Programming in C

Rex Jaeschke

Programming in C

© 1984–1996, 2001–2004, 2007, 2009, 2018 Rex Jaeschke. All rights reserved.

Edition: 5.0

The training materials associated with this book are available for license. Interested parties should contact the author.

Please address comments, corrections, and questions to the author, Rex Jaeschke, at rex@RexJaeschke.com.

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1984–1995.
2. A two-volume set of reprints called "Let's C Now", Professional Press, 1986 and 1987, ISBN 0-9614729-2-8 and 0-9614729-3-6.
3. The book "Mastering Standard C", Rex Jaeschke, Professional Press 1989, ISBN 0-9614729-8-7.
4. The book "Mastering Standard C", Rex Jaeschke, 2e, CBM Books 1996, ISBN 1-878956-55-8.

The glossary of terms from "Mastering Standard C", 1e, was spun-off and expanded into its own book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface

Welcome to the world of C. Throughout this book, we look at the statements and constructs of the C programming language. Each statement and construct is introduced by example with corresponding explanations, and, except where errors are intentional, the examples are complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind. It is intended for use both in a classroom environment as well as for self-paced learning.

Dennis Ritchie developed C around 1972 at AT&T's Bell Laboratories. C evolved from the languages CPL, BCPL, and B, in that order. The UNIX operating system, another Bell Labs development (designed by Ken Thompson and Ritchie), then was rewritten in C. (Previously, UNIX was written in assembler and B.) UNIX and C have been closely associated ever since, and every UNIX and UNIX -like operating system includes a C compiler. Today, C has a life quite separate from UNIX, with implementations of C available for every mainstream hardware and operating system platform.

C is a general-purpose, high-level language. From a grammatical viewpoint, C is a relatively simple language. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable functions while still maintaining portability, there is little need for language extensions especially considering the fact that C supports calls to procedures written in other languages. However, extensions do exist in some compilers and vary from one vendor to another. You should be aware of any nonstandard conventions in production compilers you use or evaluate. Most people learn a language by using one particular dialect of it. As a result, they are often unaware when they are using an extension.

Depending on their language backgrounds, programmers new to C may initially find programs hard to read because, traditionally, most C code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, most C language keywords must be written entirely in lowercase.[1] Keywords are also reserved words.

C supports a structured, modular approach to programming using callable subroutines, called *functions*. Source files can be compiled separately, with external references being resolved at linktime.

C lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, good code doesn't happen automatically—you must work at it. Throughout the book, I make numerous comments and suggestions regarding style. Perhaps the best advice I can give in that regard is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

[1] Starting with C99, new kewords are spelled with a leading underscore followed by an uppercase letter, followed by lowercase letters and underscores, as in `_Bool` and `_Thread_local`.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

Tip: Avoid initializing enumeration constants explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it. C is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages (including Java and C#), C is a relatively expensive language to learn and master.

Reader Assumptions

This is *not* a first course in programming. (Nor do I recommend C as a first programming language.)

I assume that you know how to use your text editor, C compiler, and debugger. Comments on the use of these utility programs will be limited to points of interest to the C programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.
- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and how to do formatted and unformatted I/O.
- Basic data structures such as linked lists.

If your programming background is in some procedural language such as PL/I, Pascal, FORTRAN, or BASIC, you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of functions and their associated argument passing and value returning machinery. If you are coming from scripting languages, Java, or C#, many things look familiar; however, C's pointer syntax and preprocessor use will be different.

Limitations

This book covers almost all the C language. It also introduces the core class library. However, a very small percentage of library facilities are mentioned or covered in any detail. The Standard C library contains so many functions that whole books have been written about that subject alone.

This book is directed at teaching the C language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced, solutions.

GUI, calling non-C routines, threading, inter-process communications, operating system-specific, and many other advanced features are outside the scope of this text.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for more than 20 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program may well have its own subdirectory. For example, the source code for the program called ba04 in the "Basics" chapter can be found in the following directory hierarchy: Source, Basics, ba04. By convention, the names of C source files end in "`.c`".

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1] For example, lab solution lbba01 in the "Basics" chapter has the following fully qualified directory hierarchy: Labs, Basics, lbba01.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Program Behavior

According to the C Standard, for correct, well-formed programs, almost all behaviors are well defined and predictable. However, in certain cases, an implementation can choose the behavior it deems best, and this behavior need not be the same as that exhibited by other implementations. Such behavior is called *implementation-defined*, and must be documented by each implementation. For example:

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Implementation-Defined Behavior: The range of values that can be stored in an object of a given arithmetic type.

In other cases, an implementation can choose whatever behavior it wants at that time *without* needing to reproduce or document that choice. Such behavior is called *unspecified*. For example:

Unspecified Behavior: Whether two string literals containing the same characters are stored in distinct arrays.

A third category is *undefined behavior*, which can result from situations for which the standard imposes no requirements or is otherwise silent. For example:

Undefined Behavior: Subscripting an array with an out-of-bounds index.

Clearly, we must be aware of implementation-defined and unspecified behaviors when writing code that is to be ported across different platforms. And we should always avoid relying on undefined behavior.

Far too many people believe that just because their C code has worked for a long time, it must be correct. When next your compiler is upgraded, the behavior it exhibits with "undefined behavior" constructs may well be different (as it is permitted to be), resulting in old code breaking. In some cases, simply using a different combination of compiler options with the same compiler can change the way the compiler works internally. You should never try to figure out how your compiler works in "undefined" cases. Any test case you write will be so trivial as to be meaningless. And having 10, 100, or even 1,000 test cases exhibit the same "undefined" behavior is still no guarantee that the next test won't behave differently.

The Status of Standard C

The history of the standardization of C is as follows:

- C89 – The first ANSI C standard, ANSI X3.159-1989, was produced in 1989 by the U.S. committee X3J11.
- C90 – The first ISO C standard, ISO/IEC 9899:1990, was produced in 1990 by committee ISO/IEC JTC 1/SC 22/WG 14 in conjunction with committee X3J11. C90 was technically equivalent to C89.
- C95 – An amendment to C90 was produced in 1995 by committee WG 14 in conjunction with the U.S. committee X3J11. The additions included digraphs, the header `iso646.h`, and many multibyte and wide-character functions via the headers `wchar.h` and `wctype.h`.
- C99 – The second edition of the ISO C standard, ISO/IEC 9899:1999, was produced by committee WG14 in conjunction with the U.S. committee INCITS/J11 (formerly X3J11). The additions included a few language features, a number of headers, and many library functions. Throughout its development, C99 was commonly referred to as C9x.
- C11 – The third edition of the ISO C standard, ISO/IEC 9899:2011, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11 (formerly INCITS/J11). The additions included support for multiple threads of execution, processing Unicode characters and strings, and the querying and specification of the alignment of objects, among other things.

- C17 – The fourth edition of the ISO C standard, ISO/IEC 9899:2017, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11 (formerly INCITS/J11). This was a maintenance release that included corrections to Defect Reports. No new functionality was added.

The C standards committee is currently working on various maintenance issues and informative Technical Reports.

Electronic copies of the latest C standard can be purchased from www.ansi.org or www.iso.ch.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, 2018

10. Miscellaneous Issues

In this chapter, we look at a number of specialized and/or esoteric.

10.1 The Comma Operator

10.1.1 The Basics

One of the most confusing (and consequently least used) operators is the comma. The confusing aspect is that this operator is represented by the comma, yet almost every comma token present in a C program is a comma punctuator, *not* an operator. Despite its unfortunate spelling, the comma operator is a very powerful tool, as we shall see.

Consider the following example:

```
i = 0;
j = 0;
for (k = 0; k > 0; --k)
{
        /* … */
        ++i;
        ++j;
}
```

By definition, the `for` construct contains three optional expressions separated by semicolons. In this example, each expression is simple. We can rewrite this example as follows:

```
for (i = 0, j = 0, k = 0; k > 0; ++i, ++j, --k)
{
        /* … */
}
```

To do more than one thing at initialization and at the end of each iteration of the loop, we can specify a set of expressions separated by comma operators. As shown above, each comma operator expression contains three subexpressions. The comma operator allows us to paste together an arbitrary number of expressions and have them treated syntactically as one large expression.

The comma is a binary operator, and its two operands can be expressions of any data type, including `void`. It has the lowest precedence of all the operators, and it associates left-to-right. A comma expression has the following general form:

exp1 `,` *exp2*

The left operand, *exp1*, is evaluated, and its value is discarded. Therefore, to be useful, *exp1* must contain a side-effect. Then the right operand, *exp2*, is evaluated, and its type and value, if any, become that of the whole comma expression. Consider the following example (see directory mscm01). It is not intended to produce a useful result; it merely demonstrates the syntax. In fact, each statement can be rewritten more clearly, as shown in the corresponding comment:

```
int f();
int g();
int i;
int j = 10;

void test()
{
/*1*/   i = (f(), g());         /* f(); i = g(); */
/*2*/   i = f(), g();           /* i = f(); g(); */
/*3*/   i = (j++, ++j);         /* j += 2; i = j; */
/*4*/   i = 5, j = i, f();      /* i = 5; j = i; f(); */
}
```

In case 1, function `f` is called and the `int` value returned is discarded. Then function `g` is called, and the `int` value it returns becomes the value of the whole comma expression. This value is then assigned to `i`. Note that the grouping parentheses force the comma to take precedence over the assignment.

In case 2, function `f` is called and the value it returns is assigned to `i`, since assignment has higher precedence than comma. The assignment side-effect is completed before the right operand is evaluated. This is important, because `g` might access the global `i`. Then function `g` is called, and the value it returns is used as that of the whole comma expression. In this case, the result actually is discarded, but that is not a property of the comma operator—we simply failed to use it.

In case 3, `j++` is evaluated. Again, the side-effect is completed before the right operand is evaluated. Then the right operand is evaluated, and its value is assigned to `i`.

Since the order of evaluation is guaranteed by each comma in case 4, it can easily be rewritten as three separate statements as shown.

There are situations in which comma operators and punctuators can be used in the same context. For example:

```
/*1*/   f(i, j)
/*2*/   g((++i, j))
```

An argument list is a possibly empty list of expressions separated by comma punctuators. Therefore, function `f` is called with two arguments, `i` and `j`. Function `g`, on the other hand, is called with only one argument, since the expression in the argument list is the comma expression `(++i, j)`. The outer parentheses represent the function call operator, while the inner pair represent grouping.

The comma operator has two important properties. First, it allows an arbitrary number of expressions to be pasted together, yet syntactically they are all considered part of one big expression. Second, the type and value of the right-most expression percolates to the front, becoming the type and value of the result.

It should be obvious that the comma operator can detract from a program's readability and we should avoid using it as much as possible.

this, we need to evaluate the argument once, store that value somewhere, and then use that stored value in the other places. [Hint: How about using a global variable?] (See labs directory lbmsco01.)

Exercise 10-5*: Thus far, we've ignored the possibility of the function being traced returning a value. Enhance the macro `f` from the exercise above to support this as well. (See labs directory lbmsco02.)

Exercise 10-6*: C99 added the ability of declaring a function inline (§4.7). If your compiler supports this, implement a solution to the trace problem that uses an inline function (and which no longer needs the comma operator). (See labs directory lbmsco05.)

Exercise 10-7*: Can this approach to tracing be applying to a function taking a variable number of arguments, such as `printf`? Certainly, a limited subset of the functionality can be achieved by defining `f` to be an object-like macro (§6.2.1). (See labs directory lbmsco03.) However, if your compiler is C99-compliant, you can define an object-like macro to have a variable number of arguments using the `...` punctuator and the special name __VA_ARGS__.. (See labs directory lbmsco04.)

10.2 Pointers to Arrays

C programmers often say, "pointer to an array" when they really mean "pointer to the first element of an array". It is not well known that these pointer types are quite different.

10.2.1 Introduction

Consider the following example (see directory mspa01):

```
#include <stdio.h>

int main()
{
/*1*/   char *pc;               /* pointer to char */
/*2*/   char (*pa)[10];         /* pointer to array of 10 char */

/*3*/   printf("sizeof(*pc) = %lu\n", (unsigned long)sizeof(*pc));
/*4*/   printf("sizeof(*pa) = %lu\n", (unsigned long)sizeof(*pa));

        return 0;
}
```

The output produced is:

```
sizeof(*pc) = 1
sizeof(*pa) = 10
```

In case 1, `pc` is declared as a pointer to a `char`. Therefore, when we dereference that pointer we get a `char`, which, by definition has size 1, as output by case 3. When we perform arithmetic on `pc`, the integer offset is automatically scaled by the size of the object to which the pointer points, in this case, by the size of a `char`.

In case 2, pa is declared to be a pointer to an array of 10 char. (It is most important to note that we have *not* allocated an array of 10 char anywhere, just a pointer capable of pointing at such an array.) Therefore, when we dereference that pointer we get an array of 10 char, which, by definition has size 10, as output by case 4. When we perform arithmetic on pa, the integer offset is automatically scaled by the size of the object to which the pointer points, in this case, by the size of an array of 10 char.

We can get a pointer to an array either by defining a variable to have that type or by taking the address of an array. For example (see directory mspa02):

```
int i[6];

/*1*/   int (*p1)[6] = &i;              /* correct   */
/*2*/   int (*p2)[6] = i;               /* incorrect */
/*3*/   int (*p3)[6] = (int (*)[6])i;   /* OK        */
```

In case 1, p1 is defined to be a pointer to an array of 6 int. By taking the address of i, we have an expression of the same type, so the initializer is accepted. This is not so in case 2, since the expression i is equivalent to &i[0], which has type pointer to int. Since these types are not assignment compatible, the initializer is rejected.

We can make p3 point to the address of i[0] by explicitly casting that address to the desired pointer type, as shown in case 3. In reality, p3 and &i[0] lead us to the same place, however, as we have seen, we get different results when we dereference these addresses or perform arithmetic on them.

Under certain circumstances, pointers to arrays are created automatically by the compiler. For example (see directory mspa03):

```
void g(double [][10]);

void f()
{
        double d[5][10];

/*1*/   g(d);
/*2*/   g(&d[0][0]);    /* error */
/*3*/   g(&d[0]);       /* OK    */
}
```

In case 1, we pass the 2-D array d to the function g. And as we known, arrays are passed by address. We also know that the name of an array is converted to the address of its first element. If they had to write out the argument to g explicitly, the vast majority of C programmers likely would write it like case 2; afterall, isn't the first element of d, d[0][0]? However, that would be incorrect. Why is this? Simply stated, **every array in C has one dimension!** Say what? How can that be? Can't we have multidimensional arrays in C?

The fact that every array in C has one dimension doesn't stop each of the elements in that dimension being arrays in their own right. The reason we write multiple dimensions separately is that [] is an operator that can be applied iteratively, and it associates left-to-right. So, the definition of the array double d[5][10] is read as "d is an array of 5 elements each of which is an array of 10 elements each of which is an object of type double". Therefore, if d has 5 elements, the address of the first element must be &d[0], as used in case 3 above.

A similar situation occurs when one of the arrays is selected and its address is returned to the caller, for later indirect access. For example (see directory mspa08):

```
double (*get_table())[3]
{
        static double a[5][3], b[5][3];

        int condition;

        /* … */

        if (condition)
                return a;       /* same as return &a[0] */
        else
                return b;       /* same as return &b[0] */
}
```

The function's return type looks interesting. However, it is no more complicated than declaring a pointer to an array. This function might be called as follows:

```
void test()
{
        double (*pd)[3];
        double d;

/*1*/   pd = get_table();
/*2*/   d = pd[1][1];

/*3*/   d = get_table()[1][1];
}
```

The variable pd really isn't needed; cases 1 and 2 can be combined as case 3, which gives rise to a most interesting expression, a doubly subscripted function call!

Pointers to arrays can require the use of some unusual notation. They are rarely used, probably because most programmers aren't aware they exist. However, they can serve a useful purpose once we know about them.

10.3 Mastering Declarations

One of the biggest strengths of C is its rich set of types. However, once we get past the basic types and elementary derived types, it no longer is obvious by quick inspection just what a given declaration means. In this section, we will learn the limits of the typing system, and how to read and write declarations of arbitrary complexity reliably.

10.3.1 Introduction

One of the most complicated function prototypes in the standard library is that for the function `signal` (declared in the header `signal.h`):

```
void (*signal(int signal_type, void (*action)(int)))(int);
```

How many arguments does this function take, what are their types, and what is the return type? By the end of this section, we should be able to say, with confidence. While we don't often need to have such complex function declarations, they are possible, and they can be useful. Certainly, we need to understand this function if we are to register signal handlers.

To exploit C fully, we must be able to read and write arbitrarily complex declarations with confidence. However, almost all of the declarations we will read and write in our C careers will be quite simple and we will just "know" how to read and write them. In these simple cases, we don't need to follow any rules, and sadly, most C veterans don't seem to even know what the rules are.

10.3.2 Basic Types

A *basic type* in C is one involving C language type keywords only. The complete set of basic types is: the signed and unsigned versions of the integral types `char`, `short`, `int`, `long`, and `long long`; the floating-point types `float`, `double` and `long double`; classes, structures, and unions; enumerated types; and `void`.

Declaring an object of one of these basic types is simple—we write the type and follow it with an identifier. (For the purposes of this discussion, we will ignore the fact that a declaration may include multiple declarators, since this has no impact on our discussion. We will also ignore the trailing semicolon, since this is not part of the declarator.) Some examples of declarations involving basic types are:

```
int i;
double d;
struct tag s;
enum color my_color;
```

10.3.3 Deriving from a Basic Type

There are three possible ways to derive a type *T1* from a basic type *T*. They are:

1. *T1* is an array of objects each having type *T*.
2. *T1* is a function having return type *T*.
3. *T1* is a pointer to type *T*.

An example containing one of each is:

```
int i;          /* basic type,    int                    */
int a[10];      /* derived type, array of 10 int         */
int f(void);    /* derived type, function returning int */
int *pi;        /* derived type, pointer to int          */
```

The simple rule to remember here involves the position of the derived type punctuation in the declarator. In the case of arrays and functions, the `[]` and `()` are postfix punctuators; that is, they follow immediately after the identifier to which they apply. The pointer notation `*` is a prefix punctuator, so it comes immediately before the identifier to which it applies.

10.3.4 Deriving from a Derived Type

Since a derived type is derived from another type, it follows that we can derive a type from another derived type, which in turn was derived from another type, *ad infinitum*. Some simple examples follow:

Operator	Example	Lvalue	Modifiable Lvalue
Others	—	never	never

† *pc designates an object provided pc is not a pointer to a function or a pointer to an array of unknown size.

Since only objects can be contained in an array, `[]` designates one of those objects. Similarly, when a pointer to an object is dereferenced, we are referring to the underlying object. And, since all members of structures or unions must be objects, `->` always designates an object. The dot operator usually generates an lvalue. The only case where it does not is in the vase of `f().m`. This construct is permitted since a function can return a structure or union by value. However, since the the function call operator `()` does not produce an lvalue, any subordinate member is also not an lvalue.

Any lvalue is also a modifiable lvalue provided it does not designate an array or a `const`-qualified object.

10.5.4 Operators That Need Modifiable Lvalues

There are only three operators needing modifiable lvalues: `++`, `--`, and the assignment operators (left-hand operand only).

Let's revisit the original example in §10.5.1 and look at each error. In line 7, `c` is an lvalue, but not a modifiable lvalue; therefore, it cannot be modified. In line 9, the `1` should have been `i`, not an uncommon mistake. Since the function call operator does not produce an lvalue, line 12 is rejected.

The problem in line 14 has to do with how tokens are recognized by the compiler. What the compiler is really seeing is not `i++ + ++j`, but rather, `i++++ + j`. The error arises since the operand to the second ++ is not an lvalue.

10.6 Variable-Length Arrays

Prior to C99, arrays had to have dimensions whose size was known at compile time. C99 added the ability to have dimension sizes be integer expressions whose value is not know until runtime, but only for arrays having automatic storage duration (§5.2), which includes automatic variables and parameters. Note, however, that this is a conditional feature; that is, a C99-comformant compiler need not actually support this. Here's how to tell if that feature is available in a C99 implementation:

```
#if __STDC_NO_VLA__ == 1
    /* variable-length arrays are not supported */
#endif
```

Consider the following (see directory msvl01):

```
f1(2, 6);

void f1(int m, int n)
{
/*1*/    double x1[4];           // fixed-length of 4 elements
/*2*/    double x2[m * 2];       // 2x2=4 elements
/*3*/    double x3[2][n];        // 2 elements, each of which has 6 elements
/*3*/    double x4[m + 2][5];    // 2+2=4 elements, each of which has 5 elements
/*4*/    double x5[n][m];        // 6 elements, each of which has 2 elements
```

```
          int i = 3;
/*5*/     double x6[i];           // 3 elements           …
}
```

Consider the following:

```
void f2(int n, int values[n][n*2]);     // OK
void f3(int values[n][n*2], int n);     // error; no n is in scope
```

In the case of `f2`, `n` is seen first, so it can be used in the dimensions for any array parameter following. However, that is not the case for the declaration of `f3`. Then, in the definition of `f2`:

```
void f2(int n, int values[n][n*2]) { … }
```

being a parameter, `values` has automatic storage duration.

Note that we can write the non-definition declaration of `f2`, as follows instead:

```
void f2(int n, int values[*][*]);
```

Pointers to variable-length arrays are also permitted; for example:

```
int m = 3, n = 5;
int a[m][n];
int (*p1)[n] = a;   // p1 == &a[0]
```

A type synonym can be created for a variable-length array; for example:

```
int k = 3;
typedef int A[k];     // A is k ints, with k evaluated now, as 3
++k;                  // k = 4
A t1;                 // t1 is 3 ints, not 4!
```

A variable-length array definition cannot have an initializer list, and it cannot be a member of a structure or union. And if the runtime size of any dimension is less than 1, the behavior is unspecified. Once the size of a variable-length array has been determined, that size of that array is fixed throughout its life.

10.7 Internationalization

Considerable thought went into making Standard C extensible for use in non-USA and non-English environments. The facilities available to aid with this are described below.

10.7.1 Introduction

An *internationalized* program is one that has no dependency on any culture. (The term *internationalization* is often referred to as I18N.)

Localization (L10N) is the process of adapting an internationalized program to a specific cultural environment.

The two main reasons for internationalizing software are:

1. To support a multi-cultural organization (which need not necessarily be multinational).
2. To allow the development of software for export to countries having a different culture.

Many of us live almost exclusively in a one-culture world. However, once we start looking for differences in cultures, we find an astounding array. Here are the most common ones:

1. Alphabet and collating sequence
2. What constitutes a letter, a number, a punctuation mark, and white space (if any)?
3. Is the concept of upper- and lowercase letters supported? If so, do all uppercase letters have a lowercase counterpart and vice versa?
4. Display width and height of characters
5. Length of translated text and size of display area
6. What hyphenation rules (if any) exist?
7. Date and time formats, month names and abbreviations, time zones, Gregorian vs. Era calendars
8. How are fractional numbers formatted? How is rounding handled? Negative numbers?
9. How are currency numbers formatted? What are the national and international currency symbols? How is rounding handled? Negative numbers?
10. Length of first and/or last names. Does the concept of first and last names and middle initial even exist?
11. Writing direction: left-to-right/right-to-left/top-to-bottom. A carriage return doesn't always go down one line and to the left margin! Can multiple writing directions be used together?
12. Address formats
13. Titles and forms of address
14. Telephone number formats
15. Spelling of country and place names (i.e., Germany vs. Deutschland, Munich vs. München, and London vs. Londres)
16. Measurement system (metric vs. imperial vs. US)

Other requirements brought on by a change in culture are:

1. On-line help screens and printed documentation need to be translated
2. On-screen forms and prompt and error messages need to be translated
3. Voice messages need to be translated. Music and other audio sounds might have to be changed.
4. Hot keys (Alt-F for “File” doesn’t work if the translated word for File doesn’t contain an F!)

The three most important things to remember when thinking about writing internationalized code are:

ABSTRACTION
ABSTRACTION
ABSTRACTION

Abstraction involves the separation of a logical interface from a physical implementation or representation. Conside the following:

1. Successful abstraction requires isolation of implementation-defined aspects.
2. It is important to consider at what level you want to achieve abstraction: compile time, link time, run time. All three approaches have their advantages and disadvantages; there is no one correct answer for every application.
3. A project requiring \internat\ is one that involves some form of portability. And since abstraction is the key to portability, and \internat\ is just another (albeit major) variable in the portability equation, abstraction is the key to achieving \internat.
4. It's okay to limit the number of target cultures you support.

5. You can't achieve something or measure your progress if you haven't defined it.
6. If you find you need something outside the framework of your original design, you either have to consider that a design limitation or set about changing the design goals.

Don't Reinvent the Wheel! See what else it out there, as

1. Defined by national, international, or de-facto industry standards
2. Proprietary to a particular vendor

10.7.2 Locales

Historically, the C library assumed a USA/English environment with regard to the testing of characters, formatting of floating-point radix point, date and time formatting, and so on. However, one presumes that Germans would like `tolower('Ö')` to produce `'ö'` while Spaniards would like `isalpha('ñ')` to test true. Many Europeans want a comma as a decimal point rather than a period. Some cultures use 24-hour clocks, and many spell the months of the year using non-English letters.

A *locale* is a set of conventions based on some nationality, culture, or language. A locale is made up of a set of *categories*. A number of standard library functions are *locale-dependent*, allowing them to correctly handle non-Roman letters.

Two locales are defined by Standard C: `"C"` and the native locale `""`. (An implementation that provides the bare minimum locale support will provide locale `"C"` and it will make the native locale `""` be the same as locale `"C"`.) Any number of other locales are permitted.

Implementation-Defined Behavior: The native locale of an implementation.

At program startup, the default locale is `"C"`. Changing one or more categories of a locale to something other than `"C"` requires a call to the function `setlocale`, declared in `<locale.h>`.

Consider the following program (see directory msin01) that performs a number of locale-specific operations based on a user-specified locale:

```
#include <locale.h>
#include <stdio.h>
#include <time.h>

int main()
{
        char locale[31];
        char *ploc_str;
        time_t system_time;
        char time_text[81];

        printf("Enter locale name: ");
/*1*/   scanf("%30s%*c", locale);
```

10.7.5 Avoiding Hard-Coding of Characters and Strings

Starting with the very first program most of us ever wrote, we've been hard-coding the content of characters and strings. However, that is something we absolutely don't want to do when designing code to be deployed across multiple locales. Instead, we need some sort of character/string repository with each entry accessed by a unique key. The source code needing to dispay locale-specific text then must use such keys at runtime to retrieve from the repository the corresponding character/string. For example, while we might start out using the following:

```
printf("The value entered was %s.", value);
```

where value is 24, we'd need to convert this to something like the following:

```
printf(getFormat(formatKey), value);
```

where the text retrieved by `getFormat` for English is

```
"The value entered was %s."
```

Then we can provide a different repository for each locale to be supported, such that the formatKey corresponds to the translated message for each locale, allowing of course for changes in word order. For example (using the results from Google Translate),

```
"Der eingegebene Wert war %s."   (German)
"Sartutako balioa %s izan da."   (Basque)
"A beírt érték %s volt."         (Hungarian)
"Girilen değer %s 'tü."          (Turkish)
```

10.7.6 Universal Character Names

C99 added support for *universal character names*, which allow characters other than those on a programmer's keyboard to be specified in source code. These names have one of the forms \u*xxxx* and \U*xxxxxxxx*, where *x* is a hexadecimal digit, whose value corresponds to a Unicode code point.

Universal character names are recognized as such in identifiers and escape seqeunces (in both character constants and string literals. For example, \u00E4 and \U000000E4 represent the German letter ä, and \u00A9 and \U000000A9 represent the copyright symbol ©.

10.8 Variable-Length Argument Lists

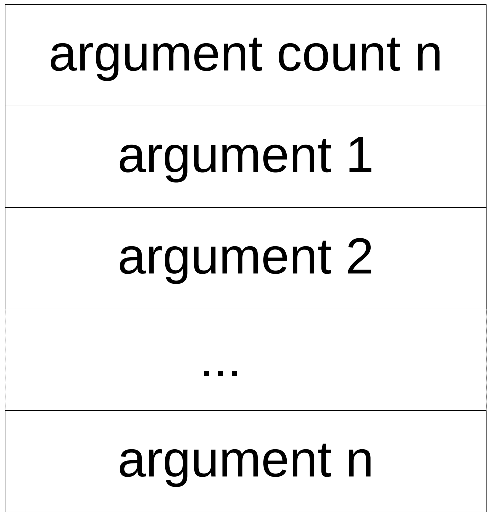
In the C library, only a small number of functions expect a variable number of arguments. They are the `printf` and `scanf` families. Since a prototype cannot be written to check calls to such functions at compile time, it is generally bad style to define functions in such a manner. However, Standard C does provide a portable facility that allows variable argument lists to be processed. In this section, we will look at the machinery provided for this purpose.

10.8.1 Introduction

When a function is called, arguments are passed to it in some unspecified fashion. Provided the called function gets access to the arguments, the implementation details are the business of the implementer. About the only time an application programmer need be aware of argument-passing details is if they are calling one language from another or are trying to take advantage of special function-call instructions provided by their hardware.[1]

If a function has a fixed-length argument list, certain optimizations are possible. For example, some arguments might be passed in registers while others are passed on some stack. Perhaps only objects of certain types can be passed in registers. Some implementations even use multiple registers to pass a single argument. If all calls to a function having a fixed-length argument list are made in the presence of the same prototype and that function is also defined in the scope of that prototype, Standard C guarantees the call will succeed as planned. Since the compiler sees both the definition and all calls, it can perform the same optimizations on both sides.

In the case of functions having variable-length argument lists, the compiler cannot tell from the function's definition the number or type of arguments expected beyond the fixed argument-list part. Therefore, it cannot necessarily take advantage of passing certain type arguments via one method versus another. In fact, it can't even tell much from looking at calls to such functions—each might have a different number and/or type of arguments. Therefore, the list of arguments passed to such a function needs to be handled in some manner that is constant for a given implementation. The following model is commonly used to pass arguments to functions:



The first argument is a count of the arguments that follow. In such an approach, the type of this argument determines the maximum number of arguments a function can have. For example, an `unsigned char`, where `char`s have eight bits, limits functions to 255 arguments, which seems like a reasonable number. In this model, the size of each argument is usually required to be the same. This gives rise to the idea of wide and narrow types. For example, why have arguments of types `char`, `short`, and `float` historically been widened to `int`, `int`, and `double`, respectively? Well on some systems, this is necessary to keep objects of certain types aligned.

This calling-argument format works well when all arguments are passed by address, or small-size arguments are passed by value. However, it breaks down when large structures are passed by value. It can also be a problem when a `long int`, a `long long int`, a `double`, or a `long double` is passed by value since the size of these may exceed the size of an argument in this model. As a result, the argument count is no longer a count of logical arguments but, rather, of entries in the argument list. For example, if each argument in the list is four bytes and we pass a 100-byte structure by value, the argument count would be 25, not 1. So, the upper limit of 255 could be reached by passing just one large structure by value.

[1] Some implementations provide `#pragma` directives to achieve this.

The model for argument passing usually employed by a C implementation is close to that shown above. However, most often, an argument count is not automatically supplied; if the user wants one they must provide it themselves. The implication of this is that calls such as the following cannot be dealt with in a portable manner:

```
maximum(10, 5, 6, 7)
maximum(2, 65, 876)
maximum(7654, 234, 2374, 3421, 6487)
```

Without a preceding argument count, how can `maximum` determine the number of arguments passed to it? We could reserve some special-valued terminator but that really isn't workable. For example, all `int` values are valid—how would we terminate a variable-length list of `int`s in a call to `maximum`? For pointer argument lists, a null pointer would certainly work, however; but what about lists of arguments having mixed types?

Languages that support calls similar to those of `maximum` typically do so because they have intrinsic functions built-in to the language; they are not really calls to externally compiled routines. As such, the compiler can generate special code behind the scenes to deal with such lists.

Essentially, the C model requires that the programmer pass the argument count as part of the list, either explicitly as a number, or implicitly by something like the conversion specifiers used by `printf` and `scanf`. For example:

```
/*1*/   maximum(4, 10, 5, 6, 7)

/*2*/   printf("%d %f", i, d)
```

In case 1, the first argument, 4, indicates that four more arguments follow. By definition, `maximum` expects all arguments to have the same type; in this case, `int`. In case 2, there are two conversion specifiers indicating that two more arguments follow. Their types are encoded in the specifiers allowing arguments of mixed types to be used.

Undefined Behavior: In a call to a function having a variable number of arguments, if the number and/or type of arguments following the explicit or implicit argument count do not match what was promised.

Therefore, passing a variable number of arguments requires the programmer to check all such calls for correctness since the compiler cannot.

10.8.2 Implementing a Maximum Function

The following example shows how to use and define a function `maxi`, which returns the maximum of a set of `int` arguments (excluding the count) passed to it (see directory msva01):

10.12.1 Sorting

10.12.1.1 Introduction

The library function `qsort` uses the quicksort method to sort the elements of an array into ascending order. It is declared as follows:

```
void qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

where `base` is the address of the first element of the array to be sorted, `nmemb` is the number of elements to be sorted starting at address `base`, `size` is the size of each element in the array, and `compar` is the address of a user-supplied function to be used in element comparisons. The comparison function is expected to return a negative value if the first argument tests less than the second, a zero if they are equal, and a positive value if the first argument tests greater than the first, just like the library function `strcmp`.

Since a `void` pointer is assignment compatible with any data pointer, the first argument to `qsort` can be the address of any data type. When `qsort` is actually called, the pointer passed will be converted, as if by assignment, to a `void` pointer.

10.12.1.2 Sorting a List of Integers

Let's begin by sorting an array of six `int`s into ascending order (see directory msss01a):

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

#define NUMELEM(a) (sizeof(a)/sizeof(a[0]))

int main()
{
        int cmpintsa(const void *, const void *);
        int array[] = {25, 3, 22, -5, 3, 24};
        int i;

        qsort(&array[0], NUMELEM(array), sizeof(int), cmpintsa);

        printf("ascending integer order\n");
        for (i = 0; i < NUMELEM(array); ++i)
        {
                printf("array[%d] = %2d\n", i, array[i]);
        }

        return 0;
}
```

```
/* compare ints in ascending order */

int cmpintsa(const void *pe1, const void *pe2)
{
        const int *pi1 = pe1;
        const int *pi2 = pe2;

        if (*pi1 < *pi2)
        {
                return -1;
        }
        else if (*pi1 == *pi2)
        {
                return 0;
        }
        else
        {
                return 1;
        }
}
```

The output produced is:

```
ascending integer order
array[0] = -5
array[1] =  3
array[2] =  3
array[3] = 22
array[4] = 24
array[5] = 25
```

Calling `qsort` is quite straightforward as is defining the comparison function `cmpintsa`. `cmpintsa` needs to compare two `int`s whose addresses it is passed. However, `cmpintsa` is required to take two `void` pointers *not* two `int` pointers. Clearly, we cannot dereference a `void` pointer, so a private copy of each argument is made in an `int` pointer. Note the need for the `const` qualifier on the `int` pointers. Once we have two `int` pointers, it is a simple matter to compare the underlying `int`s and returning the appropriate indicator, as shown.

There are two entries in the array with the value 3.

Unspecified Behavior: The ordering of duplicate entries in an array sorted by `qsort`.

The variables `pi1` and `pi2` really are unnecessary. They can be replaced by a cast, as follows (see directory msss01b), although the resulting code is less clear: