CHAPTER XVI

ANSI/ISO Standards

If you don't appreciate the value of the C language standards, you probably don't know how lucky you are.

A C programmer can expect to take a C program developed anywhere, drop it into another compiler, and have it compile. That's not entirely true; many header files and function libraries are particular to specific compilers or specific platforms. There are a (very!) few language extensions, such as the `near` and `far` keywords and register pseudo-variables for Intel-based compilers, but even they've become standard across vendors for that platform.

If this seems to you to be the normal state of affairs, like having the accelerator pedal on the left and brakes on the right, you've lived a sheltered life. There are two different standards for BASIC, but no widespread implementation for either. The most popular Pascal compiler in the world doesn't conform to either official standard. The C++ standard that's being developed has changed so fast that it has never been backed up by a widely distributed implementation. There's a rigorous Ada standard that several implementations conform to, but Ada hasn't exactly taken the world by storm.

There are technically two C standards, one from the ANSI (American National Standards Institute) X3J11 committee and one from ISO (International Standards Organization) 9899-1990. Because the few changes ISO made supersede the ANSI document, and ANSI itself accepts the international version, it's correct to talk about "the ANSI/ISO standard."

So, how does that help you? A copy of the standard, covering both the language and the library, with commentary, is available commercially: Herbert Schildt's *The Annotated ANSI C Standard* (Osborne McGraw-Hill, ISBN 0-07-881952-0). It's a lot cheaper than most official standards, which ANSI and ISO sell to help cover the costs of establishing standards. Not every C programmer needs a copy, but nothing's more definitive than this.

The bottom line is that the ANSI/ISO standard is *the* definitive answer to the question "What is C?" If your compiler vendor does something that doesn't follow the standard, you can report it as a bug and expect little argument.

The standard doesn't cover everything. In particular, it doesn't cover a lot of interesting things a C program might do, such as graphics or multitasking. There are many competing (read "incompatible") standards to cover these areas. Maybe some will be recognized as definitive. Don't hold your breath.

By the way, there are ANSI standards for a lot of things besides programming languages. One of the many things ANSI has written a standard for is a set of escape sequences for full-screen text manipulation. That's what the MS-DOS "ANSI driver" refers to in Chapter XVII, "User Interface." (Ironically, the MS-DOS ANSI.SYS implements only a fraction of the ANSI standard sequences.)

XVI.1: Does operator precedence always work?

Answer:

The rules for operator precedence are a little complicated. In most cases, they're set up to do what you need. Arguably, a few of the rules could have been done better.

Quick review: "Operator precedence" is the collection of rules about which "operators" (such as + and = and such) take "precedence," that is, which are calculated first. In mathematics, an expression such as 2×3+4×5 is the same as (2×3)+(4×5); the multiplication happens before the addition. That means that multiplication "takes precedence over" addition, or multiplication "has higher precedence than" addition.

There are no fewer than 16 levels of operator precedence in C. It turns out having that many rules can make C programs slightly harder to read sometimes, but much easier to write. That's not the only way to make that tradeoff, but it's the C way. The levels of operator precedence are summarized in Table XVI.1.

Table XVI.1. Summary of operator precedence (highest to lowest).

Level	*Operators*
1	`x[y]` (subscript) `x(y)` (function call) `x.y` (member access) `x->y` (member pointer access) `x++` (postincrement) `x--` (postdecrement)
2	`++x` (increment) `--x` (decrement) `&x` (address-of) `*x` (pointer indirection)

Level	*Operators*
	`+x` (same as `x`, just as in mathematics) `-x` (mathematical negation) `!x` (logical negation) `~x` (bitwise negation) `sizeof x` and `sizeof(x_t)` (size in bytes)
3	`(x_t)y` (type cast)
4	`x*y` (multiplication) `x/y` (division) `x%y` (remainder)
5	`x+y` (addition) `x-y` (subtraction)
6	`x<<y` (bitwise left shift) `x>>y` (bitwise right shift)
7	`x<y`, `x>y`, `x<=y`, `x>=y` (relation comparisons)
8	`x==y`, `x!=y` (equality comparisons)
9	`x&y` (bitwise AND)
10	`x^y` (bitwise exclusive OR)
11	`x|y` (bitwise OR)
12	`x&&y` (logical AND)
13	`x||y` (logical OR)
14	`x?y:z` (conditional)
15	`x=y`, `x*=y`, `x/=y`, `x+=y`, `x-=y`, `<<=`, `>>=`, `&=`, `^=`, `|=` (assignment; right associative!)
16	`x,y` (comma)

The highest level of precedence is postfix expressions, things that go after an expression. The next highest level is prefix or unary expressions, things that go before an expression. The next highest level after that is type cast.

NOTE

The most important thing to know about operator precedence is that `*p++` means the same thing as `*(p++)`; that is, the `++` operates on the pointer, not the pointed-to thing. Code such as `*p++ = *q++` is very common in C. The precedence is the same as that for `(*(p++)) = (*(q++))`. In English, that means, "Increment `q` by one but use its old value, find the thing `q` points to, decrement `p` by one but use its old value, and assign the thing pointed to by `q` to the thing pointed to by `p`." The value of the whole expression is the value of the thing originally pointed to by `q`. You'll see code like this again and again in C, and you'll have many opportunities to write code like this. You can look up the other operator precedence rules when you can't remember them. To be a good C programmer, though, you'll have to know what `*p++` means without much conscious thought.

continues

The original C compiler was written for a computer that had instructions to handle constructs such as `*p++` and `*p++ = *q++` incredibly efficiently. As a result, a lot of C code is written that way. As a further result, because there's so much C code like that, people who design new computers make sure that there are very efficient instructions to handle these C constructs.

The next level of precedence is multiplication, division, and division remainder (also known as modulus). After that comes addition and subtraction. Just as in mathematical expressions, `2*3+4*5` means the same thing as `(2*3)+(4*5)`.

The next level of precedence is bitwise shifting.

The next levels are the relational comparisons (such as `x<y`) and then the equality comparisons (`x==y` and `x!=y`).

The next three levels are bitwise AND, exclusive OR, and OR, respectively.

NOTE

The third most important thing to know about operator precedence (after what `*p++` and `x=y=z` mean) is that `x&y==z` is *not* the same as `(x&y)==z`. Because the precedence of the bitwise operators is lower than that of the comparison operators, `x&y==z` is the same as `x&(y==z)`. That means "See whether `y` and `z` are equal (1 if they are, 0 if they aren't), then bitwise AND `x` and the result of the comparison." This is a far less likely thing to do than "bitwise AND `x` and `y` and see whether the result is equal to `z`." One might argue that the precedence of the bitwise operators should be higher than that of the comparison operators. It's about 20 years too late to do anything about it. If you want to compare the results of a bitwise operation with something else, you need parentheses.

The next levels are the logical operators, such as `x&&y` and `x||y`. Note that logical AND has higher precedence than logical OR. That reflects the way we speak in English. For example, consider this:

```
if (have_ticket && have_reservation
 || have_money && standby_ok) {
    goto_airport();
}
```

In English, you would say, "If you have a ticket and you have a reservation, or if you have money and it's OK to fly standby, go to the airport." If you override the precedence with parentheses, you have a very different condition:

```
/* not a recommended algorithm! */
if (have_ticket
  && (have_reservation || have_money)
  && standby_ok) {
    goto_airport();
}
```

In English, you would say, "If you have a ticket, and if you have a reservation or you have money, and it's OK to fly standby, go to the airport."

The next level of precedence is the conditional expression, `x?y:z`. This is an if-then-else construct that's an expression, not a statement. Sometimes conditional expressions make code much simpler; sometimes they're obscure. Conditional expressions are right associative, which means that

```
a ? b : c ? d : e
```

means the same thing as this:

```
a ? b : (c ? d : e)
```

This is very much like an else-if construct.

The next level of precedence is assignment. All the assignment operators have the same precedence. Unlike all the other C binary operators, assignment is "right associative"; it's done right to left, not left to right. `x+y+z` is the same as `(x+y)+z`, and `x*y*z` is the same as `(x*y)*z`, but `x=y=z` is the same as `x=(y=z)`.

NOTE

The second most important thing to know about operator precedence (after what `*p++` means) is what `x=y=z` means. Because assignment is right associative, it means `x=(y=z)`, or in English, "Assign the value of `z` to `y`, and then assign that value to `x`." It's very common to see code such as this:

```
a = b = c = d = 0;
```

This assigns zero to `d`, then `c`, then `b`, and finally `a`, right to left.

The lowest level of precedence in C is the comma operator. The comma operator takes two expressions, evaluates the first one, throws it away, and evaluates the second one. This makes sense only if the first expression has a side effect, such as assignment or a function call. The comma and assignment operators are often used in `for` statements:

```
for (i=0, count=0; i < MAX; ++i) {
    if (interesting(a[i]) {
        ++count;
    }
}
```

Cross Reference:

I.6: Other than in a `for` statement, when is the comma operator used?

I.12: Is left-to-right or right-to-left order guaranteed for operator precedence?

I.13: What is the difference between `++var` and `var++`?

I.14: What does the modulus operator do?

II.13: When should a type cast be used?

II.14: When should a type cast not be used?

VII.1: What is indirection?

XVI.2: Should function arguments' types be declared in the argument list of a function or immediately following?

Answer:

Function arguments should be declared in the argument list, unless you're dealing with an out-of-date compiler. In that case, you should use an `#ifdef` to do it both ways.

There are two ways to define a function. Consider two functions, `foo1` and `foo2`, that take one `char*` argument and return an integer. Say they're defined in the following way:

```
/* old style */
int
foo1(p)
char* p;
{
    /* body of function goes here */
}

/* new style */
int
foo2(char* p)
{
    /* body of function goes here */
}
```

The only advantage of the old style is that it's prettier for long argument lists.

The advantage of the new style is that it provides a function prototype as well as a function definition. Thus, if any call to `foo2` is made in the same .c file in which `foo2` is defined, after `foo2` is defined, the compiler will check the arguments in the call with the arguments in the definition. If the arguments don't match, the compiler will probably inform you that something is terribly wrong. (The standard doesn't require this step, but it occurs with most compilers.) If the arguments in the call can be converted to the arguments in the definition, they will be. That happens only if the function is defined in the new style, or if a function prototype is seen. If the function is defined in the old style and no prototype was seen, no argument conversion will be performed; probably, little or no argument checking will be done either.

The only disadvantage of the new style is that there are still compilers that don't support it. (These are mostly UNIX-based compilers that are bundled, at no extra charge, with the operating system. On the other hand, many versions of UNIX come standard with ANSI-compliant C compilers.)

If you might need to deal with non-ANSI C compilers, your best bet is to pick a macro that will be defined when prototypes and new style function definitions are supported. A header file for this macro can define it automatically, for cases in which prototypes are known to be supported:

```
#ifdef __ANSI__
#ifndef USE_PROTOS
#define USE_PROTOS 1
#endif
#endif
```

Function declarations might look like this:

```
#ifdef USE_PROTOS
int foo1(char*);
int foo2(char*);
#else
int foo1();
int foo2();
#endif
```

A function definition might look like this:

```
int
#ifdef USE_PROTOS
foo1(char* p)
#else
foo1(p)
char* p;
#endif
{
    /* body of function goes here */
}
```

If your software runs only on MS-DOS, MS-Windows, or Macintosh personal computers, don't worry about the old style; always use the new style.

Cross Reference:

VIII.1: When should I declare a function?

VIII.2: Why should I prototype a function?

XVI.3: Should programs include a prototype for *main()*?

Answer:

Programs should never include a prototype for `main`.

`main()` is a function, mostly the same as any other function. However, `main()` can be defined with at least two possible parameter lists:

```
int main(void)
```

(taking no arguments) or

```
int main(int argc, char** argv);
```

NOTE

The arguments to `main()` don't have to be called `argc` and `argv`, but they almost always are. There are better ways and places to be creative than making up new names for `main()`'s arguments.

In the second case,

`argc` is the number of arguments passed to the program at runtime,

`argv[0]` is the name of the program,

`argv[1]` through `argv[argc-1]` are the arguments passed to the program, and

`argv[argc]` is a null pointer.

There might be other legitimate definitions, such as this one:

```
int main(int argc, char** argv, char** envp);
```

`envp` is an environment list, like the one used by `getenv()`. It's terminated by a null pointer the same way `argv` is.

There's no prototype that can match all the legal definitions of `main()`. The standard says no compiler will provide a prototype for `main()`; in my opinion, you shouldn't either.

Without a prototype, your program can't explicitly call `main()` and still do argument checking. Such a call isn't forbidden by the standard, but it's probably not a good idea.

NOTE

C++ programs are explicitly forbidden from calling `main()`. (Some compilers enable you to do this; they're wrong.) The C++ compiler adds some magical code to `main()` so initialization ("construction") of global variables happens. If a C++ program could run `main()` twice, this initialization could happen twice, which would be a Bad Thing.

Cross Reference:

VIII.2: Why should I prototype a function?

XVI.4: Should *main()* always return a value?

Answer:

Yes, unless it calls `exit()`.

When a program runs, it usually terminates with some indication of success or some error code. A C program controls this indication in one (or both) of two ways, which have exactly the same effect:

It returns a value (the success or failure code) from `main()`.

It calls `exit()`, passing the success or failure code as an argument.

If the program "drops off the end of `main()`" without taking either of these actions, there's no guarantee what the success or failure code will be. This is a Bad Thing.

Whenever you write a C program, quickly check the `main()` function. The last statement should always be either a `return` statement or a call to the `exit()` function. (The only exception is when the last statement will never finish, such as an infinite `for` loop with no `break` statement. In that case, your compiler will probably complain about adding another statement that can never be reached.)

Cross Reference:

VIII.9: Is using `exit()` the same as using `return`?