# Lecture 6: The basics of C programming

## Basic structure of a C program

As we saw in the `hello.c` example in the last lecture, Every C program must contain a `main` function , since the `main` function is the first function called when you run your program at the command line. In its simplest form, a C program is given by

```
int main(void) {
  printf(``Hello world!\n'');
}
```

The `int` stands for the "return type", which says that the main function returns an integer if you tell it to do so. We will get into more detail with functions and return types, but you can return a number to the command line with the `return` function in C, as in

```
int main(void) {
  printf(``Hello world!\n'');
  return 2;
}
```

When you compile and run your program, it will return a 2 to the command line, that you can access with the `$?` character, which stores the value returned by the last command executed, as in

```
$ ./a.out
Hello world!
$ echo $?
2
```

Just as in shell scripts, you can specify exit codes in C as well, which perform the same function as the `return` function in the previous example:

```
int main(void) {
  printf(``Hello world!\n'');
  exit(2);
}
```

Compiling and running this example yields the same result as the previous example. The `void` statement in `main(void)` tells the main function that there are no arguments being supplied to it at the command line. We will get into this in more detail when we look at functions.

Note that all statements in C programs end with the semicolon ; except for the preprocessor directives that begin with `#`.

# Variables

## Standard variable types

All variables in C must be defined before they are used. Variables in C are defined by their *type*, which determines how much space they use in memory. The more memory a variable requires, the larger the number it can store, and therefore the more precision you can obtain. For example, to define an integer `i` , a floating point number `x` , and a character `c` in our simple example, we would use

```
int main(void) {
  int i;
  float x;
  char c;
  printf(``Hello world!\n'');
  exit(2);
}
```

The `int` type stores 4-byte integers. Since 1 byte contains 8 bits, then each integer type can store 32 bits of information. If the first bit stores the sign, then the last 31 can store a binary number. The largest binary number that 31 bits can store is $2^{31} - 1$. We can store a larger (absolute value) negative number because we can assume that if the first bit is negative and the rest are 0 then this represents the number -1, and not 0, which is represented by the first bit being positive and the rest being 0. In C, we can specify the `unsigned` type, which doubles the size of the number because we do not need to store a bit for the sign of the number. We can also specify the `short` type, which is just a smaller version of the type `int`. The following table lists the available types in Unix and their ranges [1]

| Type | Space (bytes) | Range (min,max) |
|---|---|---|
| `char` | 1 | -,- |
| `unsigned char` | 1 | $0,2^8 - 1$ |
| `short int` | 2 | $-2^{15},2^{15} - 1$ |
| `unsigned short int` | 2 | $0,2^{16} - 1$ |
| `int` | 4 | $-2^{31}, 2^{31} - 1$ |
| `float` | 4 | $\pm 3.2 \times 10^{\pm 38}$ |
| `double` | 8 | $\pm 1.7 \times 10^{\pm 308}$ |

You can define variables sequentially in a list or separately, line by line, as in

```
int i, j, k;
float x, y, z;
```

You can initialize variables with variables at the same time you declare them, such as

```
int i=0,j=1,k=1000;
float x=0,y=2,z=3;
```

## Global variables

Global variables are defined at the top of your C codes before the `main` statement. They are recognized everywhere in the code and not only in the function in which they are defined. The following C code defines the variables `globx` and `globy` as global variables before the `main` statement:

```
float globx, globy;
int main(void) {
  ...
}
```

It is normally not good practice to declare global variables in C programs. An alternative method is to employ the preprocessor directives to define variables that you will need throughout your programs. For example, to define the above variables as global variables, you should define them using the preprocessor directives with

```
#define GLOBX 10
#define GLOBY 20
int main(void) {
  ...
}
```

Note the standard practice of defining these global variables in caps. Remember, the two methods are very different. The first method defines and stores the variables in memory and they can be changed in your program. The second method uses the C preprocessor to replace all instances of `GLOBX` with `10` and `GLOBY` with 20 *before* any compilation takes place.

## Defining your own types

C allows you to define your own types, and this can make your codes much more legible. For example, you can define a type called `price` that you can use to define variables that you use to represent the price of certain objects. To do so, you would use

```
typedef float price;
int main(void) {
  price x, y;
}
```

The `typedef` is used in the following way:

```
typedef existing_type new_type;
```

The `price` type example is identical to using

```
float x, y;
```

but it is useful to use `typedef` when you would like the ability to change the types of certain variables throughout your codes by only changing one line. That is, if you wanted to change all of your prices to type `double`, then you would only need to change the `typedef` line to

```
typedef double price;
```

### Constant types

You can declare types that you do not want to be altered in your programs with the `const` declaration. For example,

```
const int constant_integer=2;
```

declares the value of `constant_integer` to be 2 and this value cannot be changed in any part of the code. It is effectively "read-only" and any attempts to change it will result in a compiler warning such as

```
warning: assignment of read-only variable ‘constant_integer’
```

The value can ultimately be changed but the `const` declaration causes the compiler warning when you do change it. You use the `const` declaration mostly to let users of your functions know that the variables will not be altered.

## Formatted output

To print output to the screen, you use the `printf` statement. To print out a string to the screen, you would use

```
printf(‘‘This is a string.\n’’);
```

Any time the \ character is encountered, it is evaluated by `printf` as a formatting character. In this case, the `\n` character prints a carriage return to the screen. Another commonly used formatting character is the `\t` character, which prints a tab to the screen, as in

```
printf(‘‘This string is separated\tfrom this one by a tab.\n’’);
```

Note that use of the \ evaluates the first character after it to determine the format, and the next character is printed, so you do not need a space after the `\t` in this example. To print the actual \ character, you would use two backslashes, as in

```
printf(‘‘This is a backslash: \\\n’’);
```

To print out the values of variables, you use the `%` character. As an example,

```
float x=5.2543;
printf(‘‘The value is %f\n’’,x);
```

will print out `The value is 5.254300`. The default of `%f` is to print out floating point numbers with six digits. You can specify how many digits you would like to print with

```
printf(‘‘The value is %.2f\n’’,x);
```

which will print out two digits with `The value is 5.25`. If you print out several numbers, then if they have a different number of digits before the decimal point, then the numbers may be misaligned, as in

```
double x=5.2543, y=6558.2391;
printf("The value of x is %.2f\n",x);
printf("The value of y is %.2f\n",y);
```

which will print out

```
The value of x is 5.25
The value of y is 6558.24
```

You can align the numbers by specifying how many digits to print in the format, which includes the decimal point and the digits after it. Since the number `6558.24` has 7 digits, we can format the printout so that it prints at least 7 digits, as in

```
double x=5.2543, y=6558.2391;
printf("The value of x is %7.2f\n",x);
printf("The value of y is %7.2f\n",y);
```

which will align the decimals when the numbers are printed, as in

```
The value of x is    5.25
The value of y is 6558.24
```

Note that the format statement will *always* print out all of the digits before the decimal point as well as the number you specify after it. So if you specify `%4.2f` to print out `y` in the above example, you will still get `6558.24`. This is because the number of digits you specify is just a minimum. Other formatting statements include `%d`, which is used to print decimal integers , such as

```
int i=5, j=6558;
printf("The value of i is %4d\n",i);
printf("The value of j is %4d\n",j);
```

which will print out

```
The value of i is    5
The value of j is 6558
```

You can also print characters with `%c` , strings with `%s`, octal numbers with `%o`, and hexadecimal numbers with `%x`.

## Formatted input

Formatted input is performed with the `scanf` function. For example, to read two floating point numbers from the command line, you would use

```
scanf(‘‘%f %f’’,&x,&y);
```

This would convert input of the form `1 2` appropriately, even though this input is not necessarily floating point input. To read in two integers, you would use

```
scanf(‘‘%d %d’’,&i,&j);
```

This would work fine with input that does not have decimal points, but if you try to read in floating point numbers as integers, you will not achieve the desired result. The `&` sign stands for the address of the variables. We will go over this when we discuss pointers.

## File I/O

Input and output to and from files is identical to that at the command line, except the `fprintf` and `fscanf` functions are used and they require another argument. This additional argument is called a file pointer. In order to write two floating point numbers to a file, you first need to declar the file pointer with the `FILE` type, and you need to open it, as in

```
float x=1, y=2;
FILE *file;

file = fopen(``file.txt'',''w'');
fprintf(file,''%f %f\n'',x,y);
fclose(file);
```

The function `fprintf` is identical to the `printf` function, except now we see it has another argument `file`, which is a pointer to the file. Before you use the `file` variable, you need to open the file with

```
file = fopen(``file.txt'',''w'');
```

This opens up the file ``` ``file.txt'' ``` and the ``` ``w'' ``` which is the mode and indicates how the file will be used. The following three modes are allowed:

| Mode | String |
|---|---|
| Open for reading | "r" |
| Open for writing | "w" |
| Open and append | "a" |

When you are done with the file, you close it with

```
fclose(file);
```

The ``` ``r'' ``` mode is used when you would like to open a file for reading. To read two floating point numbers from a file, you would use the `fscanf` function, which is identical to `scanf`, except that it takes the file pointer as its first argument, as in

```
float x, y;
FILE *file;

file = fopen(``file.txt'',''r'');
fscanf(file,''%f %f\n'',&x,&y);
fclose(file);
```

You can check to make sure your files are opened correctly (that is, that they exist), by checking to make sure `file` is not the predefined `NULL` pointer. We'll discuss this pointer in more detail later, but for now, to ensure that your file was opened correctly, you use

```
if(!file) printf(``File did not open correctly!\n'');
```

Note that in order to use the `FILE` type, we need to include the standard C header file with

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

The `fscanf` and `fprintf` functions can be used to print to and read from the terminal using the `stdin`, `stdout`,and `stderr` file pointers defined in `stdio.h`. To write to the standard output of the terminal using `fprintf`, then, you would use

```
fprintf(stdout,''%f %f\n'',x,y);
```

which is identical to using

```
printf(''%f %f\n'',x,y);
```

To write to the standard error, you would use

```
fprintf(stderr,''%f %f\n'',x,y);
```

To read from the terminal's standard input with `fscanf`, you would use

```
fscanf(stdin,''%f %f'',&x,&y);
```

which is identical to using

```
scanf(''%f %f'',&x,&y);
```

Again, in order for the compiler to know what `stdin`, `stdout`, and `stderr` are, you need to include `stdio.h`.

## Operators

### Arithmetic Operators

The arithmetic operators are

```
*,/,%,+,-
```

The modulus operator `%` , when used as

```
z = x % y;
```

assigns `z` with the value of the remainder when `x` is divided by `y`. The modulus operator does not work with types other than `int`, as it will truncate the result. The `+` and `-` operators have the same precedence, which is lower than that of `*`, `/`, and `%`, and all arithmetic operators of equal precedence are evaluated from left to right. Therefore, the expression

```
2*z-x+y*z%2
```

is evaluated as

```
((2*z)-x)+((y*z)%2)
```

## Relational operators

The relational operators are given by

```
< > <= >=
```

and the equality operators are

```
==  !=
```

The relational operators exceed the equality operators in precedence, and the arithmetic operators exceed the relational operators.

## Logical operators

The logical operators are `&&` and `||`, which are not to be confused with the bitwise operators `&` and `|`. Logical operators are evaluated from left to right, until the evaluation is known to be either true or false, while the precedence of the `&&` operator is greater than that of the `||` operator. That is,

```
a==b && c==d || x==y && w==z
```

is evaluated as

```
(a==b && c==d) || (x==y && w==z)
```

As another example, the logical relation

```
4==2*2 && 3==5-1 && 2==1
```

will return false as soon as the `3==5-1` is encountered, since this means the entire statement is false, without having to evaluate the last false conditional `2==1`.

## Increment, decrement, and assignment operators

In C, the increment and decrement operators are `++` and `--`, so that in order to increment a variable `i` and decrement it you would use `i++` or `i--`. These operators are provided merely for source code compactness, since they are identical to `i=i+1` and `i=i-1`. A peculiar aspect to the increment or decrement operators is that they can be used as prefix operators (`++i`) as well as postfix operators (`i++`). The difference between the two is that `i++` returns the value of `i` *before* it is incremented, while `++i` increments `i` and then returns it. For example, in the following example,

```
i=2
j=i++
```

the value of `j` will be 2 while the value of `i` will be 3. If we use the prefix operator, as in

```
i=2
j=++i
```

the value of `j` will be 3 while the value of `i` will also be 3.

Assignment operators provide compact notation to assign an expression using a binary operator to a left-hand side argument. For example, `i=i+2` can be expressed using an assignment operator as `i+=2`. This works with the following operators

```
+ - * / & << >> & ^ |
```

As another examples,

```
x = x*(2+z)
```

can be expressed more compactly as

```
x*=2+z
```

Note that the argument on the right hand side of the assignment operator always takes precendence, in that `x*=2+z` **is not the same as** `x=x*2+z` but is given above by `x=x*(2+z)`.

## Precedence and order of operations

So far we have seen that certain operators take precendence over others and that determined the order of operations in which the operators were performed. It is important to understand that in C, every operator performs an operation on its arguments and then returns a result, and the calculation proceeds from there. If you think about things this way it makes it easier to understand the result from an operation which contains confusing precedence issues. For example, in the statement `i=i+1`, the only way we know what the result will be is by knowing that the `+` operator takes precendence over the `=` operator. So we write the above statement as `i=(i+1)`. We know that it is not `(i=i)+1` only because the `+` operator takes precendence over the `=` operator.

Another important aspect of operators is their associativity. That is, are the operations evaluated from right to left or from left to right? Associativity determines how things are evaluated by the way you would place parentheses around the operators. For example, since `+` is performed from left to right, we would evaluate a sum of several operands as

```
i+j+k+l+m=((((i+j)+k)+l)+m)
```

This may seem obvious only because of the associative property of addition, in that addition is identical if it is evaluated from left to right or from right to left, since
`((((i+j)+k)+l)+m)=(i+(j+(k+(l+m))))`.
However, this does not hold true for division, since
`((((i/j)/k)/l)/m)`$\neq$ `(i/(j/(k/(l/m))))`.
We know that C performs the divisions as they are on the left because associativity of division is from left to right. The following table shows the associativity of the operators in order of precedence, for which the operators at the top have the highest precedence [2].

| Operator | Associativity |
|---|---|
| `() [] -> .` | left to right |
| `! ~ ++ -- + - * & (type) sizeof` | right to left |
| `* / %` | left to right |
| `+ -` | left to right |
| `<< >>` | left to right |
| `< <= > >=` | left to right |
| `== !=` | left to right |
| `&` | left to right |
| `^` | left to right |
| `\|` | left to right |
| `&&` | left to right |
| `\|\|` | left to right |
| `?:` | right to left |
| `= += -= *= /= %= &= ^= \|= <<= >>=` | right to left |
| `,` | left to right |

We will deal with the `->` and `.` operators when we discuss structs and pointers.

## Conditionals

### If and ?

The syntax of the `if` statement in C is given by

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

and for multiple `if` statements, the syntax is given by

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

where you don't necessarily need the last default `else` statement . As an example, we might use the `if` statement as

```
if ( U == 0 )
  x = y;
else
  x = z;
```

The `if` statement first evaluates the expression. The expression, like all expressions in C, returns a value, which is either true (`1`) or false (`0`). Therefore, the expression `U == 0` returns a `0` when `U` is `0`. Rather than evaluating this expression, we can more compactly use

```
if ( U )
  x = y;
else
  x = z;
```

which will execute the `if ( U )` when `U` is nonzero. We can also use `if ( !U )` to represent `if ( U != 0 )` as well. Note that multiple expressions within the `statement` must be placed within braces (`{}`) to represent a block. For example,

```
if ( U ) {
  x = y;
  z = 1;
} else {
  x = z;
  y = 1;
}
```

A more compact form of the `if` statement involves using the `?` operator. For example, the following `if`statement

```
if ( y > z )
  x = y;
else
  x = z;
```

can be represented more compactly with

```
x = (y > z) ? y : z ;
```

The syntax is given by

```
expression1 ? expression2 : expression3
```

Which says evaluate `expression1` first. If it is true, then evaluate and return `expression2`, otherwise evaluate and return `expression3`.

## Switch

Often you may have a particular variable whose value may take on several possibilities. Rather than using an `if` statement, it is more convenient to use the `switch`statement, for which the syntax is given by

```
switch ( expression ) {
  case const-expr:
    statement
  case const-expr:
    statement
  default:
    statement
```

It is not necessary to employ the `default` statement, since if none of the cases are satisfied then the statements will execute. As an example, consider the following `switch` statement

```
int i=25;
switch ( i ) {
case 25:
  printf("i=25\n");
case 15:
  printf("i=15\n");
}
```

This `switch` statement tests the value of `i`. If the first `case` is true, or when `i==25`, then the first `printf` statement will be evaluated. But now that the first case has been satisfied, *all* subsequent statements will be evaluated as well. That is, after one case is satisfied, evaluation falls into the next case unless a break statement is issued. In the example given above, since `i==25`, this code will print out *both* `i=25` and `i=15`. To prevent this, you need to use the break statement, as in

```
int i=25;
switch ( i ) {
case 25:
  printf("i=25\n");
  break;
case 15:
  printf("i=15\n");
  break;
}
```

This will cause execution to break out of the `switch` statement after one of the cases is satisfied.

## For and while loops

The syntax of a `while` loop is given by

```
while ( expression )
  statement
```

This will repeatedly evaluate `statement` as long as `expression` is true. For loops are just special `while`loops. The syntax of a `for` loop is given by

```
for ( expression1 ; expression2 ; expression3 )
  statement
```

The equivalent `while` loop is given by

```
expression1;
while ( expression2 ) {
  statement
  expression3;
}
```

As an example, consider the `for` loop which loops through the values of `i` and prints the values

```
for(i=0;i<10;i++) {
  printf(``i=%d\n'',i);
}
```

The equivalent `while` loop is

```
i=0;
while ( i < 10 ) {
  printf(``i=%d\n'',i);
  i++;
}
```

Use of `for` or `while` for looping is purely a matter of preference, although some situations are better suited to using `for` loops. The above example was clearly suited to using a `for` loop since it looped through a specified set of values. At any time during the execution of the `for` or `while` loops, you can stop the loop with the `break` command.

## References

[1] D. Marshall. Programming in C, 1999. Online course notes. http://www.cs.cf.ac.uk/Dave/C/CE.html.

[2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.