An Introduction to the C99 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 Java and C++. C is a *procedural* programming language, not an object-oriented language like Java or C++. 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 sets of data, Java's concept of a class or objects, nested functions, nor subrange types and their use as array subscripts, and has only recently added a a Boolean datatype. C does have, however, separate compilation, conditional compilation, bitwise operators, pointer arithmetic and language independent input and output. The decision about whether C, C++, 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, Mac-OSX, and Windows-XP 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++.

C is the programming language of choice for most systems-level, engineering, and scientific programming. 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; and the software for most embedded devices, including those in cars, aircraft, robots, smart appliances, sensors, mobile phones, and game consoles, is written in C.

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.

The C Programming Language, *continued*

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.

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).

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 Standardization of the 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. in common discussions, Java is considered only 10 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 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 up'' 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, and the C99 standard is rapidly gaining wider use.

C is again being required for many government tenders and being used in all universities and significant information technology-based companies.

The GNU C Compiler, *gcc*

On our Department's LINUX PCs you will be using an 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 C compiler, *gcc*, is perhaps their best ''product'', being a C compiler supporting both the ANSI-C and ISO-C99 definitions 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 C99 program into a file named `firstprog.c` (using, say, *vi* or *emacs*), a typical compilation of the program would be:

```
prompt-1. gcc -std=c99 -o firstprog firstprog.c
```

This will result in the syntactically correct C99 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

```
prompt-2. firstprog
  ... output of firstprog
```

The `-std=c99` switch to `gcc` specifies that we want the syntax of the C99 language (rather than ''old'' K&R or ANSI-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 C99 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:

```
prompt-1. gcc -std=c99 -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 C99 in particular) that make it different than Java. 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, 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 Java, 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;
```

And like Java and C++, there is also a 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 C99.

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, 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`.

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:

`bool`	an enumerated type, either **`true`** or **`false`**
`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 C99 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 pre C99 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 -std=c99 -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 ;
}
```

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 declarations 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 declarations that they (all) require. C99 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 declarations* of C constructs, like data structures and constants used throughout the C program. In particular, they *should not* contain any executable code, variable definitions, 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 constants in C programs. For example, two frequently used `#define`-ed constants are:

```
#define  FRESHMAN      1
#define  SOPHOMORE     2
#define  JUNIOR        3
#define  SENIOR        4
```

After these definitions, each time the C preprocessor locates the sequence `JUNIOR` *as a complete word* within the C program, it will be substituted for the *character sequence* `3`. Although the new ANSI-C standard has introduced a formal **`const`** construct for supporting constants, the `#define` directive is still the preferred method of defining some forms of constants. For example, when defining 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 definition:

```
#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
```


Functions in C

Java supports *constructors* and *methods* which allocate instances of, and interrogate and modify the state of, their own (implicit) objects. Constructors and methods are typically directed by their parameters. C is a procedural programming language, meaning that its primary synchronous control flow mechanism is the function call. Strictly speaking, C has no procedures, but instead has functions, all of which return a single instance of a base or user-defined type. C's functions access and modify the global memory, and (possibly) their parameters. Although we may hope that a function can only modify memory that it can ''see'' (through C's scoping rules) or has been provided (through its parameter list), this is untrue.

By stating that there are only functions, in we suggest that all functions must return a value. While *nearly* true, C also has a **`void`** type, difficult to describe, and often used as a place holder (to keep the compiler happy!). We may think of a procedure in C, as a function that returns a **`void`** − nothing is returned. With a similar thought, we will often invoke a function, but have no use for its return value. For example, a function such as `printf()` will return an integer as its result, but we rarely need to use this integer. We can ''cast its value'' to **`void`**, effectively throwing away the value.

```
printf( .... );
```

The default return datatype of a function is **`int`** − if a function's datatype is omitted, the compiler assumes it to be an **`int`**. This has the unpleasant result, that if an external or yet to be defined function's prototype is omitted, the compile will often silently assume an **`int`** return result. This is a frequent cause of problems, particularly when dealing with functions returning floating point values, as in C's mathematics library. The use of *gcc's* `-pedantic` switch allows us to trap most such errors.

Every complete C program has an entry point named *`main`*, at which it appears the operating system calls the program. Function `main` is of type **`int`** − this **`int`** is returned as the ''result'' of execution of the whole program, with 0 indicating a successful execution, anything non-zero otherwise.

C's functions may receive zero or more parameters. *All* parameters to C's functions are passed by value. Other than within a single file, the datatype of function parameters between the function's definition and invocation is not checked, i.e. C provides no link-time cross file type checking. Perhaps surprisingly, C also permits functions to receive a variable number of parameters. At run-time it is the function's responsibility to deal with the data types received, and the compiler cannot perform any type checking on these parameters.

Function parameters are implicitly *promoted* to ''higher'' datatypes by the compiler − **`char`**s are promoted to **`int`**s, and **`float`**s are promoted to **`double`**s.

Data structures in C

C has no equivalent construct to the Java class. Instead, C provides two aggregate data structures − arrays and structures.

Arrays in C are not objects, nor strictly single variables. Instead, an array's name is the name referring to the first memory address of a contiguous block of memory of the requested length. Arrays may be declared or defined wherever scalar variables are declared or defined − arrays may be either arrays of C's base types or user-defined types.

There is no *array* keyword in C, and no bounds checking at run-time. C array subscripts commence at 0, the highest valid subscript of `int a[N]` thus being N-1.

- One dimensional arrays Defined with (for example) **int** score[20];

```
-> declare score as array of 20 int
   int score[20]
```

```
   total = 0;
   for(i=0 ; i<20 ; i++)
        total = total + score[i];
```

- Multi-dimesntional arrays?
  Strictly speaking, C does not support multi-dimensional arrays. However, if all (one-dimensional) arrays in c are considered as *vectors*, then multi-dimensional arrays are simply understood as ''vectors of vectors''.

```
 -> explain char str[10][20]
    declare str as array of 10 array of 20 char
```

The number of elements of an array can be determined with :

```
#define  NELEMENTS  (sizeof(score) / sizeof(score[0]))

    for(i=0 ; i<NELEMENTS ; i++)
         total = total + score[i];
```


User-defined C Structures

Structures in C are aggregate datatypes consisting of *fields* or *members* of base types, or other user-defined types. C structures may not include executable code, unlink methods in Java classes.

```
struct person {
    char name[20];
    char addr[80];
    int  age;
};

struct person p1, p2;
int    ages;

ages = p1.age + p2.age;
        /* the sum of their ages */

if(strcmp(p1.name, p2.name) == 0) ...
        /* do they have the same name? */
```

Character arrays and strings

C provides no base type that is a string, though the C compiler accepts the use of double quoted character string literals and ''does the obvious thing''. A string in C is a sequence of characters (bytes) in contiguous memory locations. The string is terminated by the sentinel value of the *NULL* character (zero byte). When a C compiler detects a string literal in a program, it will allocate enough contiguous global (read-only) memory to hold the characters of the string (including the NULL byte at the end).

C does not record the *length* of a string anywhere (as does Java). Instead, by convention, the length of a string is defined as the number of characters from the beginning of the string (its starting address) up to, but not including, the NULL byte. The length of `"hello"` is 5.

- Arrays of characters are typically used to store character strings. Notice that the parameter to the following function does not indicate any expected (maximum) size, or ''length'', of the array.

```
int my_strlen(char str[])
{
    int i = 0, len = 0;

    while( str[i] != '\0' ) {
        len++;
        i++;
    }
    return(len);
}
```

The Standard I/O Library

The C language itself does not define any particular file or character-based input or output routines (nor any windowing routines) − unlike Java. Instead any program may provide its own. Clearly this is a daunting task, and so the standard C library provides a collection of functions to perform file-based input and output. The standard I/O library functions provide efficient, *buffered* I/O to and from both terminals and files.

C programs requiring standard I/O should include the line:

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

All transactions through the standard I/O functions require a *file pointer*:

```
FILE  *fp;

fp = fopen("file.dat", "r");
 ......
fclose(fp);
```

Although we are strictly dealing with a C *pointer*, we simply pass this pointer to functions in the standard C library. Some texts will refer to this pointer as a *file stream* (and C++ confused this even more), but these should not be confused with nor be described as akin to Java's streams.

An number of predicate macros are provided to check the status of file operations on a given file pointer:

```
feof(fp)          /* checks for end-of-file */
ferror(fp)        /* checks for an error on a file */
```

The standard I/O functions all return `NULL` or `−1` (as appropriate) when an error is detected. For example:

```
#include  <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;

  if((fp=fopen("/etc/passwd", "w")) == NULL) {
      error message ...
  }
  else {
      /* process the file */
       ...
      fclose(fp);
  }
}
```


The Standard I/O Library, *continued*

The most frequently used functions in the C standard I/O library perform output of formatted data. We also see here the most frequent use of C's acceptance of functions receiving a variable number of arguments:

```
fprintf(FILE *fp, char *format, (T)arg1, (T)arg2, ...);

e.g.   int  res;
       char *name = "Chris";

       fprintf(fp,"res=%d name=%20s\n", res, name);
```

Many standard I/O functions accept a *format specifier* − a string indicating how following arguments are to be displayed. This mechanism is in contrast to Java's `toString` facility in which each object knows how to output/display itself as a `String` object. There are many possible format specifiers, the most common ones being 'c' for character values, 'd' for decimal values, 'f' for floating point values, and 's' for character strings. Format specifiers may be preceded by a number of format modifiers, which may further specify their data type, and to indicate the width of the required output (in characters).

As a special case, we may use a more concise version of `fprintf()` in which the `FILE` pointer of the operating system's standard output device is used (typically, the screen). Thus, the following two statements are identical:

```
fprintf(stdout, "res=%d name=%20s\n", res, name);
printf("res=%d name=%20s\n", res, name);
```

We mentioned before that the C standard I/O library provides efficient buffering. This means that although it appears that the output has ''gone'' to the `FILE` pointer, it may still be held within an internal character buffer in the library (and will hence not yet be on disk, or to the screen). We often need to *flush* our output to ensure that it is more quickly written to disk or the screen. `FILE` pointers are automatically flushed when a file is closed or the process exits:

```
/* ... format some output ...*/
fflush(fp);
```

As well as outputting to `FILE` pointers, we may also perform formatted output to a character array (a string), with a very similar series of functions:

```
int  res;
char *name = "Chris";
char buffer[BUFSIZ];

sprintf(buffer, "res=%d name=%20s\n", res, name);
```

The Standard I/O Library, *continued*

C's standard I/O library may also be used to input values from `FILE` pointers and character arrays using `fscanf()` and `sscanf()`. Because we want the contents of C's variables to be modified by the standard I/O functions, we need to pass the *address* of the variables:

```
        fscanf(fp, format, &arg1, &arg2, ...);

e.g.    int i, res;
        char buffer[BUFSIZ];

        fscanf(fp, "%d %d", &i, &res);
        sscanf(buffer, "%d %d", &i, &res);
```

We also frequently need to read all lines from a file, or to (perhaps) sum all integers values from a file. We must be careful here, with the particular return values of the C standard I/O functions. The functions themselves return NULL `FILE` pointers, or a value of −1 at the end of a file or an error condition, but we must be care *when* we check these values:

```
#define  MAXLINE      80

int  i, sum;
char line[MAXLINE];

for(;;) {
    fgets(line, sizeof(line), fp);
    if(feof(fp))
        break;

    /* ... process the line just read ...*/
}
fclose(fp);

 ........

sum = 0;
while(fscanf(fp, "%d", &i) == 1)
    sum += i;
fclose(fp);
```


The C/Operating System Interface

Operating systems, such as UNIX, LINUX, Mac-OSX, and Windows-XP, will call C programs with two parameters:

- an **int**eger argument count (`argc`),
- an array of pointers to character strings (`argv`), and

Notice that in many previous examples we've provided a `main()` without any parameters all. Remember that C does not check the length and types of parameter lists of functions which it does ''not know'' about − ones that have not been prototyped. In addition, the function `main()` has no special significance to the C compiler. Only the linker requires `main()` as the apparent starting point of any program. Most C programs you see will only have the first two parameters.

```
int main(int argc, char *argv[])

-> explain char *argv[]
   declare argv as array of pointer to char
```

A common activity at the start of a c program is to search the argument list for command-line switches commencing with a '−' character. Remaining command-line parameters are often assumed to be filenames:

```
int main(int argc, char **argv)
{
    argv0 = (argv0 = strrchr(argv[0],'/')) ? argv0+1 : argv[0];
    argc--; argv++;
    while((argc > 0) && (*argv[0] == '-')) {
        switch (*argv[1]) {
            case 'd' :  dflag = !dflag;
                        break;
            default :   argc = 0;
                        break;
        }
        argc--; argv++;
    }
    if(argc < 0) {
        fprintf(stderr, "Usage : %s %s\n",argv0,usage);
        exit(1);
    }
    if(argc > 0)
        while(argc > 0) {
            process(*argv);
            argv++; argc--;
        }
    else
        process(NULL);
    return(0);
}
```

Pointers in C

The C programming language has a very powerful feature, and if used incorrectly a very dangerous feature, which allows a program (at run-time) to access its own memory. This ability is well supported in the language through the use of *pointers*. There is much written about the power and expressiveness of C's pointers, and much (more recently) written about Java's lack of pointers. More precisely, Java *does* have pointers, termed *references*, but the references to Java's objects are so consistently and carefully constrained at both compile and run-time, that very little can go wrong.

C has both ''standard'' variables and structures, and pointers to these variables and structures (Java *only* has references to objects, and it is only possible to manipulate the computer's memory used to hold the objects, by using references). C's drawback is that while the pointers allow us to easily refer to scalar variables and aggregate structures, C has very little support to prevent us accessing anything else (accidently) at run-time. All speed advantages provided by the availability of pointers, can be trivially consumed by the time taken to debug a program incorrectly using pointers.

C's pointers allow us to refer to the *address* of a variable rather than its value. If this were all that were possible, we may be able to get away without using pointers at all. ''Unfortunately'' parameters to C's functions may only be passed by value, and so a rudimentary understanding of C's pointers is needed to use ''pass-by-reference'' parameter passing in C.

Consider the following example trying to interchange the value of two integer variables:

```
#include <stdio.h>

void swap(int i, int j)
{
    int temp;

    temp = i;
    i    = j;
    j    = temp;
}

int main(int argc, char *argv[])
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(a,b);
    printf("after  a=%d, b=%d\n",a,b);
    return(0);
}

before a=3, b=5
after  a=3, b=5
```


Pass By Reference Using Pointers

Instead, we need to pass a ''reference'' to the two integers to be interchanged, so that the function `swap()` is actually dealing with the original variables, rather than new copies of their values (passed on C's run-time stack).

```
#include <stdio.h>

void swap(int *ip, int *jp)
{
    int temp;

    temp  = *ip;
    *ip   = *jp;
    *jp   = temp;
}

int main(int argc, char *argv[])
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(&a, &b);
    printf("after  a=%d, b=%d\n",a,b);
    return(0);
}

before a=3, b=5
after  a=5, b=3
```

Here we've introduced a bit more syntax (and, typically, it uses punctuation characters).

- The address operator, &, is used to determine the (run-time) memory address of a variable. Here we require the memory address of the variables `i` and `j` before passing these addresses to the `swap()` function. Notice that we are still using pass-by-value parameter passing, but that we are passing addresses on the run-time stack.

- The two asterisks in `swap()`'s formal definition indicate that the variables `ip` and `jp` are *pointers*, or *pointer variables*, rather than just ''simple'' variables. It is typical in C programs to append 'p' or 'ptr' to a variable's name to indicate that it's a pointer.

- The asterisks always placed in front of `ip` and `jp` in function `swap()` indicate that we wish to *dereference* these variables. Instead of using the contents of these variables (which are ''meaningless'' memory addresses) we wish to use the values *pointed to* by these variables. Notice that we may dereference variables on ''both sides'' of an assignment expression.

Pointers To Arrays And Character Strings

One often confusing point in C is the synonymous use of arrays, character strings, and pointers. The name of an array in C, is actually the memory address of the array's first element. Thus the following two assignment statements are the same, and the first is the most commonly used:

```
char   buffer[BUFSIZ], *ptr;

ptr  = buffer;
ptr  = &buffer[0];
```

Using the *cdecl* program again:

```
-> explain char *ptr
   declare ptr as pointer to char
```

If we also remember that C's character strings are simply a contiguous series of characters which, by convention, are terminated by a NULL character, then we can consider strings to be arrays to, and strings may be accessed through pointers (you may wish to consider a string's first character as being stored at the memory address of the array of characters. We can thus write:

```
int   n;
char  *hex_values = "0123456789abcdef";

n = hex_values[ ...expression... ];

/* or ... don't do this! */

n = "0123456789abcdef"[ ...expression... ];
```

We will often see the use of character pointers (used to strings), and character arrays (with assumed terminating NULL characters, used interchangeably:

```
int my_strlen(char *str)
{
    int   len = 0;

    while( str[len] /* != '\0' */ )
        ++len;
    return(len);
```


Pointer Arithmetic

Another confusing facility in C is the use of *pointer arithmetic* with which we may advance a pointer to point to successive memory locations at run-time. It would make little sense to be able to ''point anywhere'' into memory, and so C automatically adjusts pointers (forwards and backwards) by values that are multiples of the size of the base types (or user-defined structures) to which the pointer points(!).

We specify pointer arithmetic in the same way we specify numeric arithmetic, using +, −, and pre- and post- increment and decrement operators (multiplication and division make little sense). We may thus traverse an array with pointer arithmetic:

```
int my_strlen(char *str)
{
    int   len = 0;

    while( *str /* != '\0' */ ) {
        ++len;
        ++str;
    }
    return(len);
}
```

Notice that we are simply ''moving the pointer along'', we are not modifying what it pointers to, simply accessing adjacent memory locations until we reach one containing the NULL character. This example is a little simple, because the character pointer will only be advanced one memory location (one byte) at a time, as a character is one byte long. Alternatively, consider the five equivalent examples:

```
int sum_array(int *values, int n)
{
    int   i, *ip;
    int   sum = 0;

    for(sum=0, i=0 ; i<n ; ++i)
        sum += values[i];

    for(sum=0, i=0 ; i<n ; ++i)
        sum += *(values+i);

    for(sum=0, ip=values; ip<&values[n] ; ++ip)
        sum += *ip;

    for(sum=0, i=0 ; i<n ; ++i) {
        sum += *values;
        ++values;
    }
    for(sum=0, i=0 ; i<n ; ++i)
        sum += *values++;
    return(sum);
}
```

Pointer Arithmetic, *continued*

Unfortunately, we frequently see an excessive use of pointer arithmetic in C with programmers trying to be too smart to speed up their programs. For example:

```
char *my_strcpy(char *dest, char *src)
{
    char   *d = dest;

    while(*dest++ = *src++ );
    return(d);
}
```

With code such as this, in which we are trying to copy all characters from `src` to `dest` until we reach the NULL character, we always have in the back of our minds the concern as to whether the NULL character is in fact copied from the end of `src` to `dest`, and thus legally terminates `dest`.

Sorting An Array Of Values

A frequently required operation is to sort an array of, say, integers or characters. The standard C library provides a generic function named `qsort()` to help with this, but we must write a pointer-based function to perform the comparison of the array's elements:

```
#include <stdlib.h>

#define  N    100

int compare(const int *ip, const int *jp)
{
    return(*i - *j);
}

int main(int argc, char *argv[])
{
    int i;
    int values[N];

    srandom( getpid() );
    for(i=0 ; i<N ; i++)
        values[i] = random();
    qsort((void *)values, (size_t)N,
                sizeof(values[0]), compare);
     ....
    return(0);
}
```


Dynamic Memory Allocation

The function `malloc()` returns a requested number of bytes from the operating system's heap. If insufficient memory is available `malloc` returns `NULL`. When we are finished using the space returned by `malloc()`, our program should be returned to the heap with a call to `free()`. If a process continues to `malloc()` memory and fails to deallocate it using `free()`, the process will quickly ''run out of memory'' and terminate ungracefully.

Unlike Java, C has no garbage collection of heap objects, and so programs must be very careful about deallocating memory that is no longer required.

Consider the following example which allocates space for a new copy of a given string. This is very similar to the standard function named `strdup()`:

```
char *newstr(const char *s)
{
    void *malloc(unsigned int nbytes);
    char *p;

    if( (p=malloc(strlen(s)+1)) == NULL ) {
        fprintf(stderr,"out of memory!\n");
        exit(1);
    }

    strcpy(p,s);
    return(p);
}
```

`malloc()` is also frequently used to allocate memory for structures.

```
#define  NEW(t)     malloc(sizeof(t))

struct l {
     char     *line;
     struct l *next;
};

struct l *hd = malloc( sizeof(struct l) );

fgets(buf, MAX, fp);
while( !feof(fp) ) {
    p       = NEW(struct l);
    p->line = newstr(buf);
    p->next = hd;
    hd      = p;
    fgets(buf,MAX,fp);
}
```