An Introduction to the C Programming Language

In one breath, C is often described as a good general purpose language, an excellent systems programming language and nothing more than a glorified assembly language. So how can it be all three?

C can be correctly described as a successful, general purpose programming language, a description also given to Pascal and, more recently, to Java. Like Pascal, C is a *procedural* programming language, not an object-oriented language like Java. Programs written in C can of course be described as ‘‘good’’ programs if they are written clearly, make use of high level programming practices, and are well documented with sufficient comments and meaningful variable names. Of course all of these properties are independent of C and are provided through many high level languages. C has the high level programming features provided by most procedural programming languages – strongly typed variables, constants, standard (or *base*) datatypes, enumerated types, a mechanism for defining your own types, aggregate structures, control structures, recursion and program modularization. C does not support Pascal’s sets, Java’s concept of a class or objects, a Boolean datatype, nested functions, nor subrange types and their use as array subscripts. C does have, however, separate compilation, conditional compilation, bitwise operators, pointer arithmetic and language independent input and output. The decision about whether C, Pascal, or Java is the best general purpose programming language (if that can or needs be decided), is not going to be an easy one.

C is frequently, and correctly, described as an excellent systems programming language. It is claimed, too, that C provides an excellent operating system’s interface through well defined library routines. Correctly, these statements should be considered in perspective. The C language began its development in the early 1970s, as a programming language in which to write significant portions on the UNIX operating system. Today, well in excess of 99% of the UNIX, LINUX and Windows operating system kernels and their standard library routines, are all written in the C programming language.

Today it is extremely difficult to find an operating system *not* written in either C or its descendant C++. Because C was designed to be an extremely portable language (at the level of its source code), it has enabled the UNIX (and LINUX) operating systems to be ported to many different computer architectures. It is perhaps not surprising, then, that C is described as an excellent systems’ programming language. Most operating systems themselves are written in C and all library routines (in particular the operating system interface provided through *system calls*) are written in C. C functions can of course easily call other C functions, and thus C works arm in arm with the constantly emerging UNIX, LINUX and Windows-NT operating systems.

C has very efficient compilers, libraries and runtime environment support. C compilers have been both developed and ported to a large number and type of computer architectures, from 8-bit microcomputers, through the traditional 16, 32, and 64 bit virtual memory architectures used in most PCs and workstations, to larger 64 and 128 bit supercomputers. Compilers have been developed for traditionally large instruction set architectures, the newer reduced instruction set architectures (RISC), more recently personal data assistants (PDAs), and parallel and pipelined architectures. C’s portability has greatly added to its (and UNIX’s) success. Once a C compiler has been developed for a new architecture (and an architecture and operating system without a C compiler is, today, extremely rare) the gigabytes of C programs and libraries available on other C-based platforms can also be ported to the new architecture. It is often quoted that a C program, when compiled, will run only 1-2% slower than the same program hand-coded in the native assembly language for the machine. But the obvious advantage of having the program coded in a readable, high level language, provides the overwhelming advantages of maintainability and portability. Very little of an operating system, such as UNIX or LINUX, is written in an assembly language – in most cases the rest is written in C. Even the operating system’s device drivers, often considered the most time-critical code in an operating system kernel, today contain assembly language numbered in only the hundreds of lines.

The C Programming Language, *continued*

C is also described as nothing more than a glorified assembly language, meaning that C programs can be written in such an unreadable fashion that they look like your terminal is set at the wrong speed (in fact there’s a humorous contest held each year named *The International Obfuscated C Code Contest*, `http://www.au.ioccc.org/`, for such code). For example, it’s difficult to see how this C program performs prime factorization:

```
int  ll[]={0,2,1,2,2,4,2,4,2,4,6,2,6,0};
char *I[]={"%dx","%d is %se\n","composit","prim"};

main(c,v) char **v; {
 int l,lO,o,O,*OO;
 while(--c)
  for(O=((o=lO=atoi(*++v))<2)<<1;O<=o;printf(*(I+(O>o)),O>o?l:O,*(I+2+(l==lO))))
   for(O=0,l=o,OO=ll;(O+= *(OO-= *++OO?*ll:8))*(o=l/O)!=l&&O<=o;);
}
```

Perhaps C’s biggest problem is that the language was designed by programmers who, folklore says, were not very proficient typists. C makes extensive use of punctuation characters in the syntax of its operators and control flow. In fact, only the punctuation characters `@`, `` ` `` and `$` are *not* used in C’s syntax! It is not surprising, then, that if C programs are not formatted both consistently and with sufficient white space between operators, and if very short identifier names are used, a C program will be very difficult to read! To partially overcome these problems, a number of editors and programs such as *indent* reformat C code for us.

C is also criticized for being too forgiving in its type-checking at compile time. It is possible to *cast* an instance of one type into another, even if the two objects have considerably different types. In particular, a pointer to an instance of one type can be coerced into a pointer to an instance of another type, thereby permitting the object’s contents to be interpreted differently.

C also has no runtime checking of constructs like pointer variables and array indices. Subject to constraints imposed by the operating system’s memory management routines (if any – c.f. the *general protection fault* and *blue screen of death*!), a pointer may point almost anywhere in a process’ address space and seemingly random addresses accessed or written to. Although all array indices in C begin at `0` it is possible to access an array’s ‘‘elements’’ with negative indices or indices beyond the declared end of the array.

Despite all of its weaknesses, and we’ve had no shame admitting them here, the C programming language is an extremely powerful and popular language, and there are probably still more people using C and C++ than any other languages today.

The ANSI Standard C Language

Despite C's long history, being first designed in the early 1970s, it underwent considerably little change until the late 1980s. This is a very lengthy period of time when talking about a programming language's evolution (c.f. Java is barely 5 years old). The original C language was mostly designed by Dennis Ritchie and then described by Brian Kernighan and Dennis Ritchie in their imaginatively titled book *The C Programming Language*. The language described in this seminal book, described as the *K&R* book, is now described as *K&R C* or "old" C. In the late 1980s a number of standards forming bodies, and in particular the American National Standards Association X3J11 Committee, commenced work on rigorously defining both the C language and the commonly provided standard C library routines. The results of their lengthy meetings are termed the ANSI-X3J11 standard, or informally as *ANSI-C*.

The formal definition of ANSI-C introduces surprisingly few modifications to the old *K&R C* language and only a few additions. Most of the additions were the result of similar *enhancements* that were typically provided by different vendors of C compilers, and these had generally been considered as essential extensions to old C. The current ANSI-C language is extremely similar to old C, the committee only introduced a new base datatype, modified the syntax of function prototypes, added functionality to the preprocessor and formalized the addition of constructs such as constants and enumerated types.

A new revision of the C language, named ISO/IEC 9899 by the ISO-JTC1/SC22/WG14 working group, of just C99 was recently completed. Again many features have been "cleaned upp" including the addition of Boolean and complex datatypes, single line comments, and variable length arrays, as well as removing some unsafe features. See `http://wwwold.dkuug.dk/JTC1/SC22/WG14/docs/c9x/`.

Today ANSI-C is now far more widely available and accepted than was old C, it being required for many, both US and Australian, government tenders and, in Australia, being used in all universities and significant information technology-based companies.

Some Web-based C References

All of these C references are accessible from the CS205 (Operating Systems) webpage.

- *C Programming Notes*, a tutorial by Steve Summit.
  `http://undergraduate.csse.uwa.edu.au/units/230.205/summit-intro/`
  and
  `http://undergraduate.csse.uwa.edu.au/units/230.205/summit-intermediate/`
- Learning C from Java.
  `http://www.csse.uwa.edu.au/programming/cfromjava/`
- The *ANSI Standard C Library*.
  `http://www.csse.uwa.edu.au/programming/ansic-library.html`

The GNU ANSI-C Compiler, *gcc*

On our Department's LINUX PCs you will be using an ANSI-C compiler developed by the GNU (pronounced *noo*) group of programmers. The GNU group, standing for *Gnu's Not UNIX*, (or correctly the Free Software Foundation) produces excellent public domain software modeled on some traditional UNIX commands and libraries.

The GNU ANSI-C compiler, *gcc*, is perhaps their best "product", being a C compiler supporting the ANSI-C definition and distributed in (C!) source form for hundreds of different architecture and operating system combinations. *gcc* generates both small and efficient code for its range of target architectures and, in the case of *gcc* running under some commercial operating systems, produces better code, (for a number of significant examples) than the proprietary C compiler distributed with the operating system itself.

Using the *gcc* Compiler Under LINUX

The GNU C compiler, *gcc* can be invoked from the shell's command line like any other LINUX command. Assuming that you've entered an ANSI-C program into a file named `firstprog.c` (using, say, *vi* or *emacs*), a typical compilation of the program would be:

```
charles-p 1. gcc -ansi -o firstprog firstprog.c
```

This will result in the syntactically correct ANSI-C program being compiled and linked into the executable binary file `firstprog`. As `firstprog` is executable and we typically have the present working directory in our shell's search path, we can execute this program with

```
charles-p 2. firstprog
  ... output of firstprog
```

The `-ansi` switch to `gcc` specifies that we want the syntax of the ANSI-C language (rather than "old" K&R C) to be expected. The `-o` switch to `gcc` specifies that we want the resulting binary *output* file to be placed in the (following) indicated file. Note that the C source file `firstprog.c` *must* have the filename extension of `.c`. In this case it is `gcc` that is imposing this restriction and not the LINUX operating system nor file system. Attempts to invoke `gcc` with incorrect switches or syntactically incorrect programs will result in a flurry of error messages.

`gcc` supports a huge number of switches, more than `ls` (!), though only a few will be used in practice. Depending on the switches and filenames presented to `gcc`, the compilation process consists of 2 or 3 independent passes, each run as a separate LINUX processes: the C-preprocessor, compilation and code generation, and optional optimization. `gcc` has the expected LINUX manual entry, though the manual entry only describes the extensive list of switches to `gcc` and its operation, and not the syntax nor semantics of the ANSI-C language itself.

To minimize the risk of programming errors, we'll have `gcc` report as many illegal and "bad practice" errors as possible. For this reason we'll compile all programs as:

```
charles-p 1. gcc -ansi -Wall -pedantic -o firstprog firstprog.c
```

The Structure of a C program

In the following sections we'll consider the aspects of the C language (and ANSI-C in particular) that make it different than Java and Pascal. We'll not spend time on describing what a variable is, nor how control structures can be used in C programs as these are concepts common to most high level languages are not peculiar to C.

C, like Java and Pascal, is described as a *free-format* language, that is statements in C, such as declarations and expressions may be entered without regard to the column position of each line. This concept is easy to grasp after some programming in Java, though different if you're used to programming in many assembly languages or earlier version of Fortran. In particular, white space characters (spaces, tabs and newlines) *should* be used without shame in a C program, particularly if their addition will add to the readability of the program.

Comments in C

Comments in C are used to "hide" some text from the C compiler itself and, of course, used to document sections of programs with natural language descriptions or pseudo-language outlines of an algorithm. Unlike as in Java and Pascal, there is only one method of opening and closing comments in C. Comments begin with the two character sequence `/*` and are closed with the sequence `*/`.

```
/* This is a pretty boring comment in C */
```

There can be no white space characters between the two characters in each case. Any sequence of ASCII characters may appear within the body of a comment and comments are usually used to temporarily "hide" some C code from the C compiler. Unlike some languages, however, comments in C cannot be nested (that is, comments may not appear in comments), and care must be taken if "hiding" C code within a comment, that this C code does not have comments itself!

Comments may appear between any two symbols of a C program, for example

```
result = a /* this is perfectly legal here */ + b;
```

Unlike Java and C++, there is no simple `// comment to end of line.`

Be aware that some older C texts will tell you that comments may be placed *within* an identifier!

```
ident/* no longer legal */ifier
```

While acceptable in old K&R C, this is no longer valid under ANSI-C.

The C Preprocessor

You will notice that a few lines, typically near the beginning, of a C program begin with the hash or pound sign, `#`. These lines are termed *C preprocessor directives* and are actually instructions (directives) to a special program called the C preprocessor (located in `/lib/cpp`). As its name suggests, the C *pre*processor processes the text of a C program *before* the C compiler sees it. The preprocessor directives (all beginning with `#`) should begin in column 1 (the 1st column) of any source line on which they appear. The C preprocessor is easily able to locate these lines and then examine the characters following the `#`. The following characters usually form a special word in the C preprocessor's syntax which typically cause the preprocessor to modify the C program before it is sent to the C compiler itself. Although there are about 20 different preprocessor directives, well only discuss the most common one here and then a few others as we need them.

Header File Inclusion

The `#include` directive, pronounced *hash include*, typically appears at the beginning of a C program. It is used to *textually* include the entire contents of another file at the point of the `#include` directive. A common `#include` directive, seen at the beginning of most C files is

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

This directive indicates that the contents of the file named `stdio.h` should be included at this point (the directive is replaced with the contents). There is no limit to the number of lines that may be included with this directive and, in fact, the contents of the included file may have further `#include` directives which are handled in the same way. We say that the inclusions are *nested* and, of course, care should be taken to avoid recursive nestings!

The example using `<stdio.h>`, above, demonstrates two important points. The filename itself appears between the characters `< ... >`. The use of these characters indicates that the enclosed filename should be found in the *standard include* directory, `/usr/include`. The required file is then `/usr/include/stdio.h`.

The *standard include* files are used to consistently provide system-wide data structures or definitions that are required in many different files. By having the standard include files centrally located and globally available, all C programmers are guaranteed of using the same data structures and definitions that they (all) require. ANSI-C only defines 15 operating system independent header files.

Have a (recursive) look in the `/usr/include` directory yourself and you see that there are over 2000 standard include files available under LINUX!

The C Preprocessor, *continued*

Importantly, it is the use of the < ... > characters which signify that the `/usr/include` directory name should be prepended to the filename to locate the required file. Alternatively, the `" ... "` characters may also be used, as in the following example:

```
#include "mystructures.h"
```

to include the contents of the file `mystructures.h` at the correct point in the C program. Because the `" ... "` characters are used, the file is sought in the *present working directory*, that is `./mystructures.h`. By using the `" ... "` characters we can specify our own include files which are located in the same directory as the C source programs themselves.

In both of the above examples the indicated filename had the "extension" of `.h`. Whereas we have previously said that the "extension" of `.c` is expected by the C compiler, the use of `.h` is only a convention within UNIX. The `.h` indicates that the file is a *header file*, because they generally contain information required at the *head* (beginning) of a C program. Header files typically (and should) contain *only definitions* of C constructs, like data structures and constants used throughout the C program. In particular, they *should not* contain any executable code, variable declarations nor C statements.

Defining Textual Constants

Another frequently used C preprocessor directive is the `#define` directive, pronounced *hash define*. The `#define` directive is used to introduce a textual value, or textual constant, which when recognized by the C preprocessor will be textually substituted by its definition. Traditionally `#define` directives were the only method available to C programmers, using old K&R C, of introducing and declaring constants in C programs. For example, two frequently used `#define`-ed constants are:

```
#define  FALSE      0
#define  TRUE       1
```

After these definitions, each time the C preprocessor locates the strings `FALSE` or `TRUE` *as complete words* within the C program, they will be substituted for the *character sequences* `0` and `1` respectively. Although the new ANSI-C standard has introduced a formal **`const`** construct for declaring constants, the `#define` directive is still the preferred method of defining some forms of constants. For example, when declaring an array of integers (described in greater detail later) we use a `#define` directive to define the maximum size of the array. Thereafter we use the `#define`-ed constant in the array declaration:

```
#define  MAXSIZE     100

    int myarray[MAXSIZE];
```

If necessary, a preprocessor token may be undefined is no longer required:

```
#undef   MAXSIZE
```


Textual, Inline Functions

The `#define` directive may also be used to define some inline functions, more correctly termed *macros*, within your C programs. An often cited example is:

```
#define  sqr(x)     x * x
```

C does not have a standard function for calculating the square of, say, an integer value, but using the inline macro defined above, we can now write:

```
    result = sqr(i);
```

where `i` is an integer variable. Notice that the macro substitution was performed with the macro's argument being `i`. In a manner akin to actual and formal parameter naming in Java (and C), the actual parameter `i` is represented in the macro as the formal parameter `x` without problems. Each time `x` appears as a unique "word" in the right-hand-side of the definition, it will be replaced in the C code by `i`.

Notice that this textual substitution may also be used for calculating (in this example) the square of an integer constant. For example:

```
    result = sqr(3);
```

is expanded in an identical way. Our definition of `sqr` is not really rigourous enough to provide correct results in all cases. For example, consider the "call" to `sqr(x+1)` which would evaluate to `2x+1`! A more correct definition would be:

```
#define  sqr(x)     ((x) * (x))
```

Conditional Compilation

Another often used feature of the C preprocessor is the use of conditional compilation directives. The C compile pre-defines a few constants to "tell" the program the operating system in use, filename being compiled, and so on:

```
#if   defined(linux)
    /* compile code specific to LINUX */
     ......

#elif defined(WIN32)
    /* compile code specific to Windows */
     ......

#elif defined(sun) && defined(SVR4)
    /* compile code specific to Sun's Solaris */
     ......

#endif
```

Operators in C

Nearly all operators in C are identical to those of Java. However the role of C in system programming exposes us to much more use of the shift and bit-wise operators than in Java.

- Assignment
  `=` (not := as in Pascal)

- Arithmetic
  +, −, *, /, %, unary − (there is no unary +)
  Only one / (not / and **`div`** as in Pascal)
  Priorities may be overridden with ( )'s.

- Relational
  >, >=, <, <= (all have same precedence)
  == (equality) and != (inequality)

- Logical
  && (and), || (or), ! (not)

- Pre- and post- decrement and increment
  Any (integer, character or pointer) variable may be either incremented or decremented before or after its value is used in an expression.

  For example :
  - `--fred` will decrement `fred` before value used.
  - `++fred` will increment `fred` before value used.
  - `fred--` will get (old) value and then decrement.
  - `fred++` will get (old) value and then increment.

- Bitwise operators and masking
  `&` (bitwise and), | (bitwise or), `~` (bitwise negation).
  To check if certain bits are on `(fred & MASK)` etc.
  Shift operators << (shift left), >> (shift right).

- Combined operators and assignment
  `a += 2;` `a -= 2;`
  `a *= 2` (should be `a = a<<2;`)
  May be combined as in `a += b;` `a = a+b;`

- Type coercion
  C permits assignments and parameter passing between variables of different types using *type casts* or *coercion*. Casts in C are not implicit, as in Pascal assignments, and are used where some languages require a "transfer function".

Precedence of operators in C

- Expressions are all evaluated from left-to-right, and the default precedence may be overridden with brackets.

```
() coercion        (highest)
++  --  !  ~
*  /  %
+  -
<<  >>
!=  ==
&
|
&&
||
? :
=
,                  (lowest)
```

Variable names in C

Variable names (and type and function names as we shall see later) must commence with an alphabetic or the underscore character `A-Za-z_` and be followed by zero or more alphabetic, underscore or digit characters `A-Za-z_0-9` (Ref: p64). Most C compilers, such as `gcc`, accept and support variable, type and function names to be up to 256 characters in length.

Some older C compilers only supported variable names with up to 8 unique leading characters and keeping to this limit may be preferred to maintain portable code.

It is also preferred that you do not use variable names consisting entirely of uppercase characters – uppercase variable names are best reserved for `#define`-ed constants, as in `MAXSIZE` above. Importantly, C variable names are *case sensitive* and

`MYLIMIT, mylimit, Mylimit` and `MyLimit`

are four different variable names.

Base Datatypes in C

Variables are declared to be of a certain *type*, this type may be either a *base* type supported by the C language itself, or a *user-defined type* consisting of elements drawn from C's set of base types. C's base types and their representation on our labs' Pentium PCs are:

`char`	the character type, 8 bits long
`short`	the short integer type, 16 bits long
`int`	the standard integer type, 32 bits long
`long`	the "longer" integer type, also 32 bits long
`float`	the standard floating point (real) type, 32 bits long (about 10 decimal digits of precision)
`double`	the extra precision floating point type, 64 bits long (about 17 decimal digits of precision)
`enum`	the enumerated type, monotonically increasing from 0

Very shortly, we will see the emergence of Intel's IA64 architecture where, like the Power-PC already, **`long`** integers occupy 64 bits.

We can determine the number of *bytes* required for datatypes with the **`sizeof`** operator. In contrast, Java defines how long each datatype may be. C's only guarantee is that:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
```

Storage Modifiers of Variables

Base types may be preceded with one of more *storage modifier* :

`auto`	the variable is placed on the stack (default, deprecated)
`extern`	the variable is defined outside of the current file
`register`	request that the variable be placed in a register (ignored)
`static`	the variable is placed in global storage with limited visibility
`typedef`	introduce a *user-defined type*
`unsigned`	storage and arithmetic is only of/on positive integers

Initialization Of Variables

All scalar **`auto`** and **`static`** variables may be initialized immediately after their definition, typically with constants or simple expressions that the compiler can evaluate at compile time.

The ANSI-C language defines that all *uninitialized* global variables, and all *uninitialized* **`static`** local variables will have the "starting" values resulting from their memory locations being filled with zeroes - conveniently the value of 0 for an integer, and 0.0 for a floating point number.

Scope Rules Of Global Variables

In Java, a "variable" is simply used as a *name* by which we refer to an object. A newly created object is given a name for later reference, and that name may be re-used to refer to another object "later" in the program. In C, a variable more strictly refers to a memory address (or contiguous memory address starting from the indicated point) and the *type* of the variable declares how that memory's contents should be interpreted and modified.

C only has two true lexical levels, *global* and *function*, though sub-blocks of variables and statements may be introduced in sub-blocks in many places, seemingly creating new lexical levels. As such, variables are typically defined globally (at lexical level 0), or at the *start* of a statement block, where a function's body is understood to be a statement block.

Variables defined globally in a file, are visible until the end of that file. They need not be declared at the top of a file, but typically are. If a global variable has a storage modifier of **`static`**, it means that the variable is only available from within that file. If the **`static`** modifier is missing, that variable may be accessed from another file if part of a program compiled and linked from multiple source files. The **`extern`** modifier is used (within "our" file) to *declare* the existence of the indicated variable in another file. The variable may be *declared* as **`extern`** in all files, but must be *defined* (and not as a **`static`**!) in only a single file.

Scope Rules Of Local Variables

Variables may also be declared at the beginning of a statement block, but may not be declared anywhere other than the top of the block. Such variables are visible until the end of that block, typically until the end of the current function. A variable's name may *shadow* that of a global variable, making that global variable inaccessible. Blocks do not have names, and so shadowed variables cannot be named. Local variables are accessible until the end of the block in which they are defined.

Local variables are implicitly preceded by the **`auto`** modifier – as control flow enters the block, memory for the variable is allocated on the run-time stack. The memory is automatically "deallocated" (or simply becomes inaccessible) as control flow leaves the block. The implicit **`auto`** modifier facilitates recursion in C – each entry to a new block allocates memory for new local variables, and these unique instances are accessible only while in that block.

If a local variable is preceded by the **`static`** modifier, its memory is not allocated on the run-time stack, but in the same memory as for global variables. When control flow leaves the block, the memory is not deallocated, and remains for the exclusive use by that local variable. The result is that a **`static`** local variable retains its value between entries to its block. Whereas the "starting" value of an **`auto`** local variable (sitting on the stack) cannot be assumed (or more correctly, should be considered to contain a totally random value), the "starting" value of a **`static`** local variable is as it was when the variable was last used.

Flow of control in a C program

Control flow within C programs is almost identical to the equivalent constructs in Java. However, C provides no exception mechanism, and so C has no **`try`**, **`catch`**, and **`finally`** constructs.

- Conditional execution

```
if ( expression )
        statement1;

if ( expression )  {
        statement1;
        statement2;
         ......
}

if ( expression )
        statement
else
        statement
```

Of significance, and a very common cause of errors in C programs, is that C has no Boolean datatype. Instead, any expression that evaluates to the integer value of 0 is considered false, and any non-zero value as true. A conditional statement's controlling expression is evaluated and if non-zero (i.e. true) the following statement is executed. Most errors are introduced when programmers (accidently) use embedded assignment statements in conditional expressions:

```
if (loop_index = MAXINDEX )
        statement;

   /* instead of ... */

if (loop_index == MAXINDEX )
        statement;
```

A good habit to get into is to place constants on the left of (potential) assignments:

```
if (0 = value )
        statement;
```

When compiling with `gcc -ansi -Wall -pedantic ...` the only way to "shut the compiler up" is to use extra parenthesis:

```
if ( ( loop_index = MAXINDEX ) )
        statement;
```


Flow of control in a C program, *continued*

C's other control flow statements are very unsurprising:

```
while ( conditional-expression ) {
        statement1;
        statement2;
         ......
}

do {
        statement1;
        statement2;
         ......
}  while ( conditional-expression );

for( initialization ; conditional-expression ; statement3 ) {
        statement1;
        statement2;
         ......
}
```

Any of the 4 components may be missing, If the conditional-expression is missing, it is always true,

Infinite loops may be requested in C with **`for`**(; ;) ... or with **`while`**(1) ...

The equivalence of `for` and `while`

```
for ( expression1 ; expression2 ; expression3 )  {
        statement1;
}

expression1;
while ( expression2 ) {
        statement1;
    expression3;
}
```

The `switch` statement

```
switch ( expression ) {
      case const1 : statement1; break;
      case const2 : statement2; break;
      case const3 :
      case const4 : statement4;
      default     : statementN; break;
}
```

One of the few differences here between C and Java is that C permits control to ‘‘drop down’’ to following **`case`** constructs, unless there is an explicit **`break`** statement.

The `break` statement

```
for ( expression1 ; expression2 ; expression3 ) {
      statement1 ;
      if( ... )
          break;
      statementN ;
}
while ( expression1 ) {
      statement1 ;
      if( ... )
          break;
      statementN ;
}
switch ( expression1 ) {
      case const1 : statement1;

      case const2 : statement2;
          break;

      default     : statementN;
}
```

The `continue` statement

```
for ( expression1 ; expression2 ; expression3 ) {
      statement1 ;
      if( ... )
          continue;
      statementN ;
}
while ( expression1 ) {
      statement1 ;
      if( ... )
          continue;
      statementN ;
}
```