Introduction to C

Source: Modified from lecture notes produced by Ian Cooke at University of Chicago.

The good news is that C syntax is almost identical to that of C++. However, there are many things you are used to that aren't available in C:

Not Available in C++:	***Do this instead:***
classes and all their associated tricks: templates, inheritance, etc.	structs (are much more limited)
`new` and `delete`	`malloc` and `free`
the stream operators << >>	printf and scanf
the bool keyword	integers (zero is false, nonzero is true)
C++ standard libraries (iostream, string)	C standard library (stdio, string)
reference parameters	pass pointers to variables

This is intended to be a very brief introduction to C. More information can be found on the man pages, the Internet, or from one of the myriad of C books available.

Compiler

The C compiler is '`gcc`'. The operation is the same as '`g++`'.

Strings

In C, strings are represented by an array of characters. The end of the string is terminated by a null character '`\0`'. This is necessary even if you are using all of the characters in the array.

Unlike C++, the user must make sure there is sufficient space in the array to store the string. Strings do not grow automatically. Overflowing the array can lead to disastrous results and hard-to-find bugs. Arrays can either be declared statically (such as "`char my_string[100];`") or be dynamically allocated (see section on Memory Allocation).

A number of functions are provided to manipulate strings and are described in the quick reference guide. These functions are unsafe. Care is necessary to make sure strings are null-terminated and that arrays have adequate space to store the string.

I/O

C doesn't have stream operators. Instead you'll want to use the functions provided in the stdio library. That means you'll need to include the library:

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

Output: printf, fprintf

The most common output function in C is `printf()` which prints characters to the screen (or wherever standard out is directed to go). Here's a quick hello world program that illustrates its use:

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

The character '`\n`' is called a newline character. It will cause a line feed in a similar fashion to `endl` in C++.

`printf` has a variable number of arguments - the first of which is a format string. The format string can contain both ordinary characters (what you want to appear on the screen like 'Hello World' above) and conversion character sequences. You can think of conversion characters as placeholders for a specific type formatted in a particular way. Conversion character sequences begin with % and end with a conversion character. An example will perhaps make this clearer. Here we're printing out the integers from 0 to 9.

```
int i;
for (i = 0; i < 10; i++) {
  printf("the variable 'i' is: %d\n", i);
}
```

The conversion sequence is `%d`, which you can think of as a placeholder for an integer. Since this is the first conversion character sequence, the value of the first argument after the format string-- in this case the value of '`i`'-- will be placed in the held spot. Conversion characters can specify type, precision, and all sorts of other formatting information. See the man pages for all the gory details, but here are essential ones:

`%d` - prints in integer in decimal notation
`%x` - prints an integer in hexadecimal notation
`%s` - prints a string (until the null character is reached)
`%c` - prints a character
`%f` - prints a floating point character
`%%` - prints a single percent sign '%'

`printf` always prints to standard output. If you want to print to another place, such as standard error, use `fprintf` which takes a file pointer as its first argument:

```
fprintf(stderr, "Fatal Error #2212. We're hosed");
```

The first argument to `fprintf` is actually a pointer to the C-file type (`FILE *`), meaning that `fprintf` can be used to output data to files. There are pre-defined file pointers for standard input, output, and error:

C++	C
cin	stdin
cout	stdout
cerr	stderr

Input: scanf, fscanf

Input is done using `scanf`. It works in a similar way as `printf`:

```
int scanf(char *format, ...);
```

The format string consists of conversion sequences. Again, there is an argument for each conversion sequence. Unlike `printf`, the arguments are pointers to variables where you want the input data to be stored. For instance, this code snippet stores integers in the variables a and b.

```
int a, b, c;
int *p = &a;
p = &a;
c = scanf(“%d %d”, p, &b);
```

The return value of `scanf` is the number of successful values inputted. It is a good practice to check the return value from `scanf` to make sure you have the proper data.

Reading in strings using “`%s`” requires some care. A maximum field width should be specified to avoid overflowing an array. This is done as follows:

```
char name[9];
int c;
c = scanf(“%8s”, name);
```

This will limit the number of characters read from input. In this case, eight characters are read. However, the array holds nine characters because `scanf` always adds an extra null character to the end of the string.

The function `fscanf` is identical to `scanf` except it has an additional argument for a file.

Consult the quick reference guide for more input/output functions.

Memory Allocation

All memory allocation is done with the function `malloc`:

```
void * malloc(int nbytes);
```

The lone argument to `malloc` is the number of bytes you want allocated and it returns a void pointer to the start of the allocated memory. The return value of `malloc` is typically casted to the pointer to the proper type:

```
int *p, *q;
struct foo *r;
p = (int *) malloc(sizeof(int));                 // single integer
q = (int *) malloc(sizeof(int) * 10);            // array of 10 ints
r = (struct foo *) malloc(sizeof(struct foo));   // single foo object
```

Freeing memory is done with the function `free`:

```
void free(void *);
```

Example: `free(p);`

Variable Declarations

In C++ you can declare variables pretty much anywhere in your program. This is not the case with C. Variables must be declared at the beginning of a function (or immediately after any '{') and must be declared before any other code. This *includes* loop counter variables, which means you can't do this:

```
for(int i = 0; i < 200; i++) {
```

When declaring variables of `struct` or `enum` types, the word `struct` or `enum` must appear in the declaration:

```
struct foo a;    // OK
foo a;           // Error
```

Constants

The standard way to declare a constant in C is to use the `#define` preprocessor directive:

```
#define MAX_LEN 1024
```

This is also valid C++ code, though the use of `#define` is usually discouraged in C++.