Introduction to C

DISCLAIMER: This document leaves out *lots* of stuff. If you want to learn C properly, read Kernighan & Ritchie (they created the language) or Prata ("C Primer Plus"). There is also an excellent online C tutorial[1] that can get you started from scratch very quickly. Apart from some "do"s and "don't"s in the next section, this document deals only with certain specific and common tasks that might not be covered explicitly in a reference manual.

Basic Rules

C can be as structured or as unstructured as you like, but it's best to follow a few simple rules:

1. Stick to one statement per line.
2. Indent appropriately to show nested logic.
3. Use descriptive variable and function names.
4. Break large tasks into smaller tasks using functions.
5. Use comments (delimited by `/*` and `*/`) effectively.
6. Avoid excessive use of global variables (in fact, avoid them altogether if possible!).
7. Always use `assert()`s or some other kind of error checking.
8. Reduce argument lists by using suitable `structs` where appropriate.
9. Define obscure numerical constants at the beginning of a source file with `#define` macros or `const` statements; this makes the code easier to understand and change later on.
10. Consider using `enums` to give descriptive names to simple integer flag values.

Note that in C all variables and functions must be defined before they are used. This is a *good* thing.

Compiling

Most platforms these days have native C compilers (usually called `cc`) as well as one or more third-party C compilers, the most common being `gcc`, the Gnu C compiler. Native compilers differ in their argument syntax; check the man pages if in doubt. Compiling can be as simple as `cc myprog.c` (which produces an executable version called `a.out`) or more complex like `cc -O2 -o myprog myprog.c -lm` (which names the executable `myprog`, applies second-level optimization, and links to the math library). If an error is encountered during the compilation, a message will appear indicating the nature of the problem and the offending line number in the code.

Hint: Always fix the first reported error first! Often one error (e.g. forgetting a semi-colon or closing brace) generates many more, making it seem like the code is full of problems when in fact there is just one simple typo.

Libraries and Header Files

To keep C as portable as possible many machine-specific tasks are kept in separate libraries. To use these library functions you must first include an appropriate header file that contains various declarations and definitions. For example, to use `printf()` you must `#include <stdio.h>` at the top of the source file. Other common header files include `<stdlib.h>` (for less-used functions like `malloc()` and `rand()`) and `<math.h>` (for certain math functions, like `sqrt()` and `pow()`). In some instances you must also "link" to the corresponding library when compiling. The most common library needed is the math library, which is linked by appending "`-lm`" to the compile line (e.g. `gcc myprog.c -lm`). Most C library functions have

[1] http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html

corresponding Unix man pages with synopses of header file and linkage requirements. Note that the order in which libraries are linked is sometimes important; it's usually a good idea to put "`-lm`" last, since other libraries may depend on the math library as well.

Handy Macro: `assert()`

The `assert()` macro is convenient for debugging or bare-bones error trapping. If the argument to `assert()` is an expression that evaluates "true" (non zero) then the statement has no effect. Otherwise execution is halted with a "failed assert" message along with a line number indicating where in the source file the failure occurred. To use `assert()`, simply `#include <assert.h>` at the beginning of the program. To disable `assert()`s without physically removing them from the code, compile with the preprocessor option `-DNDEBUG` or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement.

Arrays, Pointers, and `structs`

Arrays

In C, you can make an array (e.g., of integers) in one of two ways: with a declaration at the start of the function like `int myarray[N];`, where `N` is the number of elements in the array, or by declaring a pointer `int *myarray;` and calling `malloc()` to allocate `N ints` to it (see section). In either case, you reference element `n` in the array as `myarray[n]`, noting that array indices start at 0 in C, not 1. Regardless of how you declared the array, the name of the array (in this case `myarray`) is always a pointer to the first element, and can be passed as a pointer to another function. This means a function that is passed an array can modify the contents of the array, since it knows exactly where in memory the data is stored. As an example,

```
void init_array(int *myarray)
{
   myarray[0] = 12;
}

void main(void)
{
   int myarray[15];
   init_array(myarray);
}
```

declares an array and calls a function to initialize the first element of it. Note that `myfunc()` does not know the *size* of the array; usually you would pass this information as well, otherwise you risk accessing the array out of bounds.

Pointers

Pointers are very powerful but also very dangerous. As you can see from the previous example, there's nothing stopping `myfunc()` from trying to write to element 99 of `myarray`. Doing so would at best result in a "segmentation fault" error. At worst, your program could be corrupted in unpredictable ways as it's running, leading to nonsensical results. To avoid the headache of tracking down seg faults, learn to be paranoid about pointers. Here's a much more careful version of the previous program:

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

void init_array(int *myarray,int nelem,int elem,int val)
{
    assert(myarray != NULL);
    assert(elem >= 0 && elem < nelem);
    myarray[elem] = val;
}

#define MAX_NELEM 15

void main(void)
{
    int myarray[MAX_NELEM];
    init_array(myarray,MAX_NELEM,0,12);
}
```

Here we used `assert()`s to ensure first that the pointer has a non-zero value and second that no attempt is made to access the array out of bounds.

Generally pointers are used to allow a function to modify the contents of a memory address. This can make for very elegant code, and indeed is the only way for a function to change the value of a passed scalar, but it pays to be careful when using them.

`structs`

Structures (or `structs`) are a very convenient way of collecting disparate data types into a single data element. The following example shows how a `struct` is defined as a type, how a variable of that type is declared, and how to access data fields in the `struct`:

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

struct dates_s {
    int month;
    int day;
    int year;
};

void showdate(struct date_s *date)
{
    assert(date != NULL);
    (void) printf("Date = %02i/%02i/%4i\n",date->month,date->day,date->year);
}

void main(void)
{
    struct date_s date;

    date.month = 1;
    date.day = 27;
    date.year = 2004;
    showdate(&date);
}
```

Note that, unlike for arrays, the name of a structure is *not* a pointer to the structure. This means you can pass the actual value (contents) of the structure to a function. However, it's generally more efficient to just pass a pointer, as in the example above, to avoid copying data unnecessarily.

Passing Arguments to `main()`

`main()` is passed two arguments from the shell: an integer and a pointer to an array of strings. Traditionally these are declared as follows:

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

Here `argc` ("argument count") contains one plus the number of arguments passed to the program from the command line and `argv` ("argument vector") contains a series of `char` pointers to these arguments. The first element in `argv` is always the name of the program itself, so `argc` is always at least 1. The library function `getopt()` can perform simple parsing of command-line arguments; see the listing in section 3c of the man pages. Here's a more simple example of passing two numbers and a string. Note the error trapping to force the user to conform to the expected usage:

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

int main(int argc,char *argv[]) {
    int m,n;
    if (argc != 4) {
        printf("Usage: %s m n filename\n",argv[0]);
        return 1;
    }
    m = atoi(argv[1]); /* convert strings to integers */
    n = atoi(argv[2]);
    printf("%s received m=%i n=%i filename=%s\n",argv[0],m,n,argv[3]);
    return 0;
}
```

Recursion

C has the interesting property that functions can be called recursively, that is, a function can call itself. Often this can make for elegant, but not necessarily efficient code. Recursive functions add to memory overhead and are not easily optimized when compiling. However, there are times when recursive calls are a natural choice, for example when handling tree-like data structures. Here's a simple example of a recursive function to compute the factorial of a number:

```
long int factorial(int n)
{
    assert(n >= 0); /* negative values not allowed */
    if (n < 2) return 1; /* this is the exit condition */
    return n*factorial(n - 1);
}
```

This could equally well be written:

```
long int factorial(int n)
{
    long int f;
    assert(n >= 0);
    for (f=n;n>1;n--)
        f *= (n - 1);
    return f;
}
```

Which version to use is mostly a matter of personal taste. Note that factorials get large very quickly, so even a modest value of `n` can result in a factorial that exceeds `long int` capacity.

Variable-size arrays

The library function `malloc()` can be used to allocate memory dynamically. The function `free()` returns the memory to the pool when you're done with it. For example, to allocate an array of n `doubles`:

```
double *myArray;
myArray = (double *) malloc(n*sizeof(double));
assert(myArray != NULL);
DoSomethingWith(myArray);
free((void *) myArray);
```

The return value of `malloc()` is a pointer to `void` (i.e., a so-called "generic" pointer) so it must be cast to the appropriate type, in this case pointer to `double`. Similarly, the pointer must be recast to `void *` for freeing. If `malloc()` fails to allocate the memory (for example, if there's none left!), `NULL` is returned, hence the assert. Bad things happen when you operate on a `NULL` pointer...

Be sure to `#include <stdlib.h>` when using `malloc()`.

Variable-size multi-dimensional arrays

One way to allocate dynamically a m by n array (of `doubles`, say) is to first allocate space for m pointers to `double` and then allocate n `doubles` for each of these:

```
double **myArray;
int i;
myArray = (double **) malloc(m*sizeof(double *));
assert(myArray);
for (i=0;i<m;i++) {
    myArray[i] = (double *) malloc(n*sizeof(double));
    assert(myArray[i]);
}
DoSomethingWith(myArray);
for (i=0;i<m;i++)
    free((void *) myArray[i]);
free((void *) myArray);
```

This naturally can be extended to any number of dimensions. Perhaps a better way is to simply allocate a single m by n block of memory of the appropriate storage type and access the elements using row-column formulae like $i * m + j$, where i runs from 0 to $m - 1$ and j runs from 0 to $n - 1$.

Random Numbers

There are several random number generators available in the C library. For modern implementations, the recommended one is `int rand(void);`, which returns a random integer between 0 and `RAND_MAX` (`#include <stdlib.h>`). Use `void srand(unsigned int seed);` to seed the generator. Ideally you want a different seed for each invocation of the program. One way to do this is to use `srand(getpid());` where `getpid()` returns the ID number of the current process (`#include <unistd.h>`). A better seed can be obtained by combining `getpid()` with the Unix time (`#include <time.h>`—see next section) and the parent ID: `(int) time(NULL) % getpid() + getppid()`.

Timers

The library calls `gettimeofday()`, `time()`, `clock()`, `times()`, and `getrusage()` can all be used to measure execution times. The best resolution is obtained from `gettimeofday()` (for wallclock) and `getrusage()` (for user and system). Both these functions take structure pointers as arguments and fill the elements with timer values. The best strategy for all such timers is to *difference* the before and after values. Check the man pages for more info. Note the crucial difference between wallclock and user/system time is that the former will increase as the load on the machine increases due to other processes whereas the latter will not. Here's a simple example using `clock()`, which returns the amount of CPU time used so far:

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

double getCPU(void)
{
    return (double) clock()/CLOCKS_PER_SEC; /* convert to seconds */
}

int main(void)
{
    double t0,dt;
    t0 = getCPU();         /* get initial value */
    MyExpensiveFunction();
    dt = getCPU() - t0;    /* difference to get execution time */
    (void) printf("Execution time: %g sec\n",dt);
    return 0;
}
```