GENG5508
Robotics: C Programming
Lecture 01
(Unit Outline and Introduction to C)

Dr. Syed Mohammed Shamsul Islam
School of Electrical, Electronic and Computer Engineering
The University of Western Australia
Why Learning C Programming?

• C is the world's most popular, and widely deployed programming languages.
• The world's popular operating systems, Linux, Windows and Mac OS X, their interfaces and file systems, are written in C.
• The infrastructure of the Internet, including most of its networking protocols, web servers, and email systems, are written in C.
• Software libraries providing graphical interfaces and tools, and efficient numerical, statistical, encryption, and compression algorithms, are written in C.
• Most relevantly, the software for most embedded devices, including those in robots, cars, aircraft, smart appliances, sensors, mobile phones, and game consoles, is written in C.
• The software on the Mars Phoenix Lander is written in C.
Why Learning C Programming?

- Portability on different architectures
  - 4-bit and 8-bit microcontrollers
  - Traditional 16-, 32-, and 64-bit virtual memory architectures in most PCs and workstations
  - Larger 64- and 128-bit supercomputers

  - Traditional large instruction set architectures, such as Motorola 680x0, Sun SPARC, Intel x86, and DEC-Alpha
  - Newer reduced instruction set architectures (RISC), such as SGI MIPS, IBM/Motorola PowerPC
  - Mobile phones, home theatre equipment, routers and access-points
  - Parallel and pipelined architectures
Handbook description of GENG5508

• This unit develops programming and design skills for various robotics technologies such as mobile robots, robot simulators, robot navigation and mapping, vision guidance and tracking and artificial intelligence (AI) for robotics (neural nets and genetic algorithms).

• This also incorporates a significant amount of C programming skills. The unit introduces a procedural programming language in the context of mobile robotics as an engineering application discipline, as well as fundamental robotics concepts.

• Contents delivery of the two areas are strongly interleaved. The unit contents include
  ➢ (1) programming—advanced C programming, data structures, procedural control elements, and usage of libraries for larger projects; and
  ➢ (2) robotics—introduction to mobile robots, driving robots design and kinematics, use of real mobile robots and robot simulator (in tutorials and laboratories), robot navigation and mapping, vision guidance and tracking, and AI for robotics (neural nets and genetic algorithms).
Topics to be covered in the unit

• An introduction to the ISO-C99 programming language
  The structure of a C program, variables, initialization of variables, basic datatypes - integers, floats, characters and Booleans, enumerations, precedence of operators, flow control.

• Compilation of ISO-C99 programs
  The GNU ISO-C99 compiler, gcc, compiling and linking, scope rules of global, local, and external variables, storage modifiers.

• C programs in greater detail
  The C preprocessor - header file inclusion, textual constants, macros, conditional compilation, portability of C programs, using third-party libraries.

• Basic data structures and data representation
  Arrays and character strings, enumerated types, user-defined types and structures, bitwise operators.

• The use of functions and basic I/O
  Functions, parameter passing, return values, C's stdio functions, function prototypes, external functions, passing functions as parameters, variadic functions.
Topics to be covered in the unit…

• **Pointers in C**
  Parameter passing-by-reference using pointers, pointers to arrays and character strings, command-line parameters, pointer arithmetic.

• **Dynamic memory allocation**
  Motivation, allocation and de-allocation, self-referential structures - linked lists, trees.

• **The Standard C library**
  String handling functions, mathematical functions, formatted I/O, file I/O, buffered and un-buffered I/O, sorting vectors, dates and times.

• **GUI Toolkit**
  FLTK and FLUID.

• **Developing C projects**
  Managing multi-file projects with *make*, calling C programs and functions from other languages (Java, MATLAB...).
Assessment Schedule

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid-term Exam</td>
<td>25%</td>
<td>7th Week</td>
</tr>
<tr>
<td>Practical (Labs)- total four</td>
<td>10%</td>
<td>By Saturday each week. Late penalties apply.</td>
</tr>
<tr>
<td>Quiz</td>
<td>5%</td>
<td>4th Week</td>
</tr>
</tbody>
</table>

- **Lab Submission**
  Preferably through cssubmit https://secure.csse.uwa.edu.au/run/cssubmit. If you have difficulties, directly by emails to me.

- **Lecture and Lab dates**
  Six 2-hour lectures at the beginning of the semester: Tuesday 3:00 to 5:00pm (@ RBST:LT, G.16 - Robert Street Lecture Theatre, Robert Street Building)

- **Lab dates**
  4 Lab sessions (2 supervised hours each): Friday 9-11 am @CSSE 2.05 or **Friday 2-4pm @CSSE 2.03.**
Software Required

• If you want to run the programs on your machine (e.g., laptop), you will need to have gcc or Microsoft Visual Studio or Dav C++ installed.

• Dav C++ is free to download. Other software you can get a copy at: http://web.csse.uwa.edu.au/students/computing/software

• Contact: Faculty IT help: ithelp-ecm@uwa.edu.au; UWA Service Desk <servicedesk@uwa.edu.au>
Schaum's Outline of Programming with C, 2nd Edition
Byron Gottfried
Publisher: McGraw-Hill
532 pages
1996
Some Other Books

**Computer Science: A Structured Programming Approach Using C, 3rd Edition**
Behrouz Forouzan, Richard Gilberg
Publisher: Thomson Course Technology
1184 pages, 2007

**A First Book of ANSI C, 4th Edition**
Gary Bronson
Publisher: Thomson Course Technology
2007

**A Book on C: Programming in C, 4th Edition**
Al Kelley, Ira Pohl
Publisher: Addison Wesley Professional
ISBN: 0201183994, 752 pages
1997

**C: The Complete Reference, 4th Edition**
Herbert Schildt
Publisher: Addison Wesley Professional
ISBN: 0072121246
805 pages
2000
Other Resources

• **Online books and tutorials**
  - [The Definitive C Book Guide and List](#), by contributors to *stackoverflow*.
    *Introductory C Programming*, chapters 1-14.
  - [The C Book](#), by Mike Banahan, Declan Brady and Mark Doran (originally by Addison Wesley, 1991).
  - A tutorial on pointers and arrays in *C*, by Ted Jensen, Sept 2003
  - [C Tutorial](#), from java2s.com *(lots of information, but difficult to navigate)*.

• **C Language and Library Resources**
  - The [Wikipedia entry](#) for the C programming language.
  - [Standard C Library Functions](#). Type `man <function name>` on the CSSE laboratory computers for details.
  - [POSIX Library Functions](#) (which are not part of the Standard C Library)

About the Lecturer

• Name: Asst. Prof Syed Mohammed Shamsul Islam (Shams)
• Age: 36
• Location: CSSE Room 2.16
• Email: syed.islam [at] uwa.edu.au
• Research interests: Computer vision, image processing, artificial intelligence, robotics and computer networking.
• Consultation Time: Friday 11am-12.30pm

Acknowledgement

• Some lecture materials were prepared by Dr. Chris McDonald, Dr. L Barone and Dr. Ferdous Sohel.
So what is C?

- In one breath, C is often described as a good **general purpose language**, an excellent **systems programming language**, and just a **glorified assembly language**.

- C can be correctly described as a general purpose programming language, a description also given to Java, Visual-Basic, C++, and C#.

- C is a **procedural programming language**, having programming features provided by most procedural programming languages –
  - strongly typed variables, constants,
  - standard (or *base*) data types, enumerated types, user-defined types,
  - aggregate structures,
  - standard control flow, recursion, and
  - program modularization.
So what is C?...

- C has separate compilation, conditional compilation, bitwise operators, pointer arithmetic, and language independent input and output.

- However, C is not an object-oriented language like Java, Objective-C, or C#. It does not offer
  - tuples or sets,
  - Java's concept of classes or objects,
  - nested functions,
  - subrange types, (C has only recently added a Boolean datatype).

- Evolution of C:
  - invention in early 1970s,
  - the ANSI-C standard in 1989,
  - the ISO-C99 standard in 1999, and
The structure of C programs

• Hello World! A first C program:

```c
#include <stdio.h>
void main()
{
 printf("Hello World!");
}
```

Notes:

• Characters such as a space, tab, or newline, may appear almost anywhere (used to provide a lay out of the program)- they are stripped out and ignored by the C compiler.

• Functions in C, may be thought of as a block of statements to which we give a name. In this example, we have one function: `main`.

• Lines commencing with a '#' in blue are processed by a separate program, named the C preprocessor.
The structure of C programs...

• High-level structure of a C program:

Of note in this example:

• Keywords, in **bold**, mean very specific things to the C compiler.

• Lines in **green** are **comments**. They are ignored by the C compiler, and may contain (almost) any characters. C99 provides two types of comments -
  /* block comments */ and
  // comments to the end of a line

• A variety of brackets are employed, in pairs, to group together items to be considered in the same way. Here:
  • angle brackets enclose a filename in a `#include` directive,
  • round brackets group items in arithmetic expressions and function calls,
  • square brackets enclose the index when access arrays (vectors and matrices...) of data, and
  • curly brackets group together sequences of one or more **statements** in C. We term a group of statements a **block** of statements.
The structure of C programs...

More notes:

- When our programs are run by the operating system, the operating system always starts our program from `main`. Thus, every complete C program requires a `main` function.

- The operating system passes some special information to our `main` function, `command-line` arguments, and `main` needs a special syntax to receive these.

- When our program finishes its execution, it returns some information to the operating system. Our example here `exits` by announcing either its failure or success.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Compile this program as: 
   gcc -std=gnu99 -Wall -Werror -pedantic -o rot rot.c 
*/
#define ROT 13

static char rotate(char c)
{
    ....
    return c;
}

int main(int argc, char *argv[])
{
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    }
    else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```
Compiling and linking our C programs

• C programs are human-readable text files, that we term source-code files.
  • This makes them very easy to copy, read, and edit on different computers and different operating systems.
  • C is often described as being portable at the source-code level.

• Before we can run (execute) our C programs, we must translate, or compile, their source-code files to files that the operating system can better manage.
  • A program known as a compiler translates (compiles) source-code files into object-code files.

• Finally, we translate or link one or more object-code files to produce an executable program, often termed a 'binary' or an 'exe' file.
  • A program known as a linker performs this translation, also linking our object-code file(s) with standard libraries and (optionally) 3rd-party libraries.
Variables

• Variables
  - Variables are locations in a computer's memory. A typical desktop or laptop computer will have 1GB of memory, or one billion addressable memory locations, and C programs will typically use 4 bytes to hold 1 integer value.
  - Any variable can only hold a single value at any time - they do not maintain a history of past values they once had.

• Naming our variables
  - For better readability, we use simple names reflecting the role of the variable in our programs.
  - Name can be almost anything except-
    - cannot be same as keywords in C
    - they must commence with an alphabetic or the underscore character (_ A-Z a-z), and
    - be followed by zero or more alphabetic, underscore or digit characters (_ A-Z ,a-z, 0-9).
  - C variable names are case sensitive, thus:
    - MYLIMIT, mylimit, Mylimit and MyLimitare four different variable names.
  - Older C compilers may limit variable names to, say, 8 unique characters. Thus,
    - for them, turn_reactor_coolant_on and turn_reactor_coolant_off are the same variable!
      Keep this in mind when writing portable code.
  - While not required, it's preferred that variable names do not consist entirely of uppercase characters.
    - We'll consistently use uppercase-only names for constants provided by the C preprocessor, or user-defined type names: MAXLENGTH, AVATAR, BUFSIZ, and ROT
Basic Data types

<table>
<thead>
<tr>
<th>Type name</th>
<th>Description, and an example of variable initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bool</strong></td>
<td>Boolean (truth values), which may only hold the values of either true or false  &lt;br&gt; e.g. bool finished = false;</td>
</tr>
<tr>
<td><strong>char</strong></td>
<td>character values, to each hold a single value such as an alphabetic character, a digit character, a space, a tab...  &lt;br&gt; e.g. char initial = 'C';</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>integer values, negative, positive, and zero  &lt;br&gt; e.g. int year = 2006;</td>
</tr>
<tr>
<td><strong>float</strong></td>
<td>floating point values, with a typical precision of 10 decimal digits (on our lab machines)  &lt;br&gt; e.g. float inflation = 4.1;</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>&quot;bigger&quot; floating point values, with a typical precision of 17 decimal digits (on our lab machines)  &lt;br&gt; e.g. double pi = 3.1415926535897932;</td>
</tr>
</tbody>
</table>

Above types are standard, or base types that C provides to hold commonly required values, and later we'll see how we can also define our own user-defined types to meet our needs.
The scope of variables

• The scope (visibility or lexical range) of a variable describes the range of lines in which the variable may be used.
  ➢ Global scope (/file scope): Variables are declared outside of all functions and statement blocks, and
  ➢ Block scope: Variables are declared within a function or statement block.

Of note in this example:
• the variable count has global scope. It is defined on line 06, and may be used anywhere from line 06 until the end of the file (until line 26).

• The variable nfound has block scope. It is defined on line 10, and may be used anywhere from line 10 until the end of the block in which it was defined (until line 26).

• The variable nerrors has block scope. It is defined on line 14, and may be used anywhere from line 14 until line 18.

• The variable ntimes has block scope. It is defined on line 20, and may be used anywhere from line 20 until line 24.

• We could define a different variable named nerrors in the block of lines 20-24 - without problems.

• We could define a different variable named nfound in the block of lines 20-24 - but this would be a very bad practice!
Increment and decrement of a variable

• The "traditional" mechanism of incrementing integer values, in both assignment statements and in for loops:

```c
int value = 0;
....
value = value + 1;
....
for(int i=0 ; i < MAXVALUE ; i=i+1) {
    ....
}
```

• C provides a shorthand notation (operator) for incrementing and decrementing scalar variables, by one:

```c
int value = 0;
char ch = 'z';
++value;       // value is now 1
--ch;          // ch is now 'y'
....
for(int i=0 ; i < MAXVALUE ; ++i) {
    ....
}
for(char letter='a' ; letter <= 'z' ; ++letter) {
    ....
```
Increment and decrement of a variable...

- While pre- and post-incrementing (and decrementing) initially appears simple, we must be careful when using modified variables in expressions. Consider these results:

  ```
  int x = 0;
  int y = 0;
  int what = 0;

  // -------------- what --- X --- Y --------------
  what = ++x;   // 1 1 0
  what = y++;   // 0 1 1
  what = y++;   // 1 1 2
  what = ++y;   // 3 1 3
  ```

  - When the increment or decrement operator follows the variable, the operation is performed after its value has been obtained for use in the expression.
  - When the operator precedes the variable, the operation is performed first, and then the value of the variable is obtained for use in the expression.

- **Shorthand arithmetic**
A similar notation may be used to perform any standard arithmetic operation on a variable. For example, assuming the correct declarations:

  ```
  value += 2;   // equivalent to value = value + 2;
  total *= x;   // equivalent to total = total * x;
  half /= 2;    // equivalent to half = half / 2;
  poly += x+1;  // equivalent to poly = poly + (x+1);
  ```
Other operators in C

- **Relational Operators**: (Associativity: L→R)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

- **Logical Operators**: (Associativity: L→R)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
</tr>
</tbody>
</table>

- **Arithmetic Operators**: (Associativity: L→R)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
</tbody>
</table>

- **Conditional Operators** (?, ?): (Associativity: R→L)

\[ \text{expression 1} \ ? \ \text{expression 2} : \ \text{expression 3} \]

- If \text{expression 1} is true (i.e., if its value is nonzero), then \text{expression 2} is evaluated and this becomes the value of the conditional expression.
- However, if \text{expression 1} is false, then \text{expression 3} is evaluated and this becomes the value of the conditional expression.
Flow control

- **Conditional execution**
  Conditional statements first evaluate a Boolean condition and then, based on whether it's true or false, execute other statements.

The most common form is:

```c
if(condition) {
    more statements;
    ..... 
} else {
    more statements;
    ..... 
}
```

Sometimes, the else clause is omitted:

```c
if(condition) {
    more statements;
    ..... 
}
```

Often, the else clause itself provides further if statements:

```c
if( condition1) {
    more statements;
    ..... 
} else if(condition2) {
    more statements;
    ..... 
} else {
    more statements;
    ..... 
}
```
Flow control...

- **Multiple Conditions** (Executes from left to right)

```c
if(ConditionOne && ConditionTwo && ConditionThree)
{
    Code to execute
}

if(ConditionOne)
{
    if(ConditionTwo )
    {
        if(ConditionThree)
        {
            Code to execute
        }
    }
}

if ( a = 10 || b < 20 )
{
}
```
Flow control...

- **Bounded Loop:**

  ```c
  // here, variable i holds the values 1,2,...10
  for(int i = 1 ; i <= 10 ; i = i+1) {
    // the above introduced a loop-control variable, i
    ......
    printf("loop number %d\n", i);
    ......
    // variable i is available down to here
  }
  // but variable i is not available from here
  ``

  ```c
  // here, variable ch holds each lowercase value
  for(char ch = 'a' ; ch <= 'z' ; ch = ch+1) {
    ......
    printf("loop using character \'%c\'\n", ch);
    ......
  }
  ``

- C provides its **for** control statement to loop through a sequence of statements, a block of statements, a known number of times.

- A variable called *loop control variable* is used to count how many times we go through the loop (e.g. *i* in the top example).

- The loop control variable does not always have to be an integer (i.e. *ch* in bottom example is a *char* type variable).

- The variables may be used *inside* each loop, in the statement block, but then "disappear" once the block is finished (after its bottom curly bracket).

- It's also possible to use any other variable as the loop control variable, even if defined *outside* of the for loop. In general, we'll try to avoid this practice - unless the value of the variable is required outside of the loop.
Flow control...

• **Unbounded Loops:**

Unbounded loops are used when we *don't* know, ahead of time, how many iterations may be required.

C provides two types of unbounded loops:
- the **while** loop, where zero or more iterations are made through the loop
- the **do....while** loop, where at least one iteration is made through the loop

Notice that in both cases we still use a variable, `i`, to control the number of iterations of each loop, and that the changing value of the variable is used to determine if the loop should "keep going".

However, the statements used to modify the control variable may appear almost anywhere in the loops. They provide flexibility, but can also be confusing when loops become several tens or hundreds of lines long.

```c
i = 1;
while(i <= 20) {
    printf("iteration number %d\n", i);
    ....
    ....
    i = some_calculation_setting_i;
    n = n + 1;
}
printf("loop was traversed %d times\n", n);
```

```c
i = 1;
while(1) {
    printf("iteration number %d\n", i);
    ....
    ....
    i = some_calculation_setting_i;
    n = n + 1;
} while(i <= 20);
printf("loop was traversed %d times\n", n);
```
Flow control...

- Loops within loops:

```c
for(int i = 1; i <= 6; i = i+1) {
    for(int j = 1; j <= 4; j = j+1) {
        printf("(%d,%d) ", i, j); // print i and j as if "coordinates"
    }
    printf("\n"); // finish printing on this line
}
```

- Notice that we have two distinct loop-control variables, i and j.
- Each time that the inner loop (j's loop) starts, j's value is initialized to 1, and advances to 4.

- As programs become more complex, we will see the need for, and write, all combinations of:
  - for loops within for loops,
  - while loops within while loops,
  - for loops within while loops,
  - and so on....
Flow control...

- Changing the regular flow of control within loops:

```c
for(int i = 1 ; i <= 10 ; i = i+1) {
    // Read an input character from the keyboard
    //......
    if(input_char == 'Q')  // Should we quit?
        break;
    //......
} // Come here after the 'break'. i is unavailable
```

- Sometimes we need to leave a loop early, using the `break` statement, possibly skipping some iterations and some statements.

- In this example, we iterate through the loop at most 10 times, each time reading a line of input from the keyboard. If the user indicates they wish to quit, we `break` out of the bounded loop.

```c
for(char ch = 'a' ; ch <= 'z' ; ch = ch+1) {
    if(ch == 'm')  // skip over the character 'm'
        continue;
    //......
    statements that will never see ch == 'm'
    //......
}
```

- Sometimes we need to start the next iteration of a loop, even before executing all statements in the loop.

- In this example, we wish to perform some work for all lowercase characters, except 'm'. We use `continue` to ignore the following statements, and to start the next loop (with ch == 'n').
Flow control...

- **The equivalence of bounded and unbounded loops:**
  - We need to accept (for now), that the three "pieces" of the `for` construct, are not always *initialization, condition, modification*.
  - More generally, the three pieces may be C *expressions* - for the moment we'll consider these as C *statements* which, if they produce a value, the value is often ignored.
  - The following loops are actually equivalent:
    ```c
    for(int i = 1 ; i <= 10 ; i = i+1) {
        // Read an input character from the keyboard
        ....
        if(input_char == 'Q') // Should we quit?
            break;
        ....
        ....
    }
    // Come here after the 'break'. i is unavailable
    ```
    ```c
    for(char ch = 'a' ; ch <= 'z' ; ch = ch+1) {
        if(ch == 'm') // skip over the character 'm'
            continue;
        ....
        ....
        statements that will never see ch == 'm'
        ....
    }
    ```
  - In both cases, we're expecting *expression2* to produce a Boolean value, either `true` or `false`, as we need that truth value to determine if our loops should "keep going".
Flow control...

• Some unusual loops you will encounter

Some for or while loops may appear to have something missing. In fact, any (or all!) of the three "parts" of a for loop may be omitted. For example, the following loop initially sets i to 1, and increments it each iteration, but it doesn't have a "middle" test to see if the loop has finished:

```c
for(int i = 1 ; ; i = i+1) {
    ....
    ....
}
```

• Some loops don't even have a loop-control variable, and don't test for their termination. This loop will run forever, until we interrupt or terminate the operating system process running the C program. We term these infinite loops:

```c
#include <stdbool.h>

// cryptic - avolo tnis mecanism
for( ; ; ) {
    ....
    ....
}
```

```c
// clearer - use tnis mecanism
while( true ) {
    ....
    ....
}
```

• While infinite loops are sometimes used, they are not expected to run forever! Typically an enclosed condition and a break statement is used to terminate the loop, either based on some user input, or the state of some calculation.
Some unusual loops you will encounter

Some \texttt{for} or \texttt{while} loops may appear to have something missing. In fact, any (or all!) of the three "parts" of a \texttt{for} loop may be omitted. For example, the following loop initially sets \texttt{i} to 1, and increments it each iteration, but it doesn't have a "middle" test to see if the loop has finished:

\begin{verbatim}
for(int i = 1 ; i = i+1) {
    ....
    ....
}
\end{verbatim}

Some loops don't even have a loop-control variable, and don't test for their termination. This loop will run forever, until we interrupt or terminate the operating system process running the C program. We term these infinite loops:

\begin{verbatim}
// cryptic - avoid this mechanism
for( ; ; )
{
    ....
    ....
}
\end{verbatim}

\begin{verbatim}
#include <stdbool.h>

// clearer - use this mechanism
while( true )
{
    ....
    ....
}\end{verbatim}

While infinite loops are sometimes used, they are not expected to run forever! Typically an enclosed condition and a \texttt{break} statement is used to terminate the loop, either based on some user input, or the state of some calculation.
Flow control...

- The switch Statement

```c
switch(expression){
    case constant-expression :
        statement(s);
        break; /* optional */
    case constant-expression :
        statement(s);
        break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
        statement(s);
}
```

- The switch statement causes a particular group of statements to be chosen from several groups.

- Multiple expressions can have the same statements.

- Beware: once an expression is true, case does not check any further expressions rather executes all of the rest. Therefore, a break command is generally used.