Main topics of the week:
- Introduction
- Basics of C and C++
- Differences between C++ and Java
- Compiling a C++ program
- Header Files
- C Input and Output
- C++ Input and Output
- Standard Libraries

Introduction and Overview

This course will cover the C and C++ programming languages and also introduce you to working in the Unix environment. The C language was developed at Bell Laboratories in the early 1970s. Brian Kernighan and Dennis Ritchie (who sadly passed away in fall of 2011) were the primary contributors to the language. They co-authored a book “The C Programming Language”, which became known as the little white book and served to define the language. It still stands today as an excellent example of concise technical writing. Although the language has evolved, some compilers still provide a flag for implementing “K&R C” or “Classic C” as described originally. The C language was a general purpose typed language (although not strongly typed) that came from research into programming languages; C evolved from ideas that appeared in a language called BCPL and a language named B that was based on BCPL, both of which are typeless languages. C is a low level language and has been characterized by some as a high level assembly language. As such, it has been a suitable language for programming operating systems and real time applications as well as most any computer application, and even embedded applications burned on a chip. Fundamental characteristics of C are its simple numerical data types that correspond to native machine word sizes, and its ability to use memory location pointers and do address arithmetic.

In the early 1970s, the Unix operating system was also being developed at Bell Laboratories. Ken Thompson, the author of the B language, was also a primary author of the early Unix implementations. The relative machine independence of the C language, along with its low level characteristics, made it a good choice as the implementation language for a (relatively) portable version of Unix. Together, C and Unix have made a significant impact in computer science.

The C++ language appeared in the early 1980s, first as a stylized use of the C language. Bjarne Stroustrup designed C++ to incorporate ideas from Simula without sacrificing the efficiency and control that contributed to the success of C. C++ appeared on the scene as the object oriented programming paradigm became popular and provided support for many object oriented constructs, although it did not force developers to use the OO paradigm. The key to the success of C++ was its compatibility with the C language: to the greatest extent possible, C++ is a superset of the C language. Although it is more strongly typed than C, it did not attempt to fix the shortcomings of C at the cost of incompatibility. Also key to the success of C++ was an emphasis on preserving the efficiency of C while still allowing the programmer access to the low level control of C. In particular, programs in C++ should not have to pay (in performance) for features that they do not use.
Basics of C and C++

The C language can be described fairly quickly: A C program is simply a collection of global type and variable definitions, and function definitions, and functions cannot be nested. All program variables are typed, and the primitive types are scalars declared using the keywords `int`, `char`, `float`, and `double`. Additionally, `long` and `short` are also integral types, and `unsigned` can be used to modify the integral types to produced non signed (i.e., non-negative) quantities. The keyword `void` can indicate the absence of a function return value (or an unspecified referenced type for a pointer). Identifiers used for variable, function, or type names follow the rules of being composed of letters, digits, and underscore, and cannot begin with a digit. Literal constant values are possible for integral types (sequences of digits), precision types (digits and decimal points as well as exponential notation), character types (enclosed in single quotes, and stored as Unicode values), and string types (sequences of characters enclosed in double quotation marks). More about string types in a moment…

Another simple data type is the enumerated type, which uses the keyword `enum`. This is a way of defining meaningful symbolic constants for integer values.

A declaration may have an asterisk preceding the identifier name, indicating that the variable is a pointer to the declared type, rather than an instance of that type. This means that there are as many different pointer types as there are types. In particular, there is no pointer declaration keyword, and each “pointer” can only address a particular type. More about pointers later…

Another derived type is the array, which again uses no keyword, but instead uses the square brackets to indicate the array. Finally, other derived types are record structures using the `struct` keyword. This allows user defined record types with named fields referring to portions of the record by their type. The `typedef` keyword permits synonyms for types. A `union` is syntactically like a struct, but overlays all the “fields” in the same storage, and so provides alternative type interpretation to single storage locations. This implies an inherent type safety problem for union types.

Blocks of C code are enclosed within (curly) braces, and may be nested. A function definition is a function header (its return type, name, and formal parameter list) followed by the function body, which is a block. A block consists of a sequence of statements, each of which is terminated by a semicolon. All declaration statements must be at the beginning of a block (prior to any statements that are not declarations). The other type of statement (besides declarations) is the “executable” statement. Such statements may be control flow (using keywords `if`, `else`, `while`, `do`, `for`, `return`, `break`, `continue`, `switch`) or simply expressions which are executed for their side effect. These statement constructs are built up from other statements or blocks of statements, resulting in statement nesting. C has very rich expression syntax with many symbolic operators and precedence rules that determine how the operators are combined.

Strings in C

Notice that there is no data type keyword for string in the C language. There is a primitive data type of a single character, which is essentially a one byte numerical quantity. The literal ASCII character values (specified using single quotation marks) are used to specify character values. Strings of characters in double quotes are considered to be literal strings in a program, and are really shorthand for an array of characters, i.e., consecutive bytes in memory. What distinguishes a so called string value from any array of characters is that there is an extra terminating byte at the end with the null character (ASCII NUL – numerical value zero). The actual C type of the string literal is really `pointer to character`, but it can also be used to initialize
a character array, so its interpretation by the C compiler is context sensitive. Since strings are not really primitive types in the language, there are no string operations – all manipulation of strings is by user coded functions. However, the standard C library contains a number of efficient functions for common string manipulations, e.g., copying strings, comparing strings, finding a string’s length.

Storage Types and Variable Scope

There are a few remaining keywords in the C language, and they are used to indicate memory storage types. A variable declared within a block is, by default, automatic storage type, which means that the storage space is on the stack frame. The keyword **auto** (deprecated in C++ 11) is implicit for such declarations and so you very seldom see it used. A declaration outside of any block is static storage type, which means that the storage is permanent and independent of how the program executes. Static storage can be considered to be part of the original executable image that is loaded when the program begins execution. If the keyword **static** modifies a declaration, then such permanent storage is used for variables inside blocks instead of the stack. Variable scope is determined by the variable’s placement: the scope extends from the point of the declaration statement to the end of the block enclosing the declaration. Variables outside of any block are global in scope. (However, since the keyword static is unnecessary to convey storage type for such declarations, its meaning instead is used to restrict the scope to **file scope** – i.e., from point of declaration to the end of the file.) Since a complete C program may be produced by linking together separately compiled modules, **extern** declarations are used to alert the compiler of global symbols defined in other modules. Note that an extern declaration is just a declaration of the type of an identifier – it does not cause the allocation of any storage space and so is not a complete definition. Extern may also be used to declare functions that are used in one compilation unit, but are defined in another.

The keyword **register** is used to indicate that a machine register should be used for the storage of an automatic variable. The variable type must be scalar (e.g., an integral type or a pointer). The use of register is only a suggestion to the compiler – if there are no registers available, a register will not be used. You can tell if a register is used for a variable by trying to take the address of the variable – pointers to registers are not permitted. The register modifier is seldom used since a good optimizing compiler will make the best use of registers.

Automatic variables are kept on the execution stack (as the name suggests), and static variables in predefined locations in the memory image. Space may be allocated dynamically with the standard library function **malloc**, which takes an integer (the number of bytes to allocate) as an argument, and returns a pointer. The programmer must assign this return value to a meaningful pointer variable to access the allocated storage – interpreted as the data type corresponding to the pointer. Dynamically allocated storage persists until it is released via **free**.

Operators

C is a language rich in expressions and operators. The notation of most binary operators is typically infix, using common algebraic symbols. There are the usual arithmetic operators, both unary and binary: addition +, subtraction -, multiplication *, and division /, as well as remainder %. Characteristic of C are the unary increment and decrement operators ++ and --. These operators combine the addition (or subtraction) of one with assignment. They may appear in prefix or postfix form – in the former, the increment is performed before the value is fetched for the expression, and in postfix, the operation is performed after the value is fetched for the
expression evaluation. Arithmetic operators are overloaded since they may be applied to integral or precision types, and there are promotion rules for mixed expressions. Relation operators allow comparison for equality \(==\), inequality \(!=\), less than \(<\), less than or equal \(<=\), greater than \(>\), and greater than or equal \(>=\). Full boolean expressions may be constructed using logical and \(&&\), logical or \(||\), and logical negation \(!\). In classic C there is no boolean type, but instead the integral value zero is false, and nonzero is true. Bitwise operations may be performed on unsigned integral types using bitwise and \(\&\), bitwise or \(\|\), bitwise exclusive or \(^\^\), and bitwise complement \(~\). Other bitwise operations are shift left \(<<<\) and shift right \(>>\), both of which require an unsigned expression to shift, and an integral number of bits by which to shift. Note there is no exponentiation operator. There is one ternary operator \(op1 \, ? \, op2 \, : \, op3\), where the first operand is a Boolean expression, and if true, the value of the expression is the second operand, and if false, the value is the third operand. The most common operator is the assignment operator, a single \(=\), which has the important side effect of assigning the left hand side to have the value of the right hand side. A common error is to confuse this with the equality operator, although some compilers will warn about assignment in what most likely is the setting for an equality comparison. Many of the operators may be combined with assignment into a single compound operation. For example, \(a \, += \, b\) is equivalent to \(a \, = \, a \, + \, b\) except that the expression \(a\) is evaluated only once.

There is no explicit Boolean type in classic C. Instead, integral values were treated as truth values when needed, with zero being false and anything else being true. This means that any integral expression could be used as a condition. Most C compilers now implement type bool, although the way this is implemented (in the compiler versus in the pre-processor) may vary.

Elements of arrays are accessed by the operator \([\ ]\). This is a binary operator that combines an expression which is an array (often the name of an array variable) and an integer expression which is an offset into the array. The index expression is within the brackets and the array expression precedes the brackets. Another binary operator is the dot, which is used to select fields from a structure: a structure expression precedes the dot, and a field name follows the dot. Two other unary operators involving pointers are the asterisk, which is used to de-reference a pointer, resulting in the value in the location indicated by the pointer, and the ampersand, which evaluates to the address of the variable to which it is applied.

An important non symbolic operator is the \(\text{sizeof}\) operator. Although this often appears in code as if it is a function call, it is actually an operator and is evaluated by the compiler to be the number of bytes of the operand. The operand can be either a data type or a variable.

**C++ Additions to C**

Much of this course will be spent exploring the nuances of the C and C++ languages. Since we can mostly view C++ as a superset of C, we won’t concentrate on plain C programs. What we have described so far is the classic C programming language, and now we will briefly describe how C++ extends that language.

The main addition of C++ revolves around the keyword \textbf{class}. Syntactically a class is just like a C struct. However, the language provides a lot of support for the object oriented paradigm. Early versions of C++ were called C with Classes. Instead of a program being a collection of function definitions, all of which are global scope, C++ allows functions to be declared and defined within class definitions. This has great significance for scoping. The fields of the class are the class instance data, and are available to the functions (methods) of the class. Moreover, a method implicitly is called for a data (object) instance. In fact, the member operator \texttt{dot} (the
symbol \texttt{)} is used to indicate this relationship, so methods “belong” to objects of classes. The keyword \texttt{this} can be used to make the reference to the instance data explicit, but is not usually necessary. Classes are just an additional type. Static modifiers in classes indicate class data rather than instance data. Special naming conventions are used for constructor methods – methods whose name is the same as the class name, and are automatically called to initialize objects. Likewise, destructors have special names – the name of the class preceded by tilde (the symbol \texttt{~}). The scope operator \texttt{::} is used to qualify the class-member relationship when it needs to be explicitly stated. We’ll see later access control with the keywords \texttt{public}, \texttt{private}, \texttt{protected}, and \texttt{friend}. And we will also see the ability to define operators for class types (really syntactic sugar for usual functional notation). C++ supports class inheritance (both single and multiple) and dynamic binding. Heap memory allocation is built into the language via the \texttt{new} and \texttt{delete} operators (heap memory allocation in C is strictly through library functions).

C++ also allows aliasing through a reference type, and read-only designation through the \texttt{const} modifier.

Later in the term we will talk about several major additions to the original C++ language: templates, exceptions, and namespaces.

Just looking at the new keywords, it may seem like C++ only adds a little to the C language, but the complexity of how these constructs may be used is great, and makes for a very powerful extension of the language.

\textbf{Compiling a C/C++ Program}

C and C++ source is typically kept in files named with the suffix \texttt{.c} (or \texttt{.cpp}), but most C/C++ compilers don’t care about file names. The compiler is really a front end for a sequence of tools, beginning with the preprocessor (more about this later). The typical organization of a program is to have a source file which leads to generated code. The source may \texttt{include} (i.e., import) other files of code, which are conventionally called header files. Header files are usually collections of declarations that cause no executable code to be generated. They are housekeeping information – type definitions, macro definitions, manifest constant definitions, function declarations, class definitions, etc – that we usually want to share among a number of source modules. The compiler comes with a set of such header files that define the interface to system calls and library routines. (They may be located in the directory /usr/include, but the location is compiler dependent.)

The default action of the compiler is to preprocess, compile, and link the generated code into a fully bound executable named \texttt{a.out}. Options allow this behavior to be changed. A common option is to use \texttt{-o} to specify an executable name other than \texttt{a.out}. The compilation phase actually produces assembly code which is assembled into machine code. Note the final result is an executable in native machine format, so no interpreter or virtual machine is needed for execution - we simply type the executable name in a command shell.

The C compiler is the command \texttt{cc}, and the C++ compiler is \texttt{CC}. Gnu versions are \texttt{gcc} and \texttt{g++}.

\textbf{Differences between C++ and Java}

Java and C++ are very similar (Java stole a lot of its syntax from C++), but there are some significant differences:
Everything in Java is in a class, but C++ can have global functions and variables. In particular, the **main** function in C++ is a global, not a class, function. There must be a global main function to produce a fully linked executable.

Access modifiers in Java are on every method and data declaration. In C++, access qualifiers stand alone and apply to everything in the rest of a block or until another access modifier changes the access type.

Method bodies may be defined within classes in C++, but may also be defined in another source module, outside of the class definition. This is where the scope qualifier comes in handy.

Java has no pointers and all non primitive types are references (stealth pointers). C++ allows pointer and non-pointer types. C++ also allows explicit reference types. In particular, this means that object variables in Java must be assigned a new'd value, where in C++ you can have objects on the stack.

Java has automatic garbage collection. C++ does not, but allows you to define class specific memory allocation and de-allocation. In particular destructors are meaningful in C++ and are predictable about when they are invoked.

Java has no multiple inheritance (but does have interfaces, which C++ does not have).

Java has no operator overloading, C++ does.

Java 1.4 has no parameterized types, but Java 1.5 has introduced generic container types. C++ has fully parameterized types, which allows powerful container classes to be defined as is done with the standard template library.

C++ allows function arguments to be defaulted.

### C Input and Output

It is easiest to think of C and C++ programs as following the Unix command model of having an input and two output channels, conventionally referred to as standard input, standard output, and standard error (diagnostic output). The C language itself has no input or output constructs. Instead IO is handled by standard library functions. The most common functions are probably the formatted input and output functions scanf and printf. These functions take an argument that is a control string using controls like %d or %s or %f for decimal integer, string, and float. There are a lot of formatting options with these control strings, as can be found on the printf and scanf manual pages. Matching the controls in the control string are locations for storing the input (for scanf) or values for outputting (for printf). Scanf requires locations specified as addresses (pointer values), where printf needs only values (although the value required for %s is the address of the first byte of a string). These functions also come in other flavors (fscanf/fprintf and sscanf/sprintf) that permit input and output from arbitrary file streams or character buffers. All input/output is buffered.

Single character IO is possible with %c in scanf/printf, but is easier with getchar and putchar (or getc and putc, or fgetc and fputc). Line oriented IO is possible with gets and puts (or fgets and fputs). Finally, raw byte IO is done with read/write (or fread/fwrite). Files are opened and closed with open and close (or fopen and fclose). The former produce an integer file descriptor, the latter a FILE object which encapsulates buffering.

### C++ Input and Output

IO in C is inherently error prone because the variable argument lists means arguments can easily be left out or of the wrong type. The standard C++ library includes mechanisms for type safe IO which takes advantage of operator overloading to be notationally convenient as well. The
input operator is $\gg$ and the output operator is $\ll$. The left hand operand to these operators are input and output stream objects. For standard input and output, there are globally defined stream objects `cin` and `cout` (and `cerr`), respectively. The right hand operand for the IO operators is a variable to receive input or an expression to be output, respectively. The IO operators are defined for the built-in types, but can also be defined for any user-defined type, allowing the programmer to encapsulate whatever formatting and error checking is appropriate for the type. There are iostream manipulators to allow things like number precision formatting. This IO is buffered, and typically should not be mixed with the buffered IO of the C library.