C Lecture 1: An Introduction to C

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. 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, or a functional language like Haskell. 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 modularisation. 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, however, have separate compilation, conditional compilation, bitwise operators, pointer arithmetic, and language-independent input and output. The decision about which 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 portable (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 developed for 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 work-
stations, to larger 64- and 128-bit supercomputers. Compilers have been developed for traditional large instruction set architectures, the newer reduced instruction set architectures (RISC), 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.

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, www.au.ioccc.org, for such code). For example, it’s difficult to see how this C program performs prime factorisation:

```c
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 1,10,o,0,*0O;
  while(--c)
    for(O=((o=l0=atoi(*++v))<2)<<1;O<=o;printf(*(I+(O>o?l:O,*(I+2+(l==l0))))
      for(O=0,l=o,00=ll;0<=*(00-= **00?*ll:8))*(o=1/0)!=1&&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 @`$ 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.

C is also criticised 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 the blue screen of death), a pointer may point almost anywhere in a process’s 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, 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 language today.
The “Standard” C Language

Despite C’s long history, being first designed in the early 1970s, it underwent 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 less than ten years old). The original C language was designed mostly 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 introduced surprisingly few modifications to the old *K&R* C 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 committee introduced only a new base datatype, modified the syntax of function prototypes, added functionality to the preprocessor, and formalised 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 but commonly known as 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 the removal some unsafe features. See wwwold.dkuug.dk/JTC1/SC22/WG14/docs/c9x.

Today’s C is far more widely available and accepted than was old C, it being required for many 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 2230 webpage.

- *C Programming Notes*, a tutorial by Steve Summit.

- Learning C from Java.
  www.csse.uwa.edu.au/programming/cfromjava

The GNU C Compiler, \textit{gcc}

On our Department’s Linux PCs you will be using a C compiler developed by the GNU (pronounced with the ’G’) group of programmers. The GNU group, standing for \textit{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, \textit{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. \textit{gcc} generates small and efficient code for its range of target architectures and, in the case of \textit{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.

\section*{Using \textit{gcc} Under Linux}

\textit{gcc} can be invoked from the shell’s command line like any other Linux command. Assuming that you’ve created a C program in a file named \texttt{firstprog.c} (using, say, \textit{vi} or \textit{emacs}), a typical compilation of the program would be:

\begin{verbatim}
gcc -o firstprog firstprog.c
\end{verbatim}

This will result in the (assumed syntactically correct) C program being compiled and linked into the executable binary file \texttt{firstprog}. As \texttt{firstprog} is executable and we typically have the present working directory in our shell’s search path, we can execute this program just with the command \texttt{firstprog}.

The \texttt{-o} switch specifies that we want the resulting binary file to have the indicated name. Note that the C source file \texttt{firstprog.c} must have the extension \texttt{.c}. \textit{gcc} imposes this restriction and not Linux or the file system. Attempts to invoke \textit{gcc} with incorrect switches or syntactically incorrect programs will result in a flurry of error messages.

\textit{gcc} supports a huge number of switches, more than \textit{ls} (!), although only a few are used in practice. Depending on the switches and filenames presented to \textit{gcc}, the compilation process consists of 2–3 independent passes, each run as a separate Linux process: the C-preprocessor, compilation and code generation, and optional optimisation. \textit{gcc} has the expected Linux manual entry, although the manual entry describes only the extensive list of switches to \textit{gcc} and its operation, not the syntax nor semantics of the C language itself.

To minimise the risk of programming errors, we’ll have \textit{gcc} report as many illegal and “bad practice” errors as possible. For this reason we’ll compile all programs as:

\begin{verbatim}
gcc -Wall -pedantic -o firstprog firstprog.c
\end{verbatim}
The Structure of a C program

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

C, like Java and Pascal, is a free-format language, that is statements in C may be entered without regard to the column position of each line. This concept is easy to grasp after some programming in Java, although different if you’re used to programming in Haskell or in some earlier languages. Layout should be used extensively in C programs to enhance their readability.

Comments in C

Comments in C are used to “hide” text from the C compiler itself and, of course, to document sections of programs with natural language descriptions or pseudo-language outlines of an algorithm. Unlike Java and Pascal, C has only one method of opening and closing comments. 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. 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.
Operators in C

Nearly all operators in C are identical to those of Java. However the role of C in systems’ 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 +)
  Note there is only one /. Priorities may be overridden with brackets

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

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

- Pre- and post-increment and decrement
  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 increments fred before the value is used.
  fred++ uses the old value, then increments.
  --fred decrements fred before the value is used.
  fred-- uses the old value, then decrements.
  These “expressions” can also be used as statements.

- 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;

- 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 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 character or an underscore A-Za-z_, which can be followed by zero or more alphabetic or digit characters or underscores A-Za-z_0-9. Most C compilers 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 eight 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.

Importantly, C variable names are case sensitive, so

MYLIMIT, mylimit, Mylimit, MyLimit

are four different variable names.
Base Datatypes in C

Variables are declared to be of a certain type, which may be either a base type or a user-defined type. 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 modifiers:

- **extern** — the variable is defined outside the current file
- **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
- **register** — request that the variable be placed in a register (ignored)
- **auto** — the variable is placed on the stack (default, deprecated)

Initialisation Of Variables

All scalar **auto** and **static** variables may be initialised 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 uninitialised global variables and all uninitialised static local variables 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 used simply 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 addresses starting from the indicated point) and the type of the variable declares how that memory’s contents should be interpreted and modified.

C has only two true lexical levels, global and function, although 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 they 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 other files, if any. 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 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.
Loops

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

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

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

With for, any of the four components may be missing. If the conditional-expression is missing, it is always true. Infinite loops may be requested in C with for ( ; ; ) ... or while (1) ...

Loop equivalence

The following sequences are equivalent.

for ( expression1; expression2; expression3 ) {
    <statements>;
}

expression1;
while ( expression2 ) {
    <statements>;
    expression3;
}

Conditional execution

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

if ( expression )
    statement1
else
    statement2;

Of significance, and a common cause of errors, is that C has no real Boolean datatype. Instead:

- any expression that evaluates to the integer value of 0 is considered false,
- any non-zero value is considered true.

The condition is evaluated and if it is non-zero (i.e. true) the following statement is executed. Many errors are introduced when programmers (accidentally) use embedded assignment statements in conditional expressions, e.g.

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 -Wall -pedantic ... the only way to “shut the compiler up” is to use extra parentheses:

if ( ( loop_index = MAXINDEX ) )
    statement;

Multiple cases

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 control “drops down” to following cases, unless there is an explicit break statement.

Exiting constructs

for ( expression1; expression2; expression3 ) {
    statement1;
    if ( ... )
        break;
    statement2;
}

while ( expression1 ) {
    statement1;
    if ( ... )
        break;
    statement2;
}

switch ( expression1 ) {
    case const1 : statement1;
    case const2 : statement2; break;
    default : statement3;
}

Continuing execution

for ( expression1; expression2; expression3 ) {
    statement1;
    if ( ... )
        continue;
    statement2;
}

while ( expression1 ) {
    statement1;
    if ( ... )
        continue;
    statement2;
}