1.3 A First Program

> *A C program, whatever its size, consists of* functions *and* variables. *A function contains* statements *that specify the computing operations to be done, and variables store values used during the computation [KR88, page 6].*

The following program is the traditional first program presented in introductory C courses and textbooks.

```
1 /* First C program: Hello World */
2 #include <stdio.h>
3
4 int main(void)
5 {
6         printf("Hello World!\n");
7 }
```

1 Comments in C start with /* and are terminated with */. They can span multiple lines and are not nestable. For example,

```
/* this attempt to nest two comments /* results in just one comment,
ending here: */ and the remaining text is a syntax error. */
```


2 Inclusion of a standard library header-file. Most of C's functionality comes from libraries. Header-files contain the information necessary to use these libraries, such as function declarations and macros.

4 All C programs have `main()` as the entry point function. This function comes in two forms:

```
int main(void)
int main(int argc, char *argv[])
```

The first takes no arguments, and the second receives command-line arguments from the environment in which the program was executed—typically a command-shell. (More on command-line arguments in Section 13.4.) The function returns a value of type `int` (i.e., an *integer*).[2]

5 and 7 The braces { and } delineate the extent of the function block. When a function completes, the program returns to the calling function. In the case of `main()`, the program terminates and control returns to the environment in which the program was executed. The integer return value of `main()` indicates the program's exit status to the environment, with 0 meaning normal termination.

6 This program contains just one statement: a function call to the standard library function `printf()`, which prints a *character string* to standard output (usually the screen). Note, `printf()` is not a part of the C language, but a function provided by the standard library (declared in header `stdio.h`). The standard library is a set of functions mandated to exist on all systems conforming to the ISO C standard. In this case, the `printf()` function takes one *argument* (or input parameter): the *string constant* `"Hello World!\n"`. The `\n` at the end of the string is an *escape character* to start a new line. Escape characters provide a mechanism for representing hard-to-type or invisible characters (e.g., `\t` for tab, `\b` for backspace, `\"` for double quotes). Finally, the statement is terminated with a semicolon (`;`). C is a free-form language, with program meaning unaffected by whitespace in most circumstances. Thus, statements are terminated by ; not by a new line.

[2] You may notice in the example program above, that `main()` *says* it returns `int` in its interface declaration, but in fact does *not* return anything; the function body (lines 5–7) contains no `return` statement. The reason is that for `main()`, and `main()` only, an explicit `return` statement is optional (see Chapter 4 for more details).

13 Unlike the previous versions of this program, this one includes an explicit `return` statement for the program's exit status.

Style note. Throughout this text take notice of the formatting style used in the example code, particularly indentation. Indentation is a critical component in writing clear C programs. The compiler does not care about indentation, but it makes the program easier to read for programmers.

1.5 A Numerical Example

```
1  /* Fahrenheit to Celcius conversion table (K&R page 12) */
2  #include <stdio.h>
3
4  int main(void)
5  {
6          float fahr, celsius;
7          int lower, upper, step;
8
9          /* Set lower and upper limits of the temperature table (in Fahrenheit) along with the
10          * table increment step-size */
11         lower = 0;
12         upper = 300;
13         step = 20;
14
15         /* Create conversion table using the equation: C = (5/9)(F - 32) */
16         fahr = lower;
17         while (fahr <= upper) {
18                 celsius = (5.0/9.0) * (fahr-32.0);
19                 printf("%3.0f \t%6.1f\n", fahr, celsius);
20                 fahr += step;
21         }
22 }
```


6–7 This program uses several *variables*. These must be declared at the top of a block, before any statements. Variables are specified *types*, which are `int` and `float` in this example.

9–10 Note, the `*` beginning line 10 is not required and is there for purely aesthetic reasons.

11–13 These first three statements in the program initialise the three integer variables.

16 The floating-point variable `fahr` is initialised. Notice that the two variables are of different type (int and float). The compiler performs automatic *type conversion* for compatible types.

17–21 The `while`-loop executes while ever the expression `(fahr <= upper)` is TRUE. The operator `<=` means LESS THAN OR EQUAL TO. This loop executes a *compound statement* enclosed in braces—these are the three statements on lines 18–20.

18 This statement performs the actual numerical computations for the conversion and stores the result in the variable `celcius`.

19 The `printf()` statement here consists of a format string and two variables `fahr` and `celcius`. The format string has two *conversion specifiers*, `%3.0f` and `%6.1f`, and two escape characters, tab and new-line. (The conversion specifier `%6.1f`, for example, formats a floating-point number allowing space for at least six digits and printing one digit after the decimal point. See Section 13.1.1 for more information on `printf()` and conversion specifiers.)

20 The assignment operator `+=` produces an expression equivalent to `fahr = fahr + step`.

is the design of functions that can operate on a variety of different data types. Chapter 15 presents a selection of the fundamental data-structures that appear in many real programs and are both instructive and useful.

Chapter 16 provides a context for the book by describing how the ISO C language fits into the wider world of programming. Real world programming involves a great number of extensions beyond the standard language and C programmers must deal with other libraries, and possibly other languages, when writing real applications. Chapter 16 gives a taste of some of the issues.

prints

```
1234    2322    4d2
1234    1234    1234
```

Notice that C does not provide a direct binary representation. However, the hex form is very useful in practice as it breaks down binary into blocks of four bits (see Section 12.1).

Floating-point constants are specified by a decimal point after a number. For example, 1. and 1.3 are of type `double`, 3.14f and 2.f are of type `float`, and 7.L is of type `long double`. Floating-point numbers can also be written using scientific notation, such as 1.65e-2 (which is equivalent to 0.0165). Constant expressions, such as `3+7+9.2`, are evaluated at compile-time and replaced by a single constant value, 19.2. Thus, constant expressions incur no runtime overhead.

Character constants, such as `'a'`, `'\n'`, `'7'`, are specified by single quotes. Character constants are noteworthy because they are, in fact, not of type `char`, but of `int`. Thus, `sizeof('Z')` will equal 4 on a 32-bit machine, not one. Most platforms represent characters using the ASCII character set, which associates the integers 0 to 127 with specific characters (e.g., the character `'T'` is represented by the integer 84). Tables of the ASCII character set are readily found (see, for example, [HS95, page 421]).

There are certain characters that cannot be represented directly, but rather are denoted by an "escape sequence". It is important to recognise that these *escape characters* still represent single characters. A selection of key escape characters are the following: `\0` for NUL (used to terminate character strings), `\n` for newline, `\t` for tab, `\v` for vertical tab, `\\` for backslash, `\'` for single quotes, `\"` for double quotes, and `\b` for backspace.

String constants, such as `"This is a string"` are delimited by quotes (note, the quotes are not actually part of the string constant). They are implicitly appended with a terminating `'\0'` character. Thus, in memory, the above string constant would comprise the following character sequence: `This is a string\0`.

Note. It is important to differentiate between a character constant (e.g., `'X'`) and a `NUL` terminated string constant (e.g., `"X"`). The latter is the concatenation of two characters `X\0`. Note also that `sizeof('X')` is four (on a 32-bit machine) while `sizeof("X")` is two.

2.4 Symbolic Constants

Symbolic constants represent constant values, from the set of constant types mentioned above, by a symbolic name. For example,

```
#define BLOCK_SIZE      100
#define TRACK_SIZE      (16*BLOCK_SIZE)
#define HELLO           "Hello World\n"
#define EXP             2.7183
```

Wherever a symbolic constant appears in the code, it is equivalent to direct text-replacement with the constant it defines. For example,

```
printf(HELLO);
```

prints the string `Hello World`. The reason for using symbolic constants rather than constant values directly, is that it prevents the proliferation of "magic numbers"—numerical constants scattered throughout the code.[3] This is very important as magic numbers are error-prone and are the source of major difficulty when attempting to make code-changes. Symbolic constants keep constants together in one place so that making changes is easy and safe.

[3] For example, refer to the Fahrenheit to Celcius examples from Sections 1.5 and 1.6. The first example uses magic numbers, while the second uses symbolic constants.

completeness, we mention also the bitwise assignment operators: `&=`, `|=`, `^=`, `<<=`, and `>>=`. We return to the bitwise operators in Chapter 12.

2.11 Type Conversions and Casts

> *When an operator has operands of different types, they are converted to a common type according to a small number of rules [KR88, page 42].*

For a binary expression such as `a * b`, the following rules are followed (assuming neither operand is unsigned):

- If either operand is `long double`, convert the other to `long double`.
- Otherwise, if either operand is `double`, convert the other to `double`.
- Otherwise, if either operand is `float`, convert the other to `float`.
- Otherwise, convert `char` and `short` to `int`, and, if either operand is `long`, convert the other to `long`.

If the two operands consist of a `signed` and an `unsigned` version of the same type, then the `signed` operand will be promoted to `unsigned`, with strange results if the previously `signed` value was negative.

A simple example of type promotion is shown in the following code.

```
short a = 5;
int b = 10;
float c = 23.1f;
double d = c + a*b;
```

Here the multiply is performed first, so `a` is promoted to `int` and multiplied with `b`. The integer result of this expression is promoted to `float` and added to `c`. This result is then promoted to `double` and assigned to `d`.

Note. The promotion from `char` to `int` is implementation-dependent, since whether a plain `char` is signed or unsigned depends on the compiler. Some platforms will perform “sign extension” if the left-most bit is 1, while others will fill the high-order bits with zeros—so the value is always positive.

Assignment to a “narrower” operand is possible, although information may be lost. Conversion to a narrower type should elicit a warning from good compilers. Conversion from a larger integer to a smaller one results in truncation of the higher-order bits, and conversion from floating-point to integer causes truncation of any fractional part. For example,

```
int iresult = 0.5 + 3/5.0;
```

The division 3/5.0 is promoted to type double so that the final summation equals 1.1. The result then is truncated to 1 in the assignment to `iresult`. Note, a conversion from `double` to `float` is implementation dependent and might be either truncated or rounded.

Narrowing conversions should be avoided. For the cases where they are necessary, they should be made explicit by a *cast*. For example,

```
int iresult = (int)(0.5 + 3/5.0);
```

Casts can also be used to coerce a conversion, such as going against the promotion rules specified above. For example, the expression

```
result = (float)5.0 + 3.f;
```

will add the two terms as `float`’s rather than `double`’s.

Chapter 3

Branching and Iteration

The C language provides three types of decision-making constructs: `if-else`, the *conditional expression* `?:`, and the `switch` statement. It also provides three looping constructs: `while`, `do-while`, and `for`. And it has the infamous `goto`, which is capable of both non-conditional branching and looping.

3.1 If-Else

The basic `if` statement tests a conditional expression and, if it is non-zero (i.e., TRUE), executes the subsequent statement. For example, in this code segment

```
if (a < b)
    b = a;
```

the assignment `b = a` will only occur if a is less-than b. The `else` statement deals with the alternative case where the conditional expression is 0 (i.e., FALSE).

```
if (a < b)
    b = a;
else
    b += 7;
```

The `if-else` statement can also command multiple statements by wrapping them in braces. Statements so grouped are called a *compound statement*, or *block*, and they are syntactically equivalent to a single statement.

```
if (a < b) {
    b = a;
    a *= 2;
}
else {
    b += 7;
    --a;
}
```

It is possible to chain `if-else` statements if the following form

```
if (expression)
    statement;
else if (expression)
```

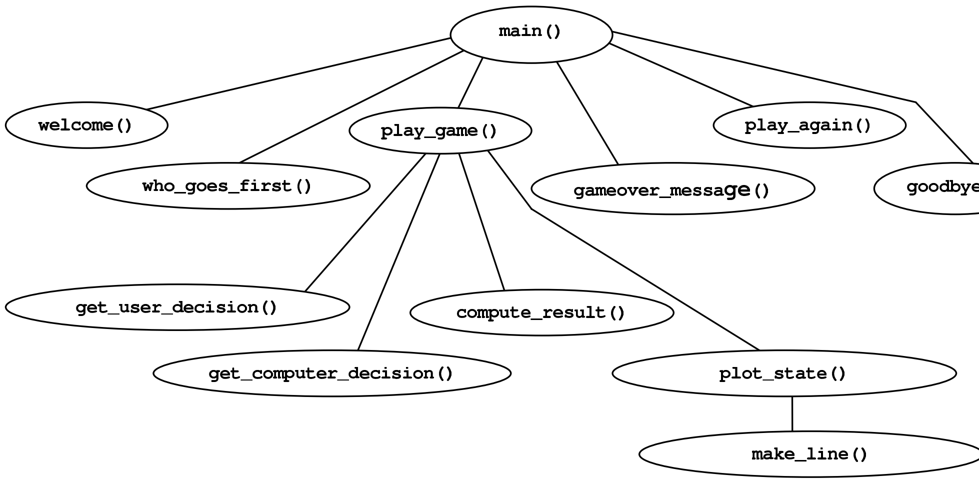

Figure 6.3: The top-down design of the Tic-Tac-Toe program.

6.5.6 Benefits of Modular Design

This design encloses each operation within a function. The function interfaces are minimal and decoupled, and completely hide any implementation details. The benefit of this modularity is that the code is flexible; changes and extensions are simple to implement.

Consider the following examples. One, it is possible to change the welcome and goodbye messages easily. Two, all input handling is encapsulated within the function `getint_from_user()`, which incorporates appropriate error checking. And three, if the game is ported to an environment with a graphical interface, only the input-output functions need to be revised.

By far the most technical part of this program was the computer decision-making function `get_computer_decision()`. To get the computer to make good choices is not trivial. However, to make the program run does not require an intelligent opponent, and a very simple random selection scheme was sufficient. Once the rest of the program was fully tested, it was straightforward to write cleverer decision-making code. This is a good example of hiding an algorithm behind an interface, allowing various implementations to be tested and compared without change to the rest of the program.

```
char c = 'A';
char *pc = &c; /* pc points to c */
double d = 5.34;
double *pd1, *pd2;

*pc = 'B'; /* Dereferenced pointer: c is now equal to 'B'. */
pd1 = &d;  /* pd1 points to d */
pd2 = pd1; /* pd2 and pd1 now both point to d. */
*pd1 = *pd2 * 2.0; /* Equivalent to d = d * 2.0; */
```

Notice that pointers have different types specifying the type of object to which they can point. It is an error to assign a pointer to an object of a different type without an explicit cast.[1]

```
float i = 2.f;
unsigned long *p1 = &i; /* Error: type mismatch, won't compile. */
unsigned long *p2 = (unsigned long *)&i; /* OK, but strange. */
```

The exception to this rule is the `void*` pointer which may be assigned to a pointer of any type without a cast.

It is dangerous practice to leave a pointer uninitialised, pointing to an arbitrary address. If a pointer is supposed to point nowhere, it should do so explicitly via the `NULL` pointer. `NULL` is a symbolic constant defined in the standard headers `stdio.h` and `stddef.h`. It is usually defined as

```
#define NULL    ((void*) 0)
```

The constant values 0 or 0L may be used in place of `NULL` to specify a null-pointer value, but the symbolic constant is usually the more readable option.

Pointers may be declared `const`; and this may be done in one of two ways. The first, and most common, is to declare the pointer `const` so that the object to which it points cannot be changed.

```
int i = 5, j = 6;
const int *p = &i;
*p = j; /* Invalid. Cannot change i via p. */
```

However, the pointer itself may be changed to point to another object.

```
int i = 5, j = 6;
const int *p = &i;
p = &j; /* Valid. p now points to j. */
*p = i; /* Invalid. Cannot change j via p. */
```

The second form of `const` declaration specifies a pointer that may only refer to one fixed address. That is, the pointer value may not change, but the value of the object to which it points may change.

```
int i = 5, j = 6;
int * const p = &i;
*p = j; /* Valid. i is now 6 */
p = &j; /* Invalid. p must always point to i. */
```

It is possible to combine these two forms to define a non-changing pointer to a non-changeable object.

```
int i = 5, j = 6;
const int * const p = &i;
*p = j; /* Invalid. i cannot be changed via p. */
p = &j; /* Invalid. p must always point to i. */
```

[1]Casting a pointer from one type to another (say `int *` to `char *`) is an operation that should only be performed if you know what you are doing. It is sometimes useful in very low-level code, and is usually non-portable. On the other hand, converting from `void*` to another pointer type (and vice-versa) is a common and useful operation; when doing so, it is not necessary to use an explicit cast, but a cast may be useful for expressing intent.

Aside. In general, the concatenation of two strings requires the use of a function like `strcat()`. However, string *constants* may be concatenated at compile time by placing them adjacent to one another. For example, `"this is " "a string"` is equivalent to `"this is a string"`. Compile-time concatenation is useful for writing long strings, since typing a multi-line string constant like

```
"this is
    a string"
```

is an error. An alternative way to write multi-line string constants is to write

```
"this is \
a string"
```

where the first character of the second half of the string occurs in the first column of the next line without preceding white-space. (This is one occasion where white-space matters in a C program.) Usually the adjacency method is preferred over the '\' method.

8.4 Arrays of Pointers

Since pointers are themselves variables, they can be stored in arrays just as other variables can. For example, an array of `N` pointers to `int`s has the following syntax.

```
int *parray[N];
```

Each pointer in an array of pointers behaves as any ordinary pointer would. They might point to an object, to NULL, to an illegal memory location, or to an array.

```
double val = 9.7;
double array[] = { 3.2, 4.3, 5.4 };
double *pa[] = { &val, array+1, NULL };
```

In the above example, element `pa[i]` is a pointer to a double, and `*pa[i]` is the double variable that it points to. The dereferenced `*pa[0]` is equal to 9.7, and `*pa[1]` is 4.3, but `pa[2]` is equal to `NULL` and may not be dereferenced.

If an element in an array of pointers also points to an array, the elements of the pointed-to array may be accessed in a variety of different ways. Consider the following example.

```
int a1[] = { 1, 2, 3, 4 };
int a2[] = { 5, 6, 7 };
int *pa[] = { a1, a2 }; /* pa stores pointers to beginning of each array. */
int **pp = pa;          /* Pointer-to-a-pointer holds address of beginning of pa. */
int *p = pa[1];         /* Pointer to the second array in pa. */
int val;

val = pa[1][1]; /* equivalent operations: val = 6 */
val = pp[1][1];
val = *(pa[1] + 1);
val = *(pp[1] + 1);
val = *(*(pp+1) + 1);
val = p[1];
```

Notice that in an expression `pa` and `pp` are equivalent, but the difference is that `pa` is an array name and `pp` is a pointer. That is, `pp` is a variable and `pa` is not.

Arrays of pointers are useful for grouping related pointers together. Typically these are one of three types: pointers to large objects (such as `struct`s), pointers to arrays, or pointers to functions.

```
char *s;
s = string_duplicate("this is a string");
...
free(s); /* Calling function must remember to free s. */
```

Neglecting the `free(s)` statement means that the memory is not released even when `s` goes out-of-scope, after which the memory becomes non-recoverable. This sort of error is known as a "memory leak", and can accumulate large quantities of dead memory if the program runs for a long period of time or allocates large data-structures.

Some common errors related to dynamic memory management are listed below.

- Dereferencing a pointer with an invalid address. If a pointer is not initialised when it is defined, it will contain an arbitrary value, such that it points to an arbitrary memory location. The result of dereferencing this pointer will depend on where it points (e.g., no effect, intermittent strange values, or program crash). Writing to memory via an invalid pointer is known as "memory corruption".

- Dereferencing a pointer that has been freed. This is a special case of the previous error. Once a memory block is released by calling `free(p)`, the pointer `p` is no longer valid, and should not be dereferenced.

- Dereferencing a `NULL` pointer. This typically occurs because a system function returns `NULL` to indicate a problem and the calling function fails to implement appropriate checking. (For many compilers, dereferencing a `NULL` pointer is a simple bug to find as it causes the program to crash immediately, but this behaviour is not standard.)

- Freeing memory that has already been freed. Passing a previously freed pointer to `free()` will cause the function to dereference an invalid address.

- Freeing a pointer to memory that was not dynamically allocated. Memory that is not allocated on the heap, but on the stack or constant data area, say, cannot be released by `free()`. Attempting to do so will have undefined results.

- Failing to free dynamically allocated memory. Dynamic memory exists until it is explicitly released by `free()`. Failing to do so results in a "memory leak".

- Attempting to access memory beyond the bounds of the allocated block. Out-of-bounds errors occur if arrays are not properly bounds checked. A particularly common problem is the "off-by-one" array indexing error, which attempts to access elements one-before-the-beginning or one-past-the-end of an array, due to indexing arithmetic being not quite correct.

Good programming practices exist avoid these memory management errors, and following these rules will greatly reduce the risk of dynamic memory related bugs.

- Every `malloc()` should have an associated `free()`. To avoid memory leaks and memory corruption, there should be a one-to-one mapping of calls to `malloc()` and calls to `free()`. Preferably the call to `free()` should appear in the same function as the call to `malloc()` (rather than have one function return a pointer to dynamically allocated memory and expect the calling function to release it). Alternatively, one might write a `create()` function that allocates memory for an object and a companion `destroy()` function that frees it.

- Pointers should be initialised when defined. A pointer should never hold an arbitrary value, but should be initialised with either a valid address or `NULL`, which explicitly marks a pointer as "points nowhere".

- Pointers should be assigned `NULL` after being freed. A pointer that has the value `NULL` cannot be accidentally freed twice, as `free(NULL)` has no effect.

```
1  double **create_matrix(int m, int n)
2  /* Dynamically allocate an (m x n) matrix. Returns a pointer to the
3   * beginning of the matrix, and NULL if allocation fails. */
4  {
5          double **p, *q;
6          int i;
7          assert(m>0 && n>0);
8
9          /* Allocate pointer array. */
10         p = (double **)malloc(m * sizeof(double *));
11         if (p == NULL) return p;
12
13         /* Allocate entire matrix as a single 1-D array. */
14         q = (double *)malloc(m * n * sizeof(double));
15         if (q == NULL) {
16                 free(p);
17                 return NULL;
18         }
19
20         /* Assign pointers into appropriate parts of matrix. */
21         for (i = 0; i < m; ++i, q += n)
22                 p[i] = q;
23
24         return p;
25 }
```


1–11 This code is virtually identical to the previous version of `create_matrix()`.

14–18 Rather than allocate each row separately, memory for the entire matrix is allocated as a single block. If this allocation fails, it is a simple matter to free `p` and return `NULL`. This implementation completely bypasses the `goto` and its more complex error-handling code.

21–22 Having allocated memory, the remaining operations cannot fail, and so do not require error checking. The pointers in the pointer array are assigned to elements in the `double` array at `n` element intervals—thus, defining the matrix.

The `destroy_matrix()` code is also greatly simplified by allocating the matrix elements as a single block. First, the size of the matrix is not required, removing the possibility of passing incorrect dimension values. And, second, the deallocation operations are performed in two lines. Note, these lines must occur in the right order as `p[0]` is invalid if `p` is freed first.

```
void destroy_matrix(double **p)
/* Destroy a matrix. Notice, due to the method by which this matrix
 * was created, the size of the matrix is not required. */
{
        free(p[0]);
        free(p);
}
```

Given the final versions of the matrix create and destroy functions, a matrix might be used as follows.

```
double **m1, **m2;

m1 = create_matrix(2,3);
m2 = create_matrix(3,2);

/* Initialise the matrix elements. */
```

```
62
63 int set_capacity(int size)
64 /* Shrink or grow allocated memory reserve for array.
65  * A size of 0 deletes the array. Return 0 if successful, -1 if fails. */
66 {
67         if (size != capacity) {
68                 int *p = (int *)realloc(data, size*sizeof(int));
69                 if (p == NULL && size > 0)
70                         return -1;
71
72                 capacity = size;
73                 data = p;
74         }
75
76         if (size < vectorsize)
77                 vectorsize = size;
78         return 0;
79 }
```

44–45 These two trivial functions return the current state of the vector in terms of its size and available space, respectively.

47–61 The function `set_size()` changes the size of the array to the specified size. If this size is greater than the current capacity, then extra memory is allocated to suit. However, if the size is reduced, `capacity` is left unchanged; this function cannot decrease the available storage.

63–79 The function `set_capacity()` provides direct access to the memory allocation of the vector. It can be used to increase or decrease the available storage. If the requested size is less than the current vector size, the vector is truncated to fit. If the requested size is zero, `realloc()` will release the memory pointed to by its first argument and return `NULL`, effectively deleting the vector. Notice that a request of size zero, will cause `data` to become `NULL` (line 73), and `vectorsize` and `capacity` to become zero—thus, returning the vector to its original state.

The above implementation of an expandable array is a good example of modular programming as far as it goes, but it is subject to a couple of serious limitations. First, it provides a single instance of the vector only, and we cannot have more than one in any given program. Second, the vector can only contain elements of type `int`. To create a vector for other types would require a new, virtually identical, implementation for each type. Neither of these limitations are insurmountable, but they require the use of language features not yet covered, such as `struct`s, `union`s, `typedef`, and the `void*` pointer. The application of these features to writing generic code, is the subject of Chapter 14.

```
if (a > b)  { temp=a; a=b; b=temp; }
;
else a = b;
```

A solution to this problem is to wrap the body of the macro in a `do-while` statement, which will consume the offending semicolon.

```
#define SWAP(x,y,tmp)   do { tmp=x; x=y; y=tmp; } while (0)
```

An alternative solution is to wrap the macro in an `if-else` statement.

```
#define SWAP(x,y,tmp)   if(1) { tmp=x; x=y; y=tmp; } else
```

A variant of `SWAP` does away with defining an explicit temporary variable by simply passing the variable type to the macro.

```
#define SWAP(x,y,type)  do { type tmp=x; x=y; y=tmp; } while (0)
```

This might be used as

```
SWAP(a, b, double);
```

Finally, a very tricky *bitwise* technique allows us to perform the swap operation without any temporary variable at all. (However, this variant is only valid if `x` and `y` are integer variables of the same type.)

```
#define SWAP(x,y)       do { x^=y; y^=x; x^=y; } while (0)
```


10.3.3 More Complex Macros

Normally a macro definition extends to the end of the line beginning with the command `#define`. However, long macros can be split over several lines by placing a \ at the end of the line to be continued. For example,

```
#define ERROR(condition, message) \
    if (condition) printf(message)
```

A more interesting example, adapted from [KP99, page 240], performs a timing loop on a section of code using the standard library function `clock()`, which returns processor time in milliseconds.[3]

```
#define TIMELOOP(CODE) {            \
    t0 = clock();                   \
    for (i = 0; i<n; ++i) { CODE; } \
    printf("%7d ", clock() - t0);   \
}
```

This macro might be used as follows.

```
TIMELOOP(y = sin(x));
```

It is possible to convert a token to a string constant by writing a macro that uses the `#` symbol in the manner of the following example.

```
#define PRINT_DEBUG(expr) printf(#expr " = %g\n", expr)
```

[3] Measuring the runtime of code is a tricky business as the function in question might have a short execution time compared to the latency of the timing function itself. Thus, reasonable measurements require running the function multiple times and timing the period of the entire loop.

Unions are usually used for one of three purposes. The first is to create a "variant" array—an array that can hold heterogeneous elements. This can be performed with a reasonable degree of type safety by wrapping the union within a structure which records the union variable's current type. For example,

```
typedef union { /* Heterogeneous type. */
    int ival;
    float fval;
} Utype;

enum { INT, FLOAT }; /* Define type tags. */

typedef struct {
    int type;  /* Tag for the current stored type. */
    Utype val; /* Storage for variant type. */
} VariantType;

VariantType array[50];  /* Heterogeneous array. */
array[0].val.ival = 56; /* Assign value. */
array[0].type = INT;    /* Mark type. */
...

for (i = 0; i < sizeof(array)/sizeof(array[0]); ++i) /* Print array. */
    if (array[i].type == INT)
        printf("%d\n", array[i].val.ival);
    else if (array[i].type == FLOAT)
        printf("%f\n", array[i].val.fval);
    else
        printf("Unknown type\n");
```


Checking for the correct type remains the programmer's responsibility, but encoding the variable type in a structure eases the pain of recording the current state.

The second use of a `union` is to enforce the alignment of a variable to a particular address boundary. This is a valuable property for implementing memory allocation functions. And the third key use of a `union` is to get "under the hood" of C's type system to discover something about the computer's underlying data representation. For example, to print the representation of a floating-point number, one might use the following function (assuming `int` and `float` are both four-bytes).

```
void print_representation(float f)
/* Print internal representation of a float (adapted from H&S page 145). */
{
    union { float f; int i; } fi = f;
    printf("The representation of %e is %#x\n", fi.f, fi.i);
}
```

Both the second and third uses of unions described here are advanced topics, and a more complete discussion is beyond the scope of this text.

The C language has no mechanism to represent binary numbers directly, but provides ways to represent values in decimal, octal, and hexadecimal. For most purposes, decimal numbers are most convenient, but for bitwise operations, a hexadecimal representation is usually more appropriate. Hexadecimal (base 16) numbers effectively break binary values into blocks of four, making them easier to manage. With experience, hex tends to be more comprehensible than plain binary.

12.2 Bitwise Operators

C provides six bitwise operators: AND `&`, OR `|`, exclusive OR (XOR) `^`, NOT `~`, left-shift `<<`, and right-shift `>>`. These operators may be applied only to integer operands, `char`, `short`, `int`, and `long`, which may be signed or unsigned; they may not be used with floating-point operands. The `&`, `|`, `^`, `<<`, and `>>` operators each have corresponding assignment operators `&=`, `|=`, `^=`, `<<=`, and `>>=`. These behave analogously to the arithmetic assignment operators. For example,

```
z &= x | y;
```

is equivalent to

```
z = z & (x | y);
```

Important. As mentioned in Section 2.9, the bitwise operators, `&`, `|`, and `~`, are different from the logical operators AND `&&`, OR `||`, and NOT `!`, and should not be confused with them. These differences will be made clear in the discussion below.

12.2.1 AND, OR, XOR, and NOT

The `&`, `|`, `^`, and `~` operators permit Boolean logic operations on individual bits. The `~` operator is a unary operator, which simply converts a 0 to 1 and vice-versa. The other three are binary operators, which compare two bits according to the rules in the following table.

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Consider the following 8-bit (1-byte) example. We define three unsigned variables

```
unsigned char x = 55; /* 55 (dec) = 0x37 (hex) = 0011 0111 (binary). */
unsigned char y = 25; /* 25 (dec) = 0x19 (hex) = 0001 1001 (binary). */
unsigned char z;
```

and perform a series of operations on them, storing the result in `z`. The first, `z = x & y`, makes `z` equal to 17.

```
0011 0111
0001 1001 &
0001 0001   /* result is 17 (dec) = 0x11 (hex) */
```

The bitwise `&` (AND) operator sets a bit in `z` to 1 only if both corresponding bits in `x` and `y` are one. This operator is typically used to reset bits to zero and to select certain bits for testing.

The second operation, `z = x | y`, makes `z` equal to 63.

Bitwise expressions tend to be faster than integer arithmetic, but such optimisations are generally redundant as modern compilers will tend to replace "power of two" arithmetic with bitwise operations automatically.

12.2.3 Operator Precedence

The precedence of the bitwise operators is lower than the arithmetic operators, with the exception of the unary `~` operator, which has equal precedence to the unary (logical) `!` operator. The left and right shift operators have greater precedence than the relational operators `<` and `>`, but `&`, `|`, and `^` have lower precedence. The precedence of `&` is greater than `^`, which is greater than `|`. All bitwise operators have greater precedence than the logical operators `&&` and `||`.

As with the arithmetic, relational and logical operators, it is unwise to rely on precedence in multi-operation expressions. Such practice tends to produce obscure and error-prone code. It is far better to make ones intent clear with parentheses.

12.3 Common Bitwise Operations

Bitwise operations are commonly used for one of two purposes. The first is to conserve space by packing several variables into a single byte (e.g., binary "flags" that need only represent the values 0 or 1). The second is to interface with hardware, where a group of bits at a certain address correspond to the pins of some device. In both cases we want to be able to manipulate and test individual bits or groups of bits.

The following example presents some common idioms for dealing with bits, which allow us to turn bits on or off and to test their current state. These techniques are collectively known as "masking" as they enable particular bits to be selected according to a specified *bit-mask*. The first step in creating a mask is to define variables to represent each bit of the integer variable; (we will consider only the lowest 4 bits in this, rather contrived, example).

```
enum {
    FIRST = 0x01, /* 0001 binary */
    SECND = 0x02, /* 0010 binary */
    THIRD = 0x04, /* 0100 binary */
    FORTH = 0x08, /* 1000 binary */
    ALL = 0x0f    /* 1111 binary */
};
```

The constants are each powers of two, so that just one bit is set and the rest are zeros. The exception is `ALL`, which defines a mask to select all four bits. An equivalent way to define the above constants is to use the left-shift operator as follows.

```
enum {
    FIRST = 1 << 0,
    SECND = 1 << 1,
    THIRD = 1 << 2,
    FORTH = 1 << 3,
    ALL = ~(~0 << 4)
};
```

The last of these is a trifle cryptic, but is a common technique for creating a specified number of ones. The key is to realise that `~0` creates a value where all bits are 1's. For example, (assuming we are dealing with an 8-bit value)

```
1111 1111 /* ~0 */
```

The `scanf()` format string consists of conversion specifiers, ordinary characters, and white-space. Where ordinary characters appear in the format string, they must match exactly the format of the input. For example, the following statement is used to read a date of the form `dd/mm/yy`.

```
int day, month, year;
scanf("%d/%d/%d", &day, &month, &year);
```

In general `scanf()` ignores white-space characters in its format string, and skips over white-space in `stdin` as it looks for input values. Exceptions to this rule arise with the `%c` and `%[` conversion specifiers, which do not skip white-space. For example, if the user types in "`one two`" for each of the statements below, they will obtain different results.

```
char s[10], c;
scanf("%s%c", s, &c);  /* s = "one", c = ' ' */
scanf("%s %c", s, &c); /* s = "one", c = 't' */
```

In the first case, the `%c` reads in the next character after `%s` leaves off, which is a space. In the second, the white-space in the format string causes `scanf()` to consume any white-space after "`one`", leaving the first non-space character (`t`) to be assigned to `c`.

While the many details of `scanf()` formatting complicates a complete understanding, its basic use is quite simple. Rarely does an input statement get more complicated than

```
short a;
double b;
char c[20];
scanf("%hd %lf %s", &a, &b, c);
```

However, it is worth noting that the above form of string (`%s`) input is not ideal. A string is read up to the first white-space character unless terminated early by a width field. Thus, a very long input of consecutive non-space characters may overflow the string's character buffer. To prevent overflow, a string conversion specification should always include a width field. Consider a situation where a user types in the words "`small supererogatory`" for the following input code.


```
char s1[10], s2[10], s3[10];
scanf("%9s %9s %9s", s1, s2, s3);
```

Notice the width fields are one-less than the array sizes to allow room for the terminating `\0`. The first word "`small`" fits into `s1`, but the second word is over-long—its first nine characters "`supererog`" are placed in `s2` and the rest "`atory`" goes into `s3`.

A few final warnings about `scanf()`. First, keep in mind that the arguments in its variable length argument list *must* be pointers; forgetting the `&` in front of non-pointer variables is a very common mistake. Second, when there is a conflict between a conversion specification and the actual input, the offending character is left unread. Thus, an expression like

```
while (scanf("%d", &val) != EOF)
```

is dangerous as it will loop forever if there is a conflict. Third, while `scanf()` is a good choice when the exact format of the input is known, other input techniques may be better suited if the format may vary. For example, the combination of `fgets()` and `sscanf()`, described in the next section, is a useful alternative if the input format is not precisely known. The `fgets()` function reads a line of characters into a buffer, and `sscanf()` extracts the data, and can pick out different parts using multiple passes if necessary.

```
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);
int putchar(int c);
```

where calling `putchar(c)` is equivalent to calling `putc(c, stdout)`. The functions `putc()` and `fputc()` are identical, but `putc()` is typically implemented as a macro for efficiency. These functions return the character that was written, or `EOF` if there was an error (e.g., the hard disk was full).

To read a character, there are the functions

```
int fgetc(FILE *fp);
int getc(FILE *fp);
int getchar(void);
```

which are analogous to the character output functions. Calling `getchar()` is equivalent to calling `getc(stdin)`, and `getc()` is usually a macro implementation of `fgetc()`.[8] These functions return the next character in the character stream unless either the end-of-file is reached or an error occurs. In these anomalous cases, they return `EOF`. It is possible to push a character `c` back onto an input stream using the function

```
int ungetc(int c, FILE *fp);
```

The pushed back character will be read by the next call to `getc()` (or `getchar()` or `fscanf()`, etc) on that stream.

Note. The symbolic constant `EOF` is returned by standard IO functions to signal either end-of-file or an IO error. For input functions, it may be necessary to determine which of these cases is being flagged. Two standard functions, `feof()` and `ferror()`, are provided for this task and, respectively, they return non-zero if the prior `EOF` was due to end-of-file or an output error.

Formatted IO can be performed on files using the functions

```
int fprintf(FILE *fp, const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
```

These functions are generalisations of `printf()` and `scanf()`, which are equivalent to the calls `fprintf(stdout, format, ...)` and `fscanf(stdin, format, ...)`, respectively.

Characters can be read from a file a line at a time using the function

```
char *fgets(char *buf, int max, FILE *fp);
```

which reads at most `max-1` characters from the file pointed to by `fp` and stores the resulting string in `buf`. It automatically appends a `\0` character to the end of the string. The function returns when it encounters a `\n` character (i.e., a newline), or reaches the end-of-file, or has read the maximum number of characters. It returns a pointer to `buf` if successful, and `NULL` for end-of-file or if there was an error.

Character strings may be written to a file using the function

```
int fputs(const char *str, FILE *fp);
```

which returns a non-negative value if successful and `EOF` if there was an error. Note, the string need not contain a `\n` character, and `fputs()` will not append one, so strings may be written to the same line with successive calls.

[8] While `putc()` is equivalent to `fputc()` and `getc()` is equivalent to `fgetc()`, it is important to note that the line IO functions `puts()` and `gets()` are not equivalent to their counterparts `fputs()` and `fgets()`. In fact, the function `gets()` is inherently flawed in its inability to limit the size of an input string and `fgets()` should always be used in preference.

implementations to perform the bulk of the algorithm but provide interfaces that permit type checking and type conversion by the compiler. Consider the following set of wrapper functions for `Vector`s of type `int`. The header file `vector_int.h` contains the public interface.

```c
#ifndef INT_EXPANDABLE_VECTOR_H_
#define INT_EXPANDABLE_VECTOR_H_

#include "vector.h"

/* Vector creation. */
Vector *vector_create_int(size_t capacity);

/* Vector access operations. */
int vector_push_back_int(Vector *v, int item);
int vector_pop_back_int(Vector *v);

#endif
```

There are several points to note from this header file. The first is operations such as pushing an integer onto the array may be performed with expressions or constants.

```c
vector_push_back_int(v, 50);
vector_push_back_int(v, i + j);
```

The second is that items of the wrong type are automatically converted to `int`s by the usual type conversion rules.

```c
char val = 32;
vector_push_back_int(v, val); /* val automatically converted to int. */
```


Finally, notice that wrappers are not provided for most of the generic public interface. This is because most operations do not require a type-specific wrapper, and the generic interface can be used directly without issue. For example, `vector_destroy()`, `vector_get_element()`, `vector_set_size()`, etc, do not rely on type information.

Style Note. It is good practice to avoid including header files in other header files where possible. This is in order to minimise dependencies between different modules. In the case of `vector_int.h`, the inclusion of `vector.h` could be avoided, and replaced with

```c
typedef struct Vector Vector;
```

as the declarations in `vector_int.h` make no reference to any other part of the `Vector` public interface. Rather, `vector.h` would be included in the source file `vector_int.c`, and the dependence between the two headers is reduced to a single declaration.

We have chosen to include `vector.h` in `vector_int.h` on this occasion because the two modules are inherently coupled. We never call `vector_create_int()` without calling the generic function `vector_destroy()`. Thus, there is no need to minimise their dependence.

The next set of functions are the contents of the source file `vector_int.c`. These functions call the generic functions to perform the actual operations, but also incorporate some type-checking code. In the following implementations, checking is very primitive—simply that the passed vector contains items of the appropriate size, which protects against memory errors. They do not check whether the actual element types are correct, and different types of compatible size will not be caught. It is possible to strengthen this type-checking by including a type-field in the `Vector` structure similar to that used for unions in Section 14.1.3.

```c
#include "vector_int.h"
```

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

#define NELEMS(x) (sizeof(x)/sizeof(x[0]))

struct Database {
        int key;
        float item;
};

int comp_dbase(const void *a, const void *b)
/* Returns -ve if a<b, 0 if a==b, +ve if a>b */
{
        struct Database *d1 = (struct Database *)a;
        struct Database *d2 = (struct Database *)b;

        if (d1->key < d2->key)
                return -1;
        if (d1->key > d2->key)
                return 1;
        return 0;
}

int main(void)
{
        int i;
        struct Database db[10];

        for (i = 0; i < NELEMS(db); ++i) {
                db[i].key = rand();
                db[i].item = (float)r[illegible]
        }

        qsort(db, NELEMS(db), sizeof db[0], comp_dbase);

        for (i = 0; i < NELEMS(db); ++i)
                printf("%5d %.1f\n", db[i].key, db[i].item);
}
```


The power of `qsort()` is that it may be used with arrays of any arbitrary data-type such as,

```
struct Dictionary {
    char *word;
    char *defn;
};
```

and each different data type may be compared via its own particular comparison function, as in the following example.

```
int comp_dict(const void *a, const void *b)
{
    struct Dictionary *d1 = (struct Dictionary *)a;
    struct Dictionary *d2 = (struct Dictionary *)b;
    return strcmp(d1->word, d2->word);
}
```

Thus, if we were to create an array `dt[100]` of type `struct Dictionary`, we could sort it as follows.

```
qsort(dt, NELEMS(dt), sizeof(dt[0]), comp_dict);
```

Chapter 16

C in the Real World

This text has covered most of the core ISO C language and its use. However, virtually all useful software systems make use of some form to extension to standard C. This chapter provides a sampling of the virtually limitless field of extensions and related topics with regard to writing C programs in the real world. Knowledge of the core language is the foundation upon which all these additional topics rely.

TODO: complete this chapter...

16.1 Further ISO C Topics

There are many details of ISO C and the standard library that are not covered in this text. For the most part, these topics are peripheral, and do not impinge on the majority of application programming. They include:

- Complete rules of operator precedence and order of evaluation.
- Keywords such as `register` and `volatile`.
- Memory alignment and padding.
- Changes to the standard with ISO C99. For the most part, this standard to backward compatible with C89, and the older standard currently remains the more important language in practice.

One topic that is fundamental but cannot be adequately covered in this book is the standard library; the majority of standard functions are not even mentioned. These functions are frequently useful and are worthy of study. They include, input and output (`stdio.h`), mathematical functions (`math.h`), strings (`string.h`), utilities (`stdlib.h`), time (`time.h`), floating-point specifications (`float.h`), errors (`errno.h`), assertions (`assert.h`), variable-length argument lists (`stdarg.h`), signal handling (`signal.h`), non-local jumps (`setjmp.h`), etc.

For more on these and other topics, consult a good reference textbook. A complete and authoritative reference is [HS95, HS02], and is highly recommended for practicing programmers. An excellent FAQ [Sum95] on the C language discusses many of the more difficult aspects. It is worth noting that many C idioms are not recorded in any textbook and can only be discovered from practical experience and reading the source code of others.

Note. Different compilers may conform to the standard to different extent. They might not permit conforming code to compile, or it might exhibit non-standard behaviour. This is less likely with modern compilers. More likely is allowing non-standard code to compile. As a rule, it is wise to compile code on several different compilers to ensure standard conformance.

16.5 Interfacing With Libraries

- many open-source C libraries - other repositories: - source forge - planet source code - www.program.com/source - linux?? - netlib

- Separate ISO C conforming code from proprietry or platform specific - Interface with precompiled libraries, open-source libraries, - discuss libraries as an example of modular design.

16.6 Mixed Language Programming

There arise situations where a C program must call a set of routines written in another programming language, such as assember, C++, FORTRAN, Matlab, etc.

- Interfacing C with FORTRAN, assembler, C++, MatLab, etc. - binding

16.7 Memory Interactions

Historically, instruction count was a premium. Computer processors were slow and memory was tiny, and the speed of an algorithm was directly proportional to the number of instructions it required. Programmers spent a lot of effort finding ways to minimise instruction count. Most algorithm textbooks today continue to use this measure in their analysis of algorithm complexity.

Modern computers, with fast CPUs, are no longer constrained primarily by instruction execution. Today, the bottleneck is memory access. While ever waiting for instructions or data to be fetched from memory, the CPU is idle and cycles are wasted. To minimise idle time, modern computer architectures employ a *memory hierarchy*, a set of memory levels of different size and speed to permit faster access to frequently used information. This hierarchy, from fastest to slowest, consists of registers, cache, main random access memory (RAM), hard-disk, and magnetic tape. Very fast memory is small and expensive, while cheap large-scale memory, such as RAM, is relatively slow. Each level in the hierarchy is typically slower than the level above by several orders-of-magnitude. Information is transferred up and down the memory hierarchy automatically by the operating system, with the exception of magnetic tape, which is usually reserved for memory backup. Essentially all modern operating systems manage the transfer of data between RAM and hard-disk, so that the hard-disk appears as additional, albeit slow, RAM known as *virtual memory*.

As the CPU accesses instructions or data, the required information is transferred up the hierarchy. If the information is already in registers, it can be executed immediately. If it resides in cache, it is moved up to the registers and the old register data is transferred back to cache. Similarly "lines" of RAM are moved up into cache, and "pages" of hard-disk memory are moved up to RAM. Since the amount of information that can be stored in the upper levels is limited, data that has not been accessed recently is passed back down to lower levels. For example, if all cache lines are full and a new line is required from RAM, the *least recently used* cache line is returned to RAM.

Information is transferred between levels in blocks, so that when a particular item is accessed, it brings with it a neighbourhood of instructions or data. Thus, if the next item required was a neighbour, that item is already in cache and is available for immediate execution. This property is called "*locality of reference*" and has significant influence on algorithm speed. An algorithm with a large instruction count but good locality may perform much faster than another algorithm with smaller instruction count. Some algorithms that look fast on paper are slow in practice due to bad cache interaction.

There are various factors that affect locality of reference. One is program size. There is usually a tradeoff between size and speed, whereby to use less memory the program requires the execution of more instructions and vice-versa. However, a program that is optimised for size, that attempts to occupy minimal space, may also achieve better speed as it is better able to fit within cache lines. Another factor is data-structures. Some data-structures such as link-lists may develop bad locality if naively implemented, whereas others, such as arrays, possess very good locality. A third

Bibliography

[Ben00] J. Bentley. *Programming Pearls*. Addison-Wesley, 2nd edition, 2000.

A unique, interesting and practical book about programming in the real world. Contains many clever ideas and thought-provoking problems. It covers many aspects of efficiency, particularly space efficiency, not found in other texts.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

A rigorous and comprehensive book on algorithms and data-structures. The word "introduction" should not be misunderstood, it does not mean a simplistic book, rather it indicates that this text is an entry-point to the vast and diverse literature on the subject of computer algorithms.

[HS95] S.P. Harbison and G.L. Jr. Steele. *C: A Reference Manual*. Prentice-Hall, 4th edition, 1995.

This book presents C from a compiler-writers perspective; Harbison and Steele have built C compilers for a wide range of processors. It is an excellent reference documenting and cross-referencing every detail of the C language.

[HS02] S.P. Harbison and G.L. Jr. Steele. *C: A Reference Manual*. Prentice-Hall, 5th edition, 2002.

The current edition of H&S includes discussion of the new C99 standard.

[Knu98a] D.E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 3rd edition, 1998.

The seminal three-volume work of Knuth set the study of computer algorithms on its feet as a scientific discipline. It is a rigorous and authoritative source on the properties of many classical algorithms and data-structures.

[Knu98b] D.E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 3rd edition, 1998.

[Knu98c] D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 2nd edition, 1998.

[KP99] B.W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

This book is a great source of expert advice on quality software design. It covers topics like coding-style, program design, debugging, testing, efficiency, and portability.

Index