CSC A48 – Unit 1 – Introduction to programming in C

1.- Our programming language

For this course, the programming language is ***C***.

The ***C***[1] programming language was developed in the early 1970's by Dennis M. Ritchie. Currently, the language is extensively used for applications ranging from operating systems, to sound, image, and video processing, to embedded systems software.

Its extensive use has made it, consistently, one of the most important languages for software development.

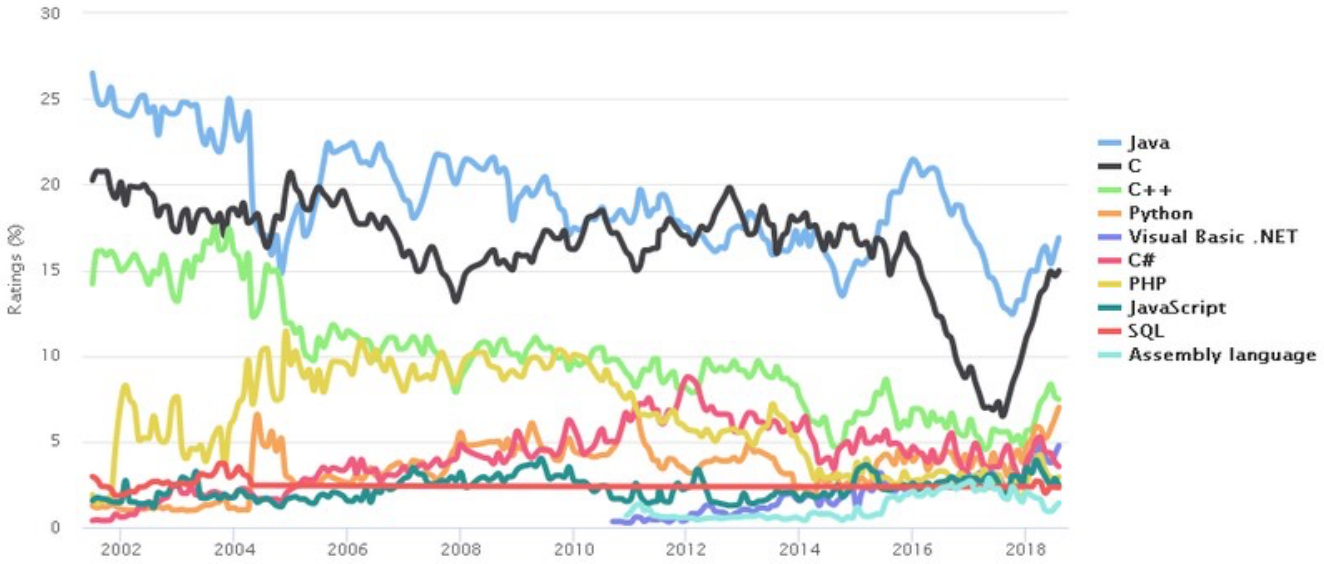


Illustration 1: Popularity of programming languages by the TIOBE Index (used with permission)

We will use ***C*** as our programming language for several important reasons:

- It is an ***imperative***[2] language, so the structure of a program, and the types of control structures (such as conditional statements, loops, etc.) will be familiar to you from what you learned of Python in your A08 course.

- It is a simple language, it provides precisely the support we need to focus on the concepts and algorithms that comprise our course, without having to become distracted by the richness of features, syntax, and programming constructs available in more recent languages.

- It will require you to know ***in detail*** what each line of your program is doing. Programming in ***C*** will help you strengthen your ability to fully understand what your code is doing at each step.

- It is a skill that will be required of you in future courses. Whether you are in CS, Math, Stats, or a different discipline altogether, the ability to think carefully and in detail about what your code is doing will help you make the most of courses where programming is needed to fully understand an exciting topic – a good example of this is Machine Learning (CSC C11), where implementing ML algorithms is essential for fully grasping the course material. If you are in CS, knowledge of C will be required for courses such as:
Operating Systems (C69), Embedded Systems (C85), Computer Graphics (D18), Computer Security (D27), Networks (D58), Programming on the Web (C09), Programming Languages (C24) and Compiler Optimization (D70).

- The memory model used by C is very simple, and accurately describes how computer storage works at a low level. It maps directly to how memory is accessed by the computer's processor, and how data and code are processed once a piece of code is translated to the CPU's machine language. Together with what you will learn in Computer Organization (B58), it will allow you to fully understand how a computer works, and how it can execute a program.

2.- Structure of a program in C

The basic structure of a C program is very similar to what you already know from Python:

- A program is split into ***functions***, each function carries out a specific task related to implementing a given algorithm.

- Unlike Python, there is *no interactive mode for C*. All programs start at a function called ***main()***. This function is in charge of using the rest of the code in the program to carry out the specified algorithm.

- The program can include and use code from ***libraries***, similar to Python modules, which provide a wide range of functionality.

- Code that belongs to a specific ***scope*** is indicated by ***delimiters***, in C these are curly braces {}.

- Comments are written as shown below:

```
/* A block of comments begins with a slash-star

        and can have as many lines as you need
        without any other symbol being required.

     The comments block ends with a star-slash      */

// Or we can use a double slash – for a 1-line comment!
// double-slash is technically not a part of standard C,
// but we need no worry about that for now. It works!
```

- Every variable in C has an associated data type. Just like Python, we can have different types that represent things such as ***integer numbers, floating point numbers, strings***, and so on. Unlike Python, once we have associated a data type with a specific variable, the type can not be changed.

The fundamental data types supported by C are

```
int                   (an integer number)
float/double          (floating point numbers)
char                  (one character)
void                  (no data type attached)
```

As you can see there is not many! Part of our course will consist of learning how to use these basic data types to build more complex containers for our information, so we can have things like the lists and dictionaries you have used in Python.

- At the coarsest level, a C program looks like this:

```
// At the top, we list the libraries our code will use.
// To import a library, we use the '#include<>' statement, similar
// to a Python import.

#include<stdlib.h>          // This is the standard C library
#include<stdio.h>           // This is the standard input/output library

// After the libraries, we will find the code that comprises the
// program, organized into functions.

f1(int x,int y,int z)    // A function definition, with parameters
{
      int w;     // Variable declarations for f1

      program statements;

      f2();// Calls to other functions
}

// Other functions (e.g. f2() and more) would be here

main()          // The main() function is where our program starts
{ // This curly brace indicates where the code for main begins

      // Variable declarations for main, e.g.
      int x;                    // An integer variable called x
      double y;                 // A floating point number called y
      char one_character;       // A single character
```

```
      program_statement_1;  // Every statement ends in ;
      program_statement_2;

      f1(1,2,3); // a sample function call!
                       // There will be more functions and code
                       // in a typical program!
} //  This curly brace indicates where the code for main ends
```

- ***Variables declared within a function are local to that function***, and can not be accessed anywhere else in the code. We will learn how to pass information between functions so we can get work done.

3.- The classic, first C program that everyone has to write at some point

Type the program below (or copy and paste it from here!) into a ***text file*** called ***hello_world.c***.

```
/*
      Hello World!
      Welcome to programming in C :)
*/

#include<stdio.h>

int main()
{
      printf("***** HELLO WORLD *****\n");
      return 0;
}
```

Notes:

The ***printf()*** function is part of the standard input/output library, and prints formatted information to the terminal. It has a large number of options that we will study later, but for now, suffice it to say that the string to be printed goes between the quotes, and the '\n' at the end is the newline character and tells the function that it should go to the next line after printing (without it, if you have another print statement, the output of the second one will be printed immediately after the first one, on the same line).

4.- Running your code!

Unlike Python, C doesn't have an interactive mode, and the text inside your program file can not be run as-is. Instead, we need to use a program called a *compiler* to generate an *executable* version of our program. The compiler's job is to read your code, check it for syntax, make sure the code follows all rules of the C programming language, and then translate the text in the program to *machine instructions in an executable program*. You can then *run the executable.*

The compiler will report any syntax or structural errors in your code (much like Python will report syntax problems when you load a program), but even if your program compiles without errors, it is possible to get *runtime errors* when you run the resulting executable program.

To compile your code:

- Open a terminal – depending on your version of Windows this will vary, but you can find out how by following the instructions at this link:

https://www.lifewire.com/how-to-open-command-prompt-2618089

- Using the '*cd*' command to change your directory to where you have your program, for help with that see here:

https://www.digitalcitizen.life/command-prompt-how-use-basic-commands

Pay attention when you save your program to the location where you saved it! You'll need to '*cd*' to that location.

- Once you have your terminal in the directory that contains your code (you can type 'dir' to list the files in that directory and check your program is there), you can call the compiler to create an executable for your program.

We will be using the ***gcc*** compiler (the name comes from ***G***NU ***C***ompiler ***C***ollection). It is a free, open-source compiler that supports C, C++, and other variants.

The compiler is already installed in all the Windows lab computer at UTSC. If you want to install it on your own Windows machine, you can do so by following this tutorial:

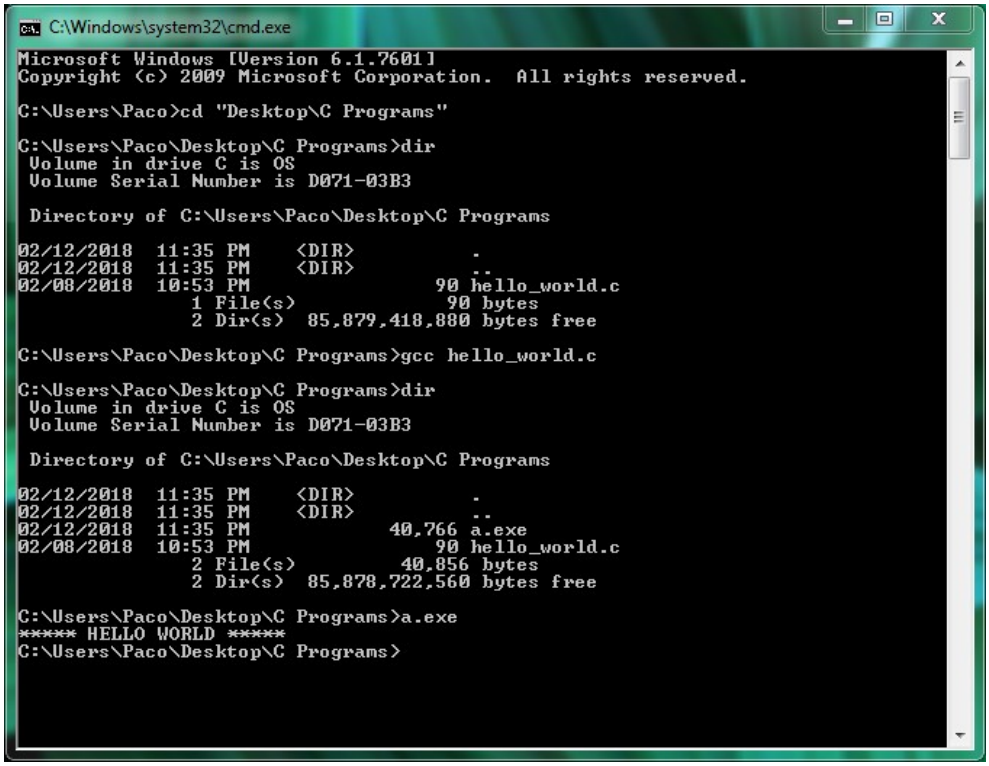
https://www.instructables.com/id/How-to-Install-MinGW-GCCG-Compiler-in-Windows-XP78/

Once you have ***gcc*** installed, you can compile your code by typing:

```
gcc hello_world.c
```

This will produce an executable file called '***a.exe***' which you can run from your terminal.

./a.out (if Mac)



By default, the compiler produces an executable called '***a.exe***' or ***'a.out'*** if you're using Linux. If you want to specify the name of your executable file, you can do so with the '-o' option:

```
gcc hello_world.c  -o hello_world
```

Which will produce 'hello_world.exe' on Windows, or simply 'hello_world' on Linux.

5 – Your first exercise!

Write your first C program, called ***my_hello_world.c*** so that when compiled and run it will tell us a bit about ***yourself***.

The output of your program should look like this:

```
Hello! My name is: Paco
I am a student in: Computer Science
My favourite colour is: Blue
My favourite fruit is: Pomegranate

I am going to learn C!  :)
```

Note that there is an empty line in the output.

Write, compile and run your code. Chances are, the first time you try this out you will run into

all kinds of compiler messages.

Here's the most important thing about learning to deal with compiler errors:

Do not stress out – they happen often and you can fix them.

Here are some typical problems you may run into:

```
#include<stdio.h>

int main()
{
     printf("***** HELLO WORLD *****\n")     // Missing semi-colon!
     return 0;
}
```

When we try to compile the above, we get:

```
\Desktop\C-programs\gcc error_0.c
error_0.c: In function ‘main’:
error_0.c:6:2: error: expected ‘;’ before ‘return’
  return 0;
  ^
```

The compiler is complaining about a missing semi-colon, just before the return statement on line 6 – be careful! The error is actually on the previous line! The printf() statement is missing the semi-colon.

Here's another example:

```
#include<stdio.h>

int main()
{
        printf('***** HELLO WORLD *****\n');    // Wrong quotes!
        return 0;
}
```

Now we're making the compiler really upset:

```
\Desktop\C-programs\gcc error_1.c
error_1.c: In function ‘main’:
error_1.c:5:9: warning: character constant too long for its type
  printf('***** HELLO WORLD *****\n'); // Wrong quotes
         ^
```

(there's a lot more messages after, but this is the first one that tells you where the problem is)

Note that the compiler helpfully gives you a '^' to indicate where in the code the problem appears to be. In C, text is delimited by double quotes, not single quotes.

Have a look at one more typical problem:

```
#include<stdio.h>

int main()
{
        printf("***** HELLO WORLD *****\n");
        return 0;
}

printf("This statement is outside any function!");
```

If you compile the above, you get:

```
\Desktop\C-programs\gcc error_2.c
error_2.c:9:8: error: expected declaration specifiers or ‘...’ before
string constant
 printf("This statement is outside any function!");
        ^
```

This tells us the compiler was expecting a function declaration (since the printf statement appears outside any function).

One more classic problem:

```
#include<stdio.h>

int main()
{
        printf("***** HELLO WORLD *****\n");
        return 0;

// Missing a closing curly brace!
```

Compiling the above produces:

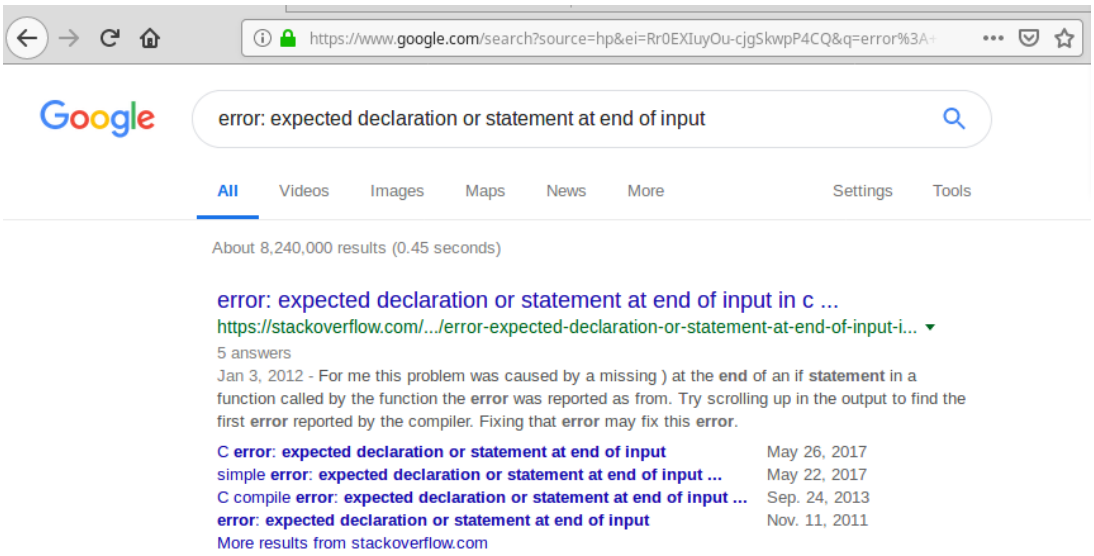
```
\Desktop\C-programs \gcc error_3.c
error_3.c: In function ‘main’:
error_3.c:6:2: error: expected declaration or statement at end of input
  return 0;
  ^
```

The compiler expected to see a closing curly brace somewhere, but instead it found the end of your file.

6 – How to deal with compiler errors

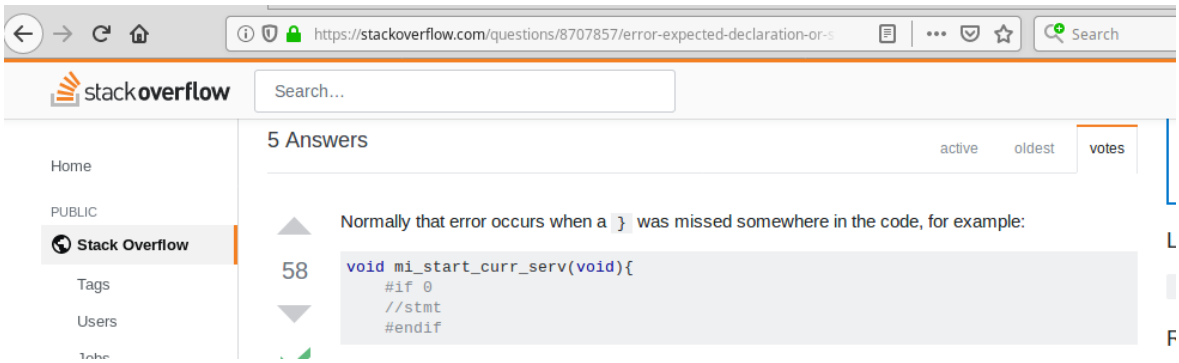
- Starting from the very first one, carefully read the error message and the line it applies to. Look at the code in that line, and the one just above, and check there are no typos or missing characters (e.g. quotes, semicolon, opening or closing parentheses, etc.)

- If you have never seen this particular error message, copy the error message onto Google



Note: Do not include the program name, or line number! Google doesn't know your specific program, but it probably has seen ***lots of developers*** who had the same error before and can tell you what may be causing it.

- Carefully review the first few hits Google comes up with, chances are the answer you need is there.



- Google is your best tool when dealing with compiler errors – learn to use it well!

- However, Google is not perfect and sometimes it won't find the solution to your problem. In that case, come talk to your TA or course instructor! Or ask for help on our forum.

- ***Keep a journal of errors you encountered, and what caused them***. So you can refer to it in the future and save time and effort. You will in time remember these things without having to visit your journal, but while you're learning, it's a great tool!

- Done fixing one error? Go on to the next one. Keep going until no errors are left.

7.- A few notes on printf()

The printf() function will be with us all through the term. We will use it, of course, to output information and results our program is producing. And we will use it extensively to *test* and *debug* what our program is doing.

You should become familiar with how it works - it should be familiar if you recall f_string.format(…) :

```
printf(“ … formatting string … “, variables, to, print, separated, by, commas);
```

The formatting string specifies what printf() is going to do with the list of variables it's printing, it tells printf the *data type* of the variable it is receiving, so that it is printed with the correct format. Formatting options are specified using the '%' character as follows:

```
%d – Prints a decimal (integer) number
%f – Prints a floating point number
%c – Prints a single character
%s – Prints a string
%g – Prints a floating point number using scientific notation
%p – Prints a pointer (this will be formated as a hexadecimal value)
```

There are many more formatting options, and you can specify things such as the number of digits to print for the integer and fractional parts of a floating point number. If you are looking for a way to give your program's output a specific format, look at the on-line documentation for printf(). Chances are the function does what you need, you only need to find the correct format specifier.

Besides these format specifier for variables, the printf() function also makes use of certain control sequences to print special characters. These are identified by the '\' character. Common control sequences include:

```
\n – New line (go to the next line and print there)
\t – Print a 'tab'
\\ - Print a '\' symbol (otherwise the '\' is taken to be part of a control sequence)
%% - Print a '%' symbol (otherwise the '%' is taken to be part of a format specifier)
```

Finally, note that in C, strings are delimited by double quotes "...string...", whereas individual characters are delimited by single quotes, as in 'C'.

Putting this all together, we can understand what a given printf() statement does:

```
printf(“My variables are: %d, %f, %s \n”, my_int, my_float, my_string);
```

The result of the above will be to print a line that looks like

```
My variables are: 10, 3.14159265, Hello World
```

And at the end of the string the '\n' sequence moves to the next line, so anything we print after will be output to the next row of text in our terminal.

Of course, the *data type* for your variables must match the format specified in printf(), otherwise you will receive a *compiler warning*.

```
#include<stdio.h>
int main()
{
    float pi;
    pi=3.14159265;
    printf("Printing an int, %d\n",pi);
}
```

Compiling the above results in (ignore the `...\`, it represents the directory where I stored my code! It will be different for you. The compile command starts with 'gcc'):

```
...\gcc printf_test.c
printf_test.c: In function ‘main’:
printf_test.c:6:12: warning: format ‘%d’ expects argument of type ‘int’, but
argument 2 has type ‘double’ [-Wformat=]
     printf("Printing an int, %d\n",pi);
            ^
```

Keep an eye out for compiler warnings – they indicate that there is something *fishy and probably wrong with your code*, but the program still compiles and runs:

```
...\a.exe
Printing an int, 190300824
```

Obviously, this is not what we would expect!

In general – You must make sure to find out, and clear, any compiler warnings. Except for very rare instances, they indicate a problem with your code. You have to be very, very sure you know what you're doing if you choose to ignore a compiler warning.

Exercise: Write a little program that initializes a variable 'pi' to 3.14159265, then prints out pi as an integer (should print 3), and as a floating point number with increasing fractional part lengths. I.e, the output of your program should look like:

```
3
3.1
3.14
3.141
3.1415
3.14159
3.141592
3.1415926
3.14159265
```

8.- Basic C control structures and loops

We're almost ready to jump into the details of how to implement algorithms using C. But first, we should have a look at how the control structures and loops you learned to use in A08 for Python are written and used in C.

As you will find out, *the concept and the way you use these structures is identical*, the only thing that changes is the syntax of how it looks in C. Happily, *you can always look up the syntax if you don't remember it at some point,* and it will become familiar and comfortable to you as we go along with the course.

Loops

The most common types of loops in C are *for loops* and *while loops*. For loops have a simple syntax:

```
#include<stdio.h>

int main()
{
     int i;

     for (i=0;  i<10;  i=i+1)
     {                                  // - start of for loop
        printf("%d\n",i);
     }                                  // - end of the for loop
}
```

Note:

- The for loop uses a *counter* variable to keep track of iterations. In the case above we are using an integer variable called 'i' which was declared in advance.
- The definition of the for loop has 3 components separated by ';':

```
(i=0   ;   i<10   ;  i=i+1)
```

This means: Set the initial value for 'i' to 0
The loop must run for as long as 'i<10'
At each iteration, i is increased by 1 (that's the i=i+1 part)

- The body of the for loop is delimited by curly braces '{', '}'

Question: What is the output of the program above?

Variations on for loops

In C, you can have for loops that use floating point variables as counters, that use non-integer increments, negative increments, or in fact no increment at all within the body of the for loop!

Examples:

```
float angle;
float pi;

pi=3.14159265;

for (angle=0.0; angle<2.0*pi; angle=angle+.01)
{
   printf("%f\n",sin(angle));
}
```

The code above will print the sine of angles between 0 and 2*pi at intervals of .01 radians.

```
int i;

for (i=100; i>=0 ; i=i-3)
{
   printf("Counting down, we have %d left!\n",i);
}
```

The code above counts down from 100 in steps of 3.

```
int i,j;   // We can declare variables of the same type in one
           // line, separated by commas.
```

```
for (i=0; i<10 ; i=i+1)
{
   for (j=0; j<i; j=j+1)
   {
    printf("%d, ",j);
   }
   printf("\n");
}
```

A nested loop where the length of the loop on 'j' depends on the value of 'i'.

Question: What is the output of the nested for loops above?

While loops

While loops in C are very similar to while loops in Python:

```
while ( condition )
{
     //    body of the loop
}
```

The *condition* is a logic statement that depends on variables accessible to the function the loop is part of. Logic statements evaluate to either *true* or *false*. The loop is executed for as long as the condition is *true*.

Breaking out of loops

It is not uncommon to have a loop that may be terminated well before its stopping condition is met. For example, suppose you are searching within a very long text document to determine whether it contains the word “chameleon” in it, in *pseudocode* this would look like:

```
while <words remain in the document>
     if current word == “chameleon”
          print out “The word is contained in the document!”
          exit loop
```

Note that we are specifically terminating the loop early – it makes sense, once we have found the word we were looking for there is no sense going over the remaining contents of the document.

For while loop where you may need to exit early, you can add one more condition to the logical expression at the top of the loop, one that makes the whole expression *false* the moment you need to exit. For *for loops*, you can exit the loop at any time by using the '*break*' statement. We will see examples where the break statement comes in handy very soon.

Conditional Statements

Conditional statements are also quite similar to what you're familiar with from A08 and Python. We use *if* statements to evaluate expressions and execute code that depends on the results. The structure of *if...else* statements in C is as follows:

```
if (condition)     // - The condition must be within parentheses
{
     // Code to be executed if condition is true
}
else
{
    // Code to be executed if condition is false
}
```

Of course, you can use nested *if...else* statements. Conditions are specified using the standard comparison operators:

```
a==b       True if a equals b
a>b        True if a is strictly greater than b
a<b        True if a is strictly lesser than b
a>=b       True if a is greater than or equal to b
a<=b       True if a is lesser than or equal to b
a!=b       True if a is not equal to b
```

Note: Not all built-in types can be compared in this way. The above comparison operators work for *int*, *float*, *double*, and *char*. You can not compare two strings using comparison operators, we will learn why in the next unit.

In addition to the above, we can use logical operators to form conditional statements that evaluate multiple relationships between variables. Common logical operators used in *if* statements include:

```
||         Logical or (this is a double vertical bar)
&&         Logical and (this is a double ampersand)
!          Logical not
```

Here's a sample conditional statement using the above operators

```
if ( (a>b && c==d) || e!=f)
{
     // Run this code!
}
else
{
     // Run this code instead!
```

```
}
```

Be careful with the way you use operators. A common mistake is to use a single '&' or a single '|' instead of the double symbol. The single '&' and single '|' perform an arithmetic *and*, and an arithmetic *or* respectively, the result is a binary number instead of a *true/false* value.

A more challenging exercise: Write a small program that uses a *for* loop to go over the numbers from 1 to 100, and prints out any that are *perfect squares*. The output of your program should look like:

```
4 = 2*2
9 = 3*3
16 = 4*4
.          (there are more perfect squares printed)
100 = 10*10
```

You will need to use *for* loops, *conditional statements*, and *printf()*.