Functions in C

Java supports *constructors* and *methods* which allocate instances of, and interrogate and modify the state of, their own (implicit) objects. Constructors and methods are typically directed by their parameters. C is a procedural programming language, meaning that its primary synchronous control flow mechanism is the function call. Strictly speaking, C has no procedures, but instead has functions, all of which return a single instance of a base or user-defined type. C's functions access and modify the global memory, and (possibly) their parameters. Although we may hope that a function can only modify memory that it can "see" (through C's scoping rules) or has been provided (through its parameter list), this is untrue.

By stating that there are only functions, in we suggest that all functions must return a value. While *nearly* true, C also has a **`void`** type, difficult to describe, and often used as a place holder (to keep the compiler happy!). We may think of a procedure in C, as a function that returns a **`void`** – nothing is returned. With a similar thought, we will often invoke a function, but have no use for its return value. For example, a function such as `printf()` will return an integer as its result, but we rarely need to use this integer. We can "cast its value" to **`void`**, effectively throwing away the value.

```
printf( .... );
```

The default return datatype of a function is **`int`** – if a function's datatype is omitted, the compiler assumes it to be an **`int`**. This has the unpleasant result, that if an external or yet to be declared function's prototype is omitted, the compile will often silently assume an **`int`** return result. This is a frequent cause of problems, particularly when dealing with functions returning floating point values, as in C's mathematics library. The use of *gcc's* `-pedantic` switch allows us to trap most such errors.

Every complete C program has an entry point named *`main`*, at which it appears the operating system calls the program. Function `main` is of type **`int`** – this **`int`** is returned as the "result" of execution of the whole program, with 0 indicating a successful execution, anything non-zero otherwise.

C's functions may receive zero or more parameters. *All* parameters to C's functions are passed by value. Other than within a single file, the datatype of function parameters between the function's declaration and invocation is not checked, i.e. C provides no link-time cross file type checking. Perhaps surprisingly, C also permits functions to receive a variable number of parameters. At run-time it is the function's responsibility to deal with the data types received, and the compiler cannot perform any type checking on these parameters.

Function parameters are implicitly *promoted* to "higher" datatypes by the compiler – **`char`**s are promoted to **`int`**s, and **`float`**s are promoted to **`double`**s.

The *cdecl* program

An interesting LINUX program named *cdecl* may be used to explain C (and C++) variable declarations in pseudo-English, and to present C language syntax from pseudo-English requests.

-
```
-> help

[] means optional; {} means 1 or more;
<> means defined elsewhere
      command:
      declare <name> as <english>
      cast <name> into <english>
      explain <gibberish>
etc ...

-> declare fred as function returning int
   int fred()

-> explain long x()
   declare x as function returning long
```

Data structures in C

C has no equivalent construct to the Java class. Instead, C provides two aggregate data structures – arrays and structures.

Arrays in C are not objects, nor strictly single variables. Instead, an array's name is the name referring to the first memory address of a contiguous block of memory of the requested length. Arrays may be declared wherever scalar variables are declared – arrays may be either arrays of C's base types or user-defined types.

There is no *array* keyword in C, and no bounds checking at run-time. C array subscripts commence at 0, the highest valid subscript of `int a[N]` thus being N-1.

- One dimensional arrays Declared with (for example) **int** score[20];

```
-> declare score as array of 20 int
   int score[20]

    total = 0;
    for(i=0 ; i<20 ; i++)
         total = total + score[i];
```

- Multi-dimesntional arrays?
  Strictly speaking, C does not support multi-dimensional arrays. However, if all (one-dimensional) arrays in c are considered as *vectors*, then multi-dimensional arrays are simply understood as "vectors of vectors".

```
-> explain char str[10][20]
   declare str as array of 10 array of 20 char
```

The number of elements of an array can be determined with :

```
#define  NELEMENTS  (sizeof(score) / sizeof(score[0]))

    for(i=0 ; i<NELEMENTS ; i++)
         total = total + score[i];
```


User-defined C Structures

Structures in C are aggregate datatypes consisting of *fields* or *members* of base types, or other user-defined types. C structures may not include executable code, unlink methods in Java classes.

```
struct person {
    char name[20];
    char addr[80];
    int  age;
};

struct person p1, p2;
int    ages;

ages = p1.age + p2.age;
 /* the sum of their ages */

if(strcmp(p1.name, p2.name) == 0) ...
 /* do they have the same name? */
```

Character arrays and strings

C provides no base type that is a string, though the C compiler accepts the use of double quoted character string literals and ‘‘does the obvious thing’’. A string in C is a sequence of characters (bytes) in contiguous memory locations. The string is terminated by the sentinel value of the *NULL* character (zero byte). When a C compiler detects a string literal in a program, it will allocate enough contiguous global (read-only) memory to hold the characters of the string (including the NULL byte at the end).

C does not record the *length* of a string anywhere (as does Pascal and Java). Instead, by convention, the length of a string is defined as the number of characters from the beginning of the string (its starting address) up to, but not including, the NULL byte. The length of `"hello"` is 5.

- Arrays of characters are typically used to store character strings. Notice that the parameter to the following function does not indicate any expected (maximum) size, or ‘‘length’’, of the array.

```
int my_strlen(char str[])
{
    int i = 0, len = 0;

    while( str[i] != '\0' ) {
        len++;
        i++;
    }
    return(len);
}
```


Pointers in C

The C programming language has a very powerful feature, and if used incorrectly a very dangerous feature, which allows a program (at run-time) to access its own memory. This ability is well supported in the language through the use of *pointers*. There is much written about the power and expressiveness of C’s pointers, and much (more recently) written about Java’s lack of pointers. More precisely, Java *does* have pointers, termed *references*, but the references to Java’s objects are so consistently and carefully constrained at both compile and run-time, that very little can go wrong.

C has both ‘‘standard’’ variables and structures, and pointers to these variables and structures (Java *only* has references to objects, and it is only possible to manipulate the computer’s memory used to hold the objects, by using references). C’s drawback is that while the pointers allow us to easily refer to scalar variables and aggregate structures, C has very little support to prevent us accessing anything else (accidently) at run-time. All speed advantages provided by the availability of pointers, can be trivially consumed by the time taken to debug a program incorrectly using pointers.

C’s pointers allow us to refer to the *address* of a variable rather than its value. If this were all that were possible, we may be able to get away without using pointers at all. ‘‘Unfortunately’’ parameters to C’s functions may only be passed by value, and so a rudimentary understanding of C’s pointers is needed to use ‘‘pass-by-reference’’ parameter passing in C.

Consider the following example trying to interchange the value of two integer variables:

```
#include <stdio.h>

void swap(int i, int j)
{
    int temp;

    temp = i;
    i    = j;
    j    = temp;
}

int main()
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(a,b);
    printf("after  a=%d, b=%d\n",a,b);
    return(0);
}

before a=3, b=5
after  a=3, b=5
```

Pass By Reference Using Pointers

Instead, we need to pass a “reference” to the two integers to be interchanged, so that the function `swap()` is actually dealing with the original variables, rather than new copies of their values (passed on C’s run-time stack).

```
#include <stdio.h>

void swap(int *ip, int *jp)
{
    int temp;

    temp  = *ip;
    *ip   = *jp;
    *jp   = temp;
}

int main()
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(&a, &b);
    printf("after  a=%d, b=%d\n",a,b);
    return(0);
}

before a=3, b=5
after  a=5, b=3
```

Here we’ve introduced a bit more syntax (and, typically, it uses punctuation characters).

- The address operator, &, is used to determine the (run-time) memory address of a variable. Here we require the memory address of the variables `i` and `j` before passing these addresses to the `swap()` function. Notice that we are still using pass-by-value parameter passing, but that we are passing addresses on the run-time stack.

- The two asterisks in `swap()`’s formal definition indicate that the variables `ip` and `jp` are *pointers*, or *pointer variables*, rather than just “simple” variables. It is typical in C programs to append ’p’ or ’ptr’ to a variable’s name to indicate that it’s a pointer.

- The asterisks always placed in front of `ip` and `jp` in function `swap()` indicate that we wish to *dereference* these variables. Instead of using the contents of these variables (which are “meaningless” memory addresses) we wish to use the values *pointed to* by these variables. Notice that we may dereference variables on “both sides” of an assignment expression.

Pointers To Arrays And Character Strings

One often confusing point in C is the synonymous use of arrays, character strings, and pointers. The name of an array in C, is actually the memory address of the array’s first element. Thus the following two assignment statements are the same, and the first is the most commonly used:

```
char   buffer[BUFSIZ], *ptr;

ptr  = buffer;
ptr  = &buffer[0];
```

Using the *cdecl* program again:

```
-> explain char *ptr
   declare ptr as pointer to char
```

If we also remember that C’s character strings are simply a contiguous series of characters which, by convention, are terminated by a NULL character, then we can consider strings to be arrays to, and strings may be accessed through pointers (you may wish to consider a string’s first character as being stored at the memory address of the array of characters. We can thus write:

```
    int   n;
    char  *hex_values = "0123456789abcdef";

    n = hex_values[ ...expression... ];

/* or ... don’t do this! */

    n = "0123456789abcdef"[ ...expression... ];
```

We will often see the use of character pointers (used to strings), and character arrays (with assumed terminating NULL characters, used interchangeably:

```
int my_strlen(char *str)
{
    int   len = 0;

    while( str[len] /* != '\0' */ )
        ++len;
    return(len);
```

Pointer Arithmetic

Another confusing facility in C is the use of *pointer arithmetic* with which we may advance a pointer to point to successive memory locations at run-time. It would make little sense to be able to "point anywhere" into memory, and so C automatically adjusts pointers (forwards and backwards) by values that are multiples of the size of the base types (or user-defined structures) to which the pointer points(!).

We specify pointer arithmetic in the same way we specify numeric arithmetic, using +, −, and pre- and post- increment and decrement operators (multiplication and division make little sense). We may thus traverse an array with pointer arithmetic:

```
int my_strlen(char *str)
{
    int   len = 0;

    while( *str /* != '\0' */ ) {
        ++len;
        ++str;
    }
    return(len);
}
```

Notice that we are simply "moving the pointer along", we are not modifying what it pointers to, simply accessing adjacent memory locations until we reach one containing the NULL character. This example is a little simple, because the character pointer will only be advanced one memory location (one byte) at a time, as a character is one byte long. Alternatively, consider the five equivalent examples:

```
int sum_array(int *values, int n)
{
    int   i, *ip;
    int   sum = 0;

    for(sum=0, i=0 ; i<n ; ++i)
        sum += values[i];

    for(sum=0, i=0 ; i<n ; ++i)
        sum += *(values+i);

    for(sum=0, ip=values; ip<&values[n] ; ++ip)
        sum += *ip;

    for(sum=0, i=0 ; i<n ; ++i) {
        sum += *values;
        ++values;
    }
    for(sum=0, i=0 ; i<n ; ++i)
        sum += *values++;
    return(sum);
}
```


Pointer Arithmetic, *continued*

Unfortunately, we frequently see an excessive use of pointer arithmetic in C with programmers trying to be too smart to speed up their programs. For example:

```
char *my_strcpy(char *dest, char *src)
{
    char   *d = dest;

    while(*dest++ = *src++ );
    return(d);
}
```

With code such as this, in which we are trying to copy all characters from `src` to `dest` until we reach the NULL character, we always have in the back of our minds the concern as to whether the NULL character is in fact copied from the end of `src` to `dest`, and thus legally terminates `dest`.

Sorting An Array Of Values

A frequently required operation is to sort an array of, say, integers or characters. The standard C library provides a generic function named `qsort()` to help with this, but we must write a pointer-based function to perform the comparison of the array's elements:

```
#include <stdlib.h>

#define  N    100

int compare(const int *ip, const int *jp)
{
    return(*i - *j);
}

int main()
{
    int i;
    int values[N];

    srandom( getpid() );
    for(i=0 ; i<N ; i++)
        values[i] = random();
    qsort((void *)values, (size_t)N,
                 sizeof(values[0]), compare);
     ....
    return(0);
}
```