Advanced C Techniques & Applications

Programming Style

This chapter presents several methods for solving complex programming in the C language. In particular, the text introduces techniques for dividing large tasks into sets of functions and smaller files so that the programs run efficiently and are easily portable among different C environments.

Separate Compilation and Information Hiding

The careful use of separate files can be used to hide implementation details. This practice not only simplifies and modularizes the total program, but also prevents the unintended overlap of variable or function names that may occur in a large program, especially if more than one person is involved in program design.

An external element can be defined only once. In addition to specifying type information, the definition allocates storage and may also set initial values. An external name is only visible in the file in which the variable is defined until the end of the file. If the variable or function name is needed at an earlier position in that file or in another file, the name can be declared with the `extern` keyword.

Information hiding is accomplished through the use of the external static storage class. If an external variable or function is defined with the `static` prefix, the variable can be accessed only within that file. Thus, functions and external variables can be shared throughout a file and yet be totally unknowable in any other file.

As an example of using `static` to hide methods:

```
static int p, g;
static double r, s;

static int f1(a, b);
int a, b;
{
    ...
}

static int f2(a, b, c)
char a, b, c
{
    ...
}

char utility (x, y, z)
int x, y, z
{
    ...
}
```

The function `utility` can make calls to functions `f1` and `f2`, and all three can read and write to the external variables `p`, `g`, `r`, and `s` However, only the `utility` function can be seen from another file. Therefore, if another file contains `extern char utility` at the beginning of the file, functions within that file can gain access to `utility` through a call like `letter = utility(x, y, z)`.

Header Files

Logically connected groups of constants and data types that are used in different parts of a large program should be collected in a single file (known as a header file) and simply copied (by means of the `#include` facility) into the files that require the use of those elements. Another good general practice is to avoid *hard coding* any numerical constant. Instead, define a suitable name for the constant and refer to that name when the value is needed.

C provides a technique that easily implements this type of program organization. The `#define` facility replaces a name with a character string throughout a file from the line of the name's definition until the end of the file. The replacement is done by a pre pass by the compiler before any syntactic or semantic analysis is performed.

External Variables

Header files are also convenient for declaring external variables and defining structures types. If a set of variables are to be shared by different files, the variables are defined only once, typically in the file containing the main routine. Then, a set of corresponding `extern` declarations can be assembled in separate file called `globals.h`.

Structure Types

Similarly, if a group of data types are needed in several files, an entire set of structure type definitions should be put into a single file.

Macros with Arguments

Another powerful feature of C is the capability of defining macros with arguments. This facility permits simple functions to be expressed as macros, thereby eliminating the overhead (parameter passing) associated with making function calls. *A single macro can be used with arguments of different data types and so be-have as a generic procedure.* This convenient feature improves the program readability and execution time.

As an example, the following macro computes the sum of its squares of its arguments:

```
// Sum of Squares
#define  SOS(x, y) ((x) * (x) + (y) * (y))
```

A function call and subsequent returns involve a certain amount of overhead; but with the macro, the compiler performs a direct in-line substitution of code. Specifically, on the compiler pre-pass, the compiler replaces the statement

```
e = SOS(a + b, c + d);

// with

e = ((a + b) * (a + b) + (c + d) * (c + d));
```

It is important to note from this example that parenthesis are important to ensure proper mathematical ordering. As a matter of safety, also enclose the entire macro definition in parenthesis. These parenthesis guarantee the correct precedence of operation even when the macro is used as an element in a complex expression.

Portability and Hardware Independence

When programming, it is always important to consider potability. Even though you may think that a particular program will always be compiled with a certain compiler and be run only on a given machine, circumstances may change. When new compilers and faster machines appear, a "dedicated application program" is suddenly a candidate for transfer to a new environment.

The consideration of portability also applies to any piece of code that is specific to a piece of hardware. This limitation typically occurs in input or output routines and may involve such details as the choice of color or monochrome for the display type, the number of horizontal pixels available, the use of specialty defined keys, or the properties of a positioning device like a mouse.

```
Application Routine -> Generic Interface Routine -> Device Dependent Routine
```

Any routine of the kind should be written using a layered approach, with the application routine calling the generic routine that make calls to other hardware specific functions. The application makes calls to a set of interface functions that are to perform the hardware specific tasks. Each of these interface functions would be implemented using the primitives of the target machine. In this way, the code contained in the application routine will be fixed; only the set of interface routines needs to be changed for a new environment.

Pointers and Dynamic Allocation

Pointers and Arrays

An array is effective only when the precise number of elements is known at compile time so that the correct storage can be allocated. If the number of elements cannot be known until run time, an array implementation either allocates too much space and wastes resources or, worse, does not allocate enough space and thereby creates a "disaster waiting to happen".

Pointers

Pointers can be used effectively in situations where the number of elements in the data structure changes as the program is running. Two C operators, & and *, give the programmer access to pointers.

The operator `$(...)` can be read as "the address off (…)". The operator `*(...)` stands for "the context of (…)"

In definitions and declarations, the * operator is also used to indicate that certain variables or functions are pointers.

Storage Allocators

When an array is defined, storage is automatically allocated by the compiler. When you are using pointers, however, you must allocate the storage. Two C standard library functions are provided for this purpose: `malloc` (memory allocate) and `calloc` (contiguous allocate). The `malloc` function allocates a single piece of storage space, `calloc` allocates an entire set of storage locations through a single function call. In addition `calloc` fills the total storage space with zeros, but `malloc` leaves the area uninitialized.

```
// malloc returns a character pointer
char * mallloc(size)
unsigned size;
{

}
```

In the definition above, `malloc` returns a character pointer. This may seem odd because this function is to be used for allocating space for any time of element. Because all C functions must be formally defined to return a specific type of variable, a character pointer was chosen arbitrarily. However, the function properly allocates the required space in all situations. The potential mismatch of types is easily fixed with a simple cast operation. Second, notice that the argument of the function, size, indicates the amount of storage to be allocated. In practice, `size` is normally the value returned by the `sizeof` operator. You can either supply the name of a literal C variable name or expression, or you can use a standard or use-defined variable type name.

Using `calloc`

When using `calloc`, you need to supply two pieces of information: the number of elements and the size of the individual elements.

Dealing with Changing Storage Needs

You want to create a string copy method called `strcpy`. We can do that using the `calloc` method as shown:

```
char *start, *cons, *calloc();
cons = "This is a C string constant";
start = calloc(strlen(cons) + 1, sizeof(char));
strcpy(start, cons);
```

Now consider the situation where you require dynamic storage allocation for a single `float` variable. First define a `float` pointer, then make a call to one of the storage allocation routines as follows:

```
float *place
place = (float *) malloc (sizeof(float));
```

Even though the `malloc` or `calloc` function sense that it must reserve storage for a float-sized element, the function still returns a character pointer. Because `place` is a `float` pointer, you can correct this mismatch by a cast operation that performs type coercion, the create of a new variable having the desired type.

Because the use of this combination of storage allocation and type coercion occurs so frequently, macros are useful to implement these operations automatically. In particular you may use,

```
#define MALLOC(x)   ((x*) malloc (sizeof(x)))
#define CALLOC(n, x)    ((x*) calloc (n, sizeof(x)))
```

Using `NULL` Pointers

A valid pointer value cannot be zero. This fact provides a scheme for detecting errors in requests for storage space.

```
if (place = MALLOC(float))
{
    // Proceed with intended action
}
```

```
else
{
    // Error
}
```

Freeing Blocks of Memory

To do this, the `free` method is employed. The method has a definition of:

```
free(location)
char* location
{
    ...
}
```

Note that the calls to free previous allocated blocks of storage can be made in any order, regardless of how the blocks were originally obtained.

Using `realoc`

This function changes the size of a previously allocated area of memory. The form of the function is as follows:

```
char *realloc(location, size)
char* location;
unsigned size;
{
    ...
}
```

Structures, Unions, and Fields

Structures can be used to create new data types. Variables can then be defined to be one of these types and manipulated within programs. The general format for defining structures is:

```
struct struct-tag
{
    // variable definitions
} variable-list;
```

This format is the most direct way of defining variables to be of a certain structure type; but for complex data structures and applications, the C `typedef` facility is more convenient and increases program readability.

```
struct program
{
    char* name
    int num_lines, num_chars,
    double run_time,
};

typedef struct program prog_t;

prog_t my_prog, ur_prog;
```

If you did not type define the structure, you would have to declare new variables of that structure type like: `struct program my_prog`.

Pointers To Structures

As with pointers to character strings, pointers to structures improve execution speed. If a structure is large, transferring pointers from one part of a program to another is much faster than copying all the individual components of the structure.

Structures cannot be passed as parameters or returned in functions, but you can get around this by passing pointers to structures.

When you use pointers to structures as function arguments, you can gain two advantages: pointers can serve simply as a convince; you do not have to list each structure component in the argument list and you can get around the "call by value" rule of C. When returning a pointer to a structure from a function, you achieve an advantage not otherwise available: the ability to return multiple variables from a function. All you need to do is define a new structure type whose components are

the set values that need to be returned. Each time the function is called, it returns a pointer that indicates where the data can be found.

A useful programming convention involving pointers and structures is the class of functions that *instantiate* (allocates and initializes) storage for elements of a given data type. For example, consider a data type that specified a rectangle.

```
struct rect_primitive
{
    int left, bottom, right, top;
};

typedef struct rect_primitive rect_t;
```

You can write a function that allocates space to hold a `rect_t` and initialize its components:

```
rect_t* inst_rect(l, b, r, t)
int l, b, r, t
{
    rect_t* box;

    if (box = MALLOC(rect_t))
    {
        box->left   = l;
        box->bottom = b;
        box->right  = r;
        box->top    = top;
    }
    else
    {
        printf("Allocation error on call to inst_rect with parameters: %d, %d, %d, and %d", l, b, r, t);
        exit(1);
    }

    return (box);
}
```

Structure Nesting

You can also make structures the elements in other larger structures. Nesting may be accomplished by placing structures inside other structures or by using pointers to structures as the elements. The latter approach promotes generic code and my allow faster data transfer.

Remember that if a structure contains pointers, allocating space for the structure does *not* allocate space for the elements that are pointed to by those pointers. This space must be allocated separately by appropriate calls to the macros `MALLOC` or `CALLOC`.

Structures may contain components that are pointers to structures of that same type. You will have problems if you attempt to place actual instances of a structure inside of itself, but all that is being discussed here is using pointers to that elements.

These recursive structure definitions are useful in many complex data types such as lists and trees.

```
struct self
{
    int x, y, z;
    struct self *p, *g, *r;
}
typedef struct self self_t;

self_t s, t, u;
```

Unions

A union provides a way for different types of data to be stored in a common block of memory space. This feature is useful in situations where a program uses items that play logically similar roles, but these items differ in their detailed forms. *The compiler allocates sufficient memory sufficient space to hold the largest item in the union (if the items are of different sizes), but the programmers is responsible for keeping track of what is currently stored in the union.*

A simple of a union is as follows:

```
union value
{
    int    int_item;
    double dbl_item;
};
typedef union value value_t;
```

To *keep track* of what type of item is stored at any given time, define an associated integer variable and form a composite element, known as a *variant record* as follows:

```
struct var_rec
{
    int type;
    value_t item;
}
typedef struct var_rec var_rec_t;
```

In this case you can use a series of `#defines` to encode the indicated type:

```
#define INT 0
#define DBL 1
```

The rules for accessing a union is the same as structures.

Fields

In some programming applications, you must manipulate individual bits with certain registers or memory locations. This necessity arises in two different situations: memory space may be at a premium, and ever tying must be done to conserve storage, or you may need to interact with specific hardware, such as with an external device driver routine. C provides a data construct, called a *field* that makes either situation easy to deal with.

As an example, consider a situation in which you must keep track of the data corresponding to a parallel port in a micro-processor system. You will have

```
struct port
{
    unsigned bit_0 : 1;
    unsigned bit_1 : 1;
    unsigned bit_2 : 1;
    unsigned bit_3 : 1;
    unsigned bit_4 : 1; // low byte
    unsigned bit_5 : 1;
    unsigned bit_6 : 1;
    unsigned bit_7 : 1;
    unsigned hi_by : 8; // high byte
};
typedef struct port port_t;
```

Where the structure for the union is the same.

The integer after the colon is the number of bits assigned to that variable. Because all these quantities are highly machine dependent, the field definition properly (the code listing above) belongs in a file separate from the rest of the application program.

When the compiler encounters a field definition, the compiler allocates storage as if the field were one ore more integer variables. For example, in a very small machine that has integers stored as eight-bit quantities, a variable of type `port_t` requires two integer variable positions. On the other hand, for machines where integers are sixteen bits or larger, a single integer location suffices.

The following are important facts when using fields: the detailed storage of fields is machine dependent, arrays of fields may not be formed, and trying to take the address of a field component is illegal.