

Waterloo C Development System
Version 3.2
for
OS, MVS and MVS/XA

User's Guide

WATCOM Products Inc.
415 Phillip Street
Waterloo, Ontario
Canada N2L 3X2

M.J. Carmody
D.W. Mulholland
E.M. Ruest
G.L. Simmons

February 1, 1989

6.1 Supported Language

The Waterloo C compiler translates programs that are written in the C language to IBM System/370 object or assembler code. It conforms, as far as practical in the OS, MVS and MVS/XA environment, to the *Draft Proposed American National Standard for Information Systems - Programming Language C*, (ANSI X3J11/88-159), December 1988.

The ANSI language definition was, to a large extent, based on *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie (Englewood Cliffs, New Jersey: Prentice-Hall, 1978); however, the newer document reflects the experience of several years of C language use. Many of the constructs that have been formalized in the ANSI draft were introduced in Version 7 of UNIX and have subsequently been adopted in most implementations of the language. Beyond what is defined in these publications, consideration has also been given to the *IEEE Trial-Use Standard Portable Operating System for Computer Environments*, (IEEE Std 1003.1), April, 1986.

UNIX is a trademark of AT&T Bell Laboratories.

6.2 ANSI Additions to C

Extensions to the Kernighan and Ritchie definition of the C language that have been incorporated into the ANSI draft standard include the following:

- enumerated data types
- structure values as function parameters and return values
- a relaxed structure member name convention.
- trigraph sequences
- the `void` data type
- new preprocessor features
- generalized rules for value preserving type promotion
- function prototypes

6.2.1 Enumerated Data Types

The enumerated data type facility, a feature common to languages such as Ada, Pascal and Modula-2, allows a set of symbolic constants to be associated with a specific type. The constants are given either a default or explicit value definition. For C enumerated types, the syntax is similar to the definition of a `struct`. For example,

```
enum colours { red, blue, yellow, green }
```

defines the tag `colours` as a data type that consists of the four member values `red`, `blue`, `yellow` and `green`. Each member is defined as an integer constant and can be used anywhere in a C program that a constant is valid. By default, the first member represents the integer constant 0, and subsequent constants are set to the previous value plus one. Alternatively, members can be given explicit values, such as the character constants `'F'` and `'M'`, as in

```
enum sex { female='F', male='M' }
```

Variables can then be declared with a tag type, as follows:

```
enum colours x
```

This implies that the variable `x` can only be assigned a member of `colours`. For example, the assignment expression

```
x = red
```

is valid; however,

```
x = 1
```

is considered a type incompatibility because `1` is not a member of `colours`.

6.2.2 Structure Parameters and Return Values

Function calls with structures as arguments cause entire structure values to be passed. Similarly, functions declared with structure return value types return structure values.

```
struct a { int b; char c; } y;
...
struct a func( struct a x )
{
    ...
    return( x );
}

int main()
{
    ...
    y = func( y );
    ...
}
```

This program passes the value of `y` when the function is called, and then assigns `y` the value returned from `func`.

6.2.3 Relaxed Structure Member Names

In the original definition of C, the names of the members for a structure were required to be unique for all member names in a program. This rule has been relaxed so that member names only have to be unique within a structure.

6.2.4 Trigraph Sequences

For any occurrence in a C source file of one of the following characters, the corresponding three character sequence could be used instead.

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

For example, the following two C strings are equivalent

```
"hello, world??/n"
"hello, world\n"
```

6.2.5 Void Data Type

`void` is a reserved keyword that specifies the type of an object that has no defined value. A special case for the `void` type occurs when a function is defined as taking a single parameter of this type. Such a function is defined as taking no parameters. For example,

```
static void func( void )
```

is a declaration of the function `func` that takes no parameters and returns no value.

A useful application of the `void` type is as the target of a pointer type object. Any pointer type object can be assigned to a `void*` type object and that value can then be assigned back to the original pointer without changing the value of the original pointer. Several of the run-time library buffer manipulation functions (such as `memchr` and `memcpy`) process parameters or return a value of type `void*`.

6.2.6 ANSI Preprocessor Directives

The following preprocessor directives are provided:

<code>#if</code>	<code>#define</code>
<code>#ifdef</code>	<code>#undef</code>
<code>#ifndef</code>	<code>#include</code>
<code>#elif</code>	<code>#line</code>
<code>#else</code>	<code>#pragma</code>
<code>#endif</code>	<code>#error</code>

The following preprocessor features may differ on some (non-standard) implementations of the preprocessor:

- Macro-function invocations must provide the same number of arguments as the macro definition specified. For example, consider the following macro definition for `times2`:

```
#define times2( param ) (param*2)
```

The following is an acceptable use of `times2`:

```
times2( 1 )
```

The following is *not* a correct use of `times2` and would be diagnosed with a compiler error message:

```
times2( 1, 2 )
```

- The `#` operator can be used within a macro function to replace a parameter value with a string literal that contains the text of the corresponding macro

argument. For example, consider the following macro definition and invocation:

```
#define same(p1, p2) func2(p1, #p2)

same( str1, str2 );
```

If the program is compiled with the PPC command line option, a file with the following expansion is generated:

```
func2(str1, "str2");
```

- The ## operator can be used within a macro function to concatenate two tokens into a single token. For example, consider the following macro definition and invocation:

```
#define same(p1, p2) func2(p1##p2)

same( str1, str2 );
```

If the program is compiled with the PPC command line option, a file with the following expansion is generated:

```
func2(str1str2);
```

- The following preprocessor macros are defined by the compiler before the first statement of the program is processed. For this example, the macros were contained in the default source input stream, which was compiled at 15:34:56 in the afternoon of September 25, 1987.

Macro	Expansion
__DATE__	"Sep 25 1987"
__TIME__	"15:34:56"
__FILE__	"SYSIN"
__LINE__	4
__STDC__	1

6.2.7 Value Preserving Type Promotions

A value of type `char` can be assigned to a value of type `int` and a value of type `float` can be assigned to a value of type `double` without a change in the value.

6.2.8 Function Prototypes

Function declarations can define the number and type of arguments that are to be passed to a function, as well as the storage class and return value type of a function. For example, the function (prototype) declaration

```
static char *mystrfunc( char *str );
```

defines the function `mystrfunc` to be a function with static storage class, returning a value of type `char*` and processing a single parameter of type `char*`.