nesting occurs when source files include other files.

null pointer constant
The value zero, when used in a place where a pointer type is expected, is considered to be a null pointer constant, which is a value that indicates that the pointer does not currently point to anything. The compiler interprets the zero as a special value, and does not guarantee that the actual value of the pointer will be zero.

The macro `NULL` is often used to represent the null pointer constant.

null character The character with all *bits* set to zero is used to terminate *strings,* and is called the null character. It is represented by the *escape sequence* `\0` in a string, or as the character constant `'\0'`.

object An object is a collection of *bytes* in the storage of the computer, used to represent values. The size and meaning of the object is determined by its *type.* A *scalar* object is often referred to as a *variable.*

parameter A parameter to a function is a "local copy" of the argument values determined in the call to the function. Any modification of a parameter value does not affect the argument to the function call. However, an argument (and hence a parameter) may be a pointer to an object, in which case the function may modify the object to which its parameter points.

pointer An object that contains the *address* of another object is said to be a pointer to that object.

portable Portable software is written in such a way that it is relatively easy to make the software run on different hardware or operating systems.

precedence Precedence is the set of implicit rules for determining the order of execution of an expression in the absence of parentheses.

preprocessor The preprocessor:

- examines *tokens* for *macros* and does appropriate substitutions if necessary,
- includes headers or other source files, and,
- includes or excludes input lines based on `#if` directives

before the compiler translates the source.

recursion Recursion occurs when a *function* calls itself either directly, or by calling another function which calls it. See recursion. (!)

register A register is a special part of the computer, usually not part of the addressable storage. Registers may contain values and are generally faster to use than storage.

The *keyword* `register` may be used when declaring an object with *automatic storage duration,* indicating to the compiler that this object will be heavily used, and the compiler should attempt to optimize the use of this object, possibly by placing it in a machine register.

return value A return value is the value returned by a *function* via the `return` statement.

rounding A value is rounded when the representation used to store a value is not exact. The value may be increased or decreased to the nearest value that may be accurately represented.

scalar A scalar is an object that is not a structure, union or array. Basically, it is a single item, with type such as character, any of the various integer types, or floating-point.

scope The scope of an *identifier* identifies the part of the *module* that may refererence it. An object with *block* scope may only be referenced within the block in which it is defined. An object with *file* scope may be referred to anywhere within the file in which it is defined.

sequence point A sequence point is a point at which all *side-effects* from previously executed statements will have been resolved, and no side-effects from statements not yet executed will have occurred. Normally, the programmer will not need to worry about sequence points, as it is the compiler's job to ensure that side-effects are resolved at the proper time.

side-effect A side-effect modifies a value of an object, causing a change in the state of the program. The most common side-effect is *assignment,* whereby the value of the left operand is changed.

signed A signed value can represent both negative and positive values.

The *keyword* `signed` may be used with the types `char`, `short int`, `int`, `long int` and `long long int`.

statement A statement describes the actions that are to be taken by the program. (Statements are distinct from the declarations of objects.)

static storage duration
An object with static storage duration is created when the program is invoked, and destroyed when the program exits. Any value stored in the object will remain until explicitly modified.

string A string is a sequence of characters terminated by a *null character.* A reference to a string is made with the *address* of the first character.

string literal A string literal is a sequence of zero or more characters enclosed within double-quotes and is a constant. Adjacent string literals are concatenated into one string literal. The value of a string literal is the sequence of characters within the quotes, plus a *null character* (`\0`) placed at the end.

structure A structure is a *type* which is a set of named members of (possibly different) types, which reside in memory starting at adjacent and sequentially increasing storage locations.

subscript A subscript (or *index*) is a number used to reference an element of an *array.* It is a non-negative integral value. The first element of an array has the subscript zero.

tag A tag is an identifier which names a structure, union or enumeration. In the declaration,

```
enum nums { ZERO, ONE, TWO } value;
```

`nums` is the tag of the enumeration, while `value` is an object declared with the enumeration type.

token A token is the unit used by the *preprocessor* for scanning for *macros,* and by the *compiler* for scanning the input source lines. Each identifier, constant and comment is one token, while other characters are each, individually, one token.

type The type of an *object* describes the size of the object, and what interpretation is to be used when using the value of the object. It may include information such as whether the value is `signed` or `unsigned`, and what range of values it may contain.

undefined behavior Undefined behavior occurs when an erroneous program construct or bad data is used, and the standard does not impose a behavior. Possible actions of undefined behavior include ignoring the problem, behaving in a documented manner, terminating the compilation with an error, and terminating the execution with an error.

union A union is a *type* which is a set of named members of (possibly different) types, which reside in memory starting at the same memory location.

unsigned An unsigned value is one that can represent only non-negative values.

The *keyword* `unsigned` may be used with the types `char`, `short int`, `int`, `long int` and `long long int`.

variable A variable is generally the same thing as an *object.* It is most often used to refer to *scalar* objects.

void The `void` *type* is a special type that really indicates "no particular type". An object that is a "pointer to `void`" may not be used to point at anything without it first being *cast* to the appropriate type.

The *keyword* `void` is also used as the type of a *function* that has no *return value,* and as the *parameter* list of a function that requires no parameters.

3

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W