Lecture 01: Introduction to C++ Programming

This lecture contains:

- The Compilation Process
- Overview of C/C++
- The Structure of C++ Program
- Header Files
- Primitive Types and Variables
- Preview of Functions, Arrays, and Pointers
- Input/Output Functions
- Arithmetic Expression and Assignments
- Mathematics Functions
- Summary and Concluding Remarks

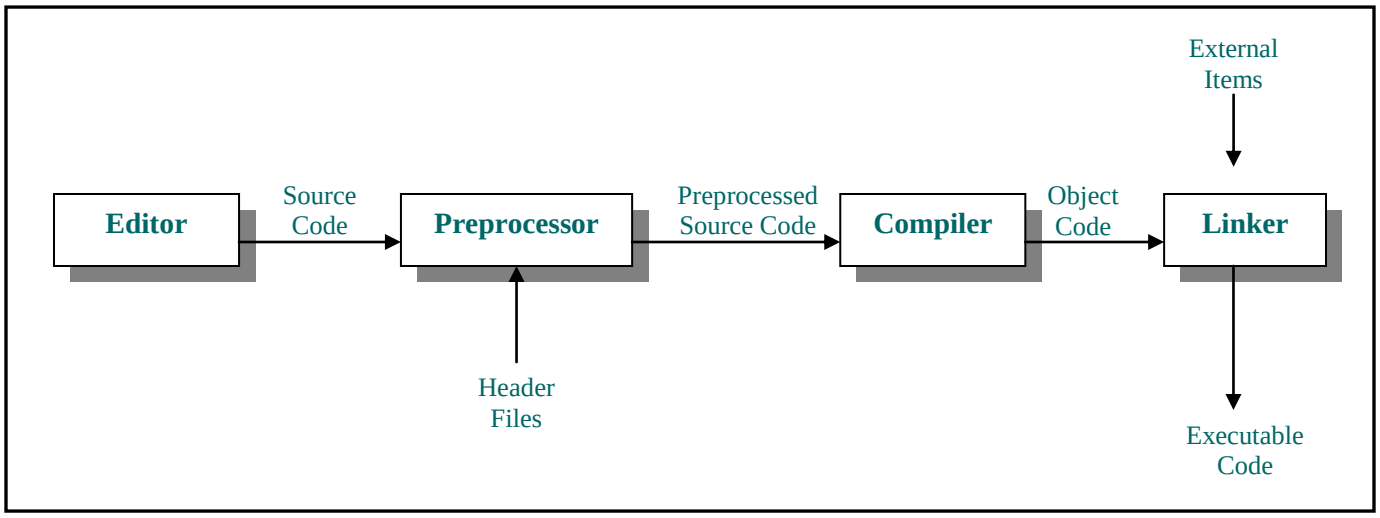
1.1 Compilation Process

Your program passes through a number of important stages before it is executed by the computer. These steps are often referred to as the compilation process and include:

1. Source code entry via the Editor
2. Preparation for compilation via the Preprocessor
3. Compilation
4. Linkage
5. Execution

Figure 1.1 illustrates the interrelated components of a programming language environment and the various processes that the program passes through.

Figure 1-1: C++ Compilation Process



Editor: The editor is a program that allows the user (programmer) to key in his/her program (source code). The editor may be a traditional line editor, or a graphical editor; this affects to a large extent, your programming environment. Typically, it provides facilities for the following:

- Entering and editing the program;
- Loading a program (from disk) into memory;
- Compiling the program;
- Debugging the program;
- Running the program

1.1 Compilation Process (continued)

Preprocessor: The preprocessor is a program that removes all comments from the source code and modifies it according to directives supplied to the program. In a C^{++} environment, a directive begins with the pound symbol (#).

Example 1-1: The following statement allows your program to access features store in the **<iostream>** header file.

```
#include <iostream>
```

Compiler: The compiler is a program that accepts as input, the preprocessed source code, analyzes it for syntax errors, and produces one of two possible outputs:

- If syntax error(s) is (are) found, an error listing is provided.
- If the program is free of syntax errors, it is converted to object code (assembler language code or machine code).

Note:

1. If the preprocessed code is converted to assembler code, an assembler then converts it to machine code.
2. Machine code varies from one (brand of) machine to the other. Each machine (brand) has an assembler. Assembler language programming is particularly useful in system programming and writing communication protocols. An assembler language is an example of a low level language.

Linker: A linker (linkage editor) is a program that combines all object code of a program with other necessary external items to form an executable program.

Programming Environment: The programming environment is composed of

- The editor;
- The preprocessor;
- The compiler;
- The linker;
- A Library of enhancement facilities that the programmer may find useful.

When you install a programming language, all these items are automatically included in a seamless manner. When you run the programming language, you are typically communicating to the editor.

1.2 Overview of C/C++

The language C was developed for the Unix operating system (by Dennis Richie and other colleagues), but has since became portable for all major operating system platforms. It was developed from specifications of earlier languages, namely BCPL (Basic Combined Programming Language) and B. (B is an enhancement of BCPL, done by Ken Thompson; C is a second enhancement of BCPL by Richie).

C is a relatively low level language, dealing mainly with characters, numbers and addresses. These are typically confined with arithmetic and logic operations, implemented by real machines (computers).

No operations are provided for dealing with composite objects such as strings, sets, lists or arrays. They are typically programmed by the programmer.

No input/output facilities and built-in file access mechanisms are available. These are also typically developed by the programmer.

Due to the foregoing points, C is usually marketed with a set of generic (add-on) functions (in a library) to aid the programmer. Further enhancements include:

- ANSI C (in the early 1980's);
- C^{++} which was developed by Bjarne Stroutstrup;
- Visual C^{++}.

C is not a strongly typed language, when compared to other high level languages (e.g. Pascal, COBOL, Fortran, etc.). It is case sensitive.

Among the outstanding advantages of C/C++ are the following:

1. **Portability:** Most operating systems will communicate with a C compiler without any problem.
2. **Flexibility:** The range of applications that C and C^{++} find relevance varies from business application development to real-time programming to communication protocol writing. C and C++ are regarded as a "middle level" languages – reaching up to situations suited for high-level languages (HLLs), and down to situations requiring interaction with machine locations.
3. **Efficiency:** The C/C++ compiler is small, concise but extremely powerful. In terms of efficiency, C out performs most rival HLLs.

1.3 The Structure of a C/C++ Program

Throughout this course, we will be using the BNF notation for representing the syntax of C++ programming language commands. The symbols used are shown in figure 1.2

Figure 1-2: BNF Notation Symbols

Symbol	Meaning
::=	"is defined as"
[. . .]	Denotes optional content (except when used for array subscripting)
<Element>	Denotes that the content is supplied by the programmer and/or is non-terminal
|	Indicates choice (either or)
{<Element>}	Denotes zero or more repetitions
<Element>*	Alternate notation to denote zero or more repetitions
<l>*<m><Element>	Denotes l to m repetitions of the specified element
[* <Element> *]	Alternate and recommended notation to denote zero or more repetitions for this course

Note:
1. The construct {<Element>} is the original construct for repetition. However, C-based languages use the left curly brace ({) and right curly brace (}) as part of their syntax. To avoid confusion, it has been recommended that for these languages, the construct <l>*<m> <Element> or <Element>* be used. But that too is potentially confusing. Therefore, for this course, we will sometimes use the construct [* <Element> *] to denote zero or more repetitions.
2. For C/C++, there are also exceptions with respect to the square braces and the angular braces; these will be pointed out at the appropriate time.

A C++ program consists essentially of functions, which in turn are made up of declarations and instructions. Logic control is initiated through a special mandatory function called **main**. Every C/C++ program must contain such a function.

Figure 1-3: C++ Program Structure

C++Program ::= [<GlobalStatement> [* <GlobalStatement>*]]
<FunctionSpecification>
[* FunctionSpecification*]

Global_Statement ::= <IncludeStatement> |<DefineStatement>|
<VariableDeclaration> |<VoidStatement>|
<ConstantDeclaration>

FunctionSpecification ::= <ReturnType> <FunctionName> ([<ParameterDef>[*;<ParameterDef.>*])
{<Function-Body>}

Note:
1. **FunctionName** is either **main** or user supplied. Every program that is to be called directly from the operating system command interface must have the function called **main**.
2. **ParameterDef** is similar to a variable declaration (which will be clarified shortly);
3. **Function-Body** consists of declarations and arithmetic or logic statements.

1.3 The Structure of a C++ Program (continued)

Statements and declarations may be grouped in blocks by use of the curly braces ({…}). Each statement is terminated by a semicolon.

Note: Your C++ program takes the name of the file to which it is saved. No program name is included in the program heading. In fact, your C++ program has no heading; it typically begins with several lines of comments. A comment follows a double slash (//) or is enclosed within a slash and an asterisk.

Example 1-2: Below are two ways to write comments in your C++ program.

```
 /* This is a comment */
// So is this.
```

1.4 Header Files

The C/C++ environment provides standard header files which consist of useful (external) function(s) that you will need to use. You can also write your own header file. In fact, large, complex programs may be split into smaller component header files.

The **Include-statement** is a (preprocessor) command that instructs the programmer to load the file named when the program is completed.

IncludeStatement ::= #include [<] <Filename> [>]

Note:
1. The statement is **not** terminated with a semicolon.
2. Standard library files are specified in angular braces. Non-library files are specified in double quotations marks.

Example 1-3:

```
#include <stdio.h>          /* original I/O header */
#include <iostream.h>              /* New I/O header */
#include "Misc\Globe.h"  /* User named header */
```

The C^{++} environment provides a number of standard header files. Among the more popular ones are the following (the appendix of most textbooks will have a comprehensive list):

Figure 1-4: Common C++ Header Files

Header File	Purpose
Miscellaneous Support:	
<assert.h>	Contains diagnostic functions.
<ctype.h>	Contains character functions.
<errno.h>	Provides a declaration of **errno**, which includes non-zero values for various symbolic constants, each representing an error.
<float.h>	Contains various constants and floating point limits.
<limits.h>	Defines various symbolic constants for the primitive data types of C++.
<math.h>	Contains mathematical functions.
<stdio.h>	Standard I/O header file-containing print, scan and string functions.
<stlib.h>	Contains utility functions (integer & arithmetic, sorting, & searching, memory allocation, communication with environment).
<stdlib.h>	Contains utility functions (sting conversion).
<string.h>	Contains various string-manipulation functions and classes.
<time.h>	Contains date and time functions.
Language Support Library:	
<exception>	Relates to exception handling
<limits>	Contains compiler-imposed limits on various data types.
<new>	Contains several types and functions that pertain to memory allocation.
<typeinfo>	Contains several types associated with the type-identification operator **typeid**.
Diagnostic Library:	
<stdexcept>	Useful when defining classes us for reporting exceptions.
General Utilities Library:	
<utility>	Useful when defining templates for general use.
<functional>	Useful when defining templates for constructing function objects.
<memory>	Useful when defining memory-allocation classes, operators, or templates.
Localization:	
<locale>	Useful when defining template classes and functions for manipulating locales.
Containers, Iterators & Algorithms Libraries:	
<algorithm>	Contains several standard algorithms used in data structures and advanced programming.
<bitset>	Describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions.
<deque>	Useful in manipulating queues.
<queue>	Useful in manipulating queues.
<iterator>	Useful when defining classes, template classes, and template functions that manipulate iterators.
<list>	Useful when manipulating lists.
<stack>	Useful when manipulating stacks.
<vector>	Useful when manipulating vectors.
Other header files in this category include <set>, <map>, <unordered_map>, <unordered_set>, and <vector>.	
Standard Numeric Library:	
<complex>	Useful when manipulating complex numbers.
<numeric>	Useful in certain numeric calculations.
<valarray>	Controls a sequence of elements of type **Type** that are stored as an array, defined for performing high-speed mathematical operations.

Figure 1-4: Common C++ Header Files (continued)

Header File	Purpose
Standard I/O Library:	
<fstream>	Useful for external file manipulations
<iomanip>	Useful in defining I/O manipulations.
<ios>	Useful for I/O manipulations.
<iosfwd>	Useful when declaring forward references to template classes used in the I/O stream.
<iostream>	Useful for I/O manipulations.
<istream>	Useful when manipulating input streams.
<ostream>	Useful when manipulating output streams.
C++ Headers for Standard C Library:	
Includes header files such as <cassert>, <cctype>, <cerrno>, <cfloat>, <ciso646>, <climits>, <clocale>, <cmath>, <csetjmp>, <csignal>, <cstdio>, <cstdlib>, <cstring>, <ctime>	

1.5 Types and Variables

All variables must be declared before or at the time of usage. The syntax for variable declaration is shown in figure 1-5. Notice that a variable may be assigned an initial value at declaration, if the optional assignment operator (=) is specified ahead of an initialization expression.

Figure 1-5: BNF Representation of Syntax for C++ Variable Declaration

Variable_Declaration ::= [static | register | const] <Type> <Identifier_List>;
Type ::= [short | long | unsigned | signed] int | char | float | double | enum | bool
Identifier_List ::= <Identifier> [= <Expression>][* , <Identifier> [= <Expression>] *]

We shall revisit types and variables in lecture 2. From the definition of figure 1-5, note the following:

1. C/C++ is not a highly typed language — there are five primitive types, but variations of them could result in eleven possibilities: **long int**, **short int**, **int**, **signed int**, **unsigned int**, **signed char**, **unsigned char**, **char**, **float**, **double**, **enum**.
2. More complex data types (structures, arrays, unions, pointers, and classes) can be constructed from these primitive types.
3. A variable may be initialized at the point of declaration.

1.5 Types and Variables (continued)

An *identifier* is a name that is given to a program element. This object may be a class, an instance of a class, a method, a constant, a function, or a variable. Below are some basic rules for naming identifiers:

1. The identifier must start with a letter or an underscore, and may include other valid letters, underscores, and digits; it is highly recommended to start your identifiers with a letter.

2. There are two prominent conventions for naming identifiers, both drawn from the traditional Hungarian notation, and both of which may be adopted with appropriate modifications:
   - Option A: Begin identifiers names for variables and methods/functions with a lower-case letter; begin identifier names for classes with an upper-case letter. For multipart names, begin each part after the first part with an upper-case letter, optionally using the underscore to improve readability. Use upper-case letters and optional underscores for constants.
   - Option B: Begin identifiers names for all declared objects with an upper-case letter, giving exception only to single-letter variable names (which should be in lower-case). For multipart names, begin each part with an upper-case letter, optionally using the underscore to improve readability. Use upper-case letters and optional underscores for constants.

3. Identifiers within a programming block must be unique; this is necessary for them to be correctly represented in the executing program.

4. The identifier cannot be a C++ reserve word. The C++ compiler will not accept such declarations, and even if it did, that would violate the principle of uniqueness, and thus create confusion.

5. The identifier may be of any length; some compilers have an upper ceiling of 2048 bytes but for practical reasons, you obviously do not need to concern yourself with this.

In C/C++, characters are simply integers, so character variables and constants are identical to integers in arithmetic expressions. A character in single quotes represents an integer value equal to the numeric value of the character. Thus,

Count = *'0';*

represents a valid integer expression as well as assignment statement. A character in double quotes denotes the actual ASCII character.

Example 1-4: The following are examples of variable declarations.

```
// The following declarations are based on option A of guideline 2 above
int thisDate;
const int BASE_DATE = 190101; // This is a constant
…
double thisSalary;

// The following declarations are based on option B of guideline 2 above
int ThisDate;
const int BASE_DATE = 190101; // This is a constant
…
double ThisSalary;
```

1.6 Preview of Functions, Arrays, and Pointers

Functions, arrays, and pointers will be discussed in subsequent lectures. However, due to their importance in C++ programming, a brief preview is required for each topic.

1.6.1 Functions

From the definition of a C++ program in section 1.3, it is apparent that mastery of functions is essential to mastery of the C++ language. Very soon, you will be writing your own C++ functions. However, even before you get to that, you will need to acquire knowledge of certain built-in C++ functions that are shipped with the C++ compiler, in order to write meaningful programs in the language. These will be introduced throughout the course. For now, you need to know what a function is, and how it will often be represented in the course.

A function is a division of a C++ program that carries out a specific task or set of related tasks. The function may return data after execution; it may also have *parameters*. If the function has parameter(s), it must be called with *argument(s)* corresponding to its parameter(s); the argument(s) is (are) automatically copied to parameter when the function is called. As shown in figure 1.3, a function specification has the following syntactic structure:

```
<ReturnType> <Function_Name> ([<ParmDef>[*,<ParmDef.>*]) // The function heading or signature
{<Function_Body>} // The function body
```

The return-type of a function can be any valid primitive data type, or programmer-defined data type (to be discussed later in the course). If the function does not return a value, the return-type must be specified as **void**. If the function has no parameters, it must still be specified with a pair of empty parentheses. Quite often, we refer to a function by simply stating its *signature* (i.e. its heading), since it provides an essential and useful summary of the function.

1.6.2 Arrays

Because C++ is not a strongly typed language, one of the things you will need to learn quickly, is how to construct more complex types from its primitive types. An *array* is perhaps the simplest example of this. An array is a finite list of items belonging to a particular base data type. To define a one dimensional (1D) array, the syntax is shown in figure:

Figure 1-6: Array Definition

Array_Definition ::= <Type> <Identifier> [<Expression>] [* <Expression> *]
Note: The square braces here do not denote optional content, but are part of the declaration; the expression within the square brackets must evaluate to a positive integer; each such declaration represents a dimension of the array.

1.6.2 Arrays (continued)

Example 1-5: The following statements illustrate 1-D array declaration.

```
char inLine [80];  /* Declares a string of 80 characters */
int inScore [30]; // Declares a 1D array of 30 integers
double inSale [20] [12]; Declares a 2D array possibly representing monthly sales for 20 sales-persons
```

Note:
1. Array subscripting starts at zero (0). Thus, for an array of **N** items, subscript values range from 0 to **N-1**.
2. Referencing array elements by subscript also involves the specification of literals or integer expressions within square braces.
3. Null-terminated character strings (referred to as NTS) must be declared of length 1 greater than the minimum length desired, since they are automatically terminated by the null (‘\0’) character.

Example 1-6: The following statements illustrate how array subscripting is done.

```
int inScore [30]; int x, scoreTotal;
// …
scoreTotal = scoreTotal + inScore[x];
```

1.6.3 Pointers

Like arrays, you cannot get very far in C++ without a basic understanding of *pointers*. A pointer is a reference to an address in memory where data is found (the data may be another reference to another memory location). In C++, a pointer is defined by specifying the data type, followed by an asterisk. You may use multiple asterisks to represent multiple pointers in a recursive way, but we will avoid such complications for now. Syntactically, a pointer specification may be represented as shown in figure 1-6.

Figure 1-6: Pointer Definition

```
SinglePointerDefn ::= <Type> *
DoublePointerDefn ::= <Type> **
```

As you will see later in the course, a pointer technically defines a dynamic list of items of a specified base type. Following are three simple examples:

Example 1-7: Simple pointer declarations

```
char* someName; // Here an NTS called someName is defined via pointer
int *thisList; // Defines a pointer to a possible list of integers
int **thisListRef; // Defines a pointer to a pointer of integers
```

1.7 Input and/or Output Functions

Since input and output statements are not part of standard C/C++, these facilities are provided through add-on functions included in the standard library. Some of these functions are fairly straightforward, but unfortunately, there are a few complex (but more powerful) alternatives. We will start with the simple ones here.

1.7.1 The cout Function

The **cout** (for character/console output) function is perhaps the simplest way to print output. The syntax for usage is shown in figure 1-7.

Figure 1-7: Syntax for using the cout Function

cout_Statement ::= cout << <outputString> [* << <String> *];
Note: The **cout** function works with the shift-left or insertion operator (<<) to redirect output from a file stream to the screen. In the basic form, the function works with each string specified after an insertion operator to effect its output onto the screen.

The output string supplied with **cout** may be a literal, variable, or reserve word. The **cout** function is part of the header file **<iostream.h>**.

Example 1-8: Following is the essential code for the famous hello-world program.

```
// The famous hello world program
#include <iostream.h>
void main ( )
{
 cout << “Hello world! I am here…”;
}
```

You may specify *extension characters* with the output string, or use certain keywords in the output. For instance, the reserve word, **endl,** may be used to denote end of line. Some possible extension characters are shown in figure 1-8, followed by an example.

Figure 1-8: C++ Extension Characters

\n	new line	\b	back-space	\t	horizontal tab
\”	double quote	\\	back-slash	\a	alert (bell) character
\f	form feed	\r	carriage return	\v	vertical tab
\xhh	hexadecimal number	\’	single quote	\?	Question mark
\ooo	octal number				

1.7.1 The cout Function (continued)

Example 1-9: The code samples illustrate the use of strings and extension characters with **cout**:

```
cout << "I love you sweet-heart.\n";
cout << "I would marry you again.";

// The above is equivalent to
cout << "I love you sweet-heart.\n" << "I would marry you again.";

// Also is equivalent to
cout << "I love you sweet-heart." << endl << "I would marry you again.";
```

The **cout** function also works with numeric data, variables and constants. No quotation marks are required when these items are supplied. This will become clear with additional subsequent examples.

Example 1-10: The code samples illustrate the use of numeric literals with **cout**:

```
cout << "8 is greater than 6\n";

// The above is equivalent to
cout <<  8 << "is greater than" << 6 << endl;
```

1.7.2 The cin Function

The **cin** (for character/console input) function instructs the computer to read a keyboard entry into a specified variable. Like **cout**, it is part of the **<iostream.h>** header file. The syntax for usage is shown in figure 1-9, followed by an example.

Figure 1-9: Syntax for using the cout Function

cin_Statement ::= cin >> <inputSeam> [* >> <inputStream> *];

Note: The **cin** function works with the shift-right or extraction operator (>>) to redirect output from a file stream to the specified variable. In the basic form, the function works with each variable specified after an extraction operator to direct input from the keyboard to the variable specified after the extraction operator.

Example 1-11: The following code snippet illustrates how data may be read from the keyboard into specified variables.

```
#include <iostream.h>
…
int myAge, yourAge;
…
cin >> yourAge >> myAge;
cout << "Your age is " << yourAge <<endl;
cout << "My age is " << myAge << endl;
```

1.7.3 The getchar and putchar Functions

The **getchar** and **putchar** functions are particularly useful when working with text files (which we will discuss later). They are both part of the **<stdio.h>** header file.

The **getchar** function returns to the calling statement, the next input character, or the symbolic constant EOF (end of file). It is called by including it in an expression (or assignment statement). It requires no argument.

Example 1-12: Below is a simple example using the **getchar** function.

```
char thisChar;
thisChar = getchar( );  // Reads a character into thisChar
```

If a program **ProgA** uses **getcha**r to obtain its input, then the source of the input may be redirected to come from a specified (text) file. The required construct is:

```
ProgA < <iputFile>;
```

Example 1-13:

```
ProgA < myFile;
```

The **putchar** function is opposite to the **getchar** function; it outputs a character to the standard output (which by default is the screen). Its syntax for usage is simply

```
putchar();
```

Redirection of output is possible: if a program **ProgB** uses **putchar** to output, then the construct

```
ProgB > <outputFile>;
```

causes output to be redirected to the file named.

Example 1-14:

```
ProgB > MyFile;
```

1.7.4 The printf and sprintf Functions

The **printf or sprintf** function is one way to print output (the latter is used if you are preparing output for a formatted string). These functions are found in the header file <**stdio.h**>. Either function is called with a variable number of arguments. The syntax for usage is summarized in figure 1-10.

Figure 1-10: Syntax for the printf or sprint Function

print_Statement ::= printf | sprintf ([<string>] [, <Format>] [, <Variable> [* , <Variable> *]]);
Format ::= "% [-] [n] | [n.m] [*] <Conversion_Character>"
Conversion_Character ::= d | i | o | X | x | c | s | f | e | E | g | G | p | h | l

Note:
1. Arguments supplied must be separated by a comma. The simplest form of usage is simply to specify a string only.
2. The format is specified in double quotations.
3. Any valid extension character may be specified within a string or format.
4. Here is the meaning of the elements in the conversion format:
   - % indicates the start of a format;
   - - indicates left justification;
   - n indicates width of the field (no of characters);
   - m indicates number of decimal positions;
   - * indicates that the value is computed by converting the next argument (which must be an integer).

Example 1-15: Following are two simple illustrations.

```
char* outString;
printf ( "Hello World…I am here\n"); // Prints the string on the monitor
// …
double srcAmount;
sprintf(outString, "%0.2f", srcAmount);  // Converts srcAmount to string and stores the result in outString
```

When the The **printf or sprintf** function is used with formats and variable(s), the format and variables must be positional, i.e. they must be specified in a position relative to their corresponding format specification. The variable(s) will be converted according to corresponding format specification(s). The possible conversion characters are clarified in figure 1-11.

1-11: Conversion Characters for the printf Function

Character	Type	Printed As
d,i	int	decimal number
o	int	unsigned octal without leading zero
x,x	int	unsigned hex without leading zero
u	int	unsigned decimal
c	int	character
s	char*	characters from a string until '\o' is encountered or precision is reached
f	double	floating point, default 6 decimal spaces
e,E	double	
g,G	double	
p	void*	pointer
h	int	half integer
l	int	long integer

Example 1-16: Following are three additional illustrations.

```
printf ("Values are:", "% 6.2f;  %6.3f", value1, value2);
        // prints for example, values are: 621.44; 4.121
….
printf ("% 10s : %30s", "My Name: ", myName); // prints for example, My Name : Elvis Foster
sprintf (outName, "% 10s, %30s", "My Name: ", myName);
         // outName stores for example, My Name: Elvis Foster
```

1.7.5 The scanf, gets and sscanf Functions

The **scanf** function (opposite to the **printf** function) is more commonly used for reading formatted input. It reads characters from the standard input, interprets them according to the specification format, and stores the results in the arguments specified. The syntax for calling **scanf** appears in figure 1-12. The function returns an integer indicating the number of arguments that were assigned values.

Figure 1-12: Syntax for the scanf or sprint Function

scanf_Statement ::= scanf (<Format>, <Variable> [* , <Variable> *]);
Format ::= "% [-] [n] | [n.m] [*] <Conversion_Character>"
Conversion_Character ::= d | i | o | X | x | c | s | f | e | E | g | G | p | h | l

Note:
1. The format is specified in double quotations.
2. Any valid extension character may be specified within a string or format.
3. Here is the meaning of the elements in the conversion format:
   - % indicates the start of a format;
   - \- indicates left justification;
   - n indicates width of the field (no of characters);
   - m indicates number of decimal positions;
   - * indicates that the value is computed by converting the next argument (which must be an integer).

1.7.5 The scanf, gets and sscanf Functions (continued)

The **scanf** function stops when the input is exhausted, or some input argument fails to match the format specification. Like **printf**, the arguments must be positional, relative to the format specification. It returns the number of successfully matched and assigned inputs; at end of file, EOF is returned.

Arguments to **scanf** must be pointers; this means variables must be prefixed by the & (address of) operator, arrays being an exception to this rule (because an array is already a pointer).

Example 1-17: In the following example, a floating point input is read into variable **thisValue**.

```
float thisValue;
scanf ("%6.2f", &thisValue); // keyboard entry is read into thisValue
```

The **scanf** function skips over white spaces (blank, tab, new line, carriage return, vertical tab, horizontal tab) to the next input.

Basic conversion characters are included in figure 1-13. You will notice that they are similar to the characters for the **printf and sprint** functions.

Figure 1-13: Conversion Characters for the scanf Function

Character	Input Data	Argument Type	Comment
d	decimal integer	int *	
i	integer	int *	
o	octal integer	int *	octal or hex
u	unsigned integer	unsigned int *	
x	hex integer	int *	
c	character	char *	
s	string	char *	
e,f,g	floating point	float *	

Exceptions:
1. Conversions d, i, o, u, x may be preceded by h (half length) or l (long).
2. Conversions e, f, g may be preceded by l (double precision).

1.7.5 The scanf, gets and sscanf Functions (continued)

Example 1-18: Following are a few additional illustrations.

```
scanf ("%lf", &thisValue);              /* read a double floating point number into thisValue */
// ….
if scanf ("%6.2", &nextValue) == 1      // if a floating point number was read into nextValue
{
  // ….
}
```

```
// To read data in the format YYYYMMDD :

int year, month, day;
// ….
scanf (" % 4d  %2d  %2d ", &year, &month, &day);
```

The **gets** (get string) function is an enhancement of the **getchar** function. It reads a string of characters. The syntax for usage is shown in figure 1-14:

Figure 1-14: Syntax for using the gets Function

```
gets_Statement ::= gets (<stringVariable>);
```

Note: Reads the keyboard entry into the specified variable. The function does not check for boundary limits, so if the input string is longer than the argument, memory overflowing occurs.

Example 1-19: The following code reads a line of keyboard entry into a specified variable.

```
char   inputLine[80];
// …
gets (inputLine)      // Equivalent to scanf ("%s", inputLine) */
```

The **sscanf** (string scan) function reads from a string (instead of the standard input) and stores the results in the arguments specified. The function returns an integer indicating the number of arguments that were assigned values. A simplified signature of the function is shown below.

```
int sscanf(const char * inputS,  const char *  formatS char ** resultVars)
```

The syntax for using the **sscanf** function is shown in figure 1-15.

Figure 1-15: Syntax for using the sscanf Function

sscanf_Statement ::= sscanf (<stringVariable>, <format>, <variable> [* ,<variable> *]);
Note: Uses **stringVariable** as the source; extracts substrings into other specified variables based on the specified format.

Example 1-20: In the code snippet below, three variables representing the date (YYYYMMDD) are updated from an input string.

```
char   inputLine[80]; int year, month, day;
gets (inputLine);
if sscanf ((inputLine, "%4d    %2d  %2d",  &year, &month, &day) == 3)
        printf ("Valid: %s\n", inputLine);
else    printf ("Invalid: %s\n", inputLine);
```

1.7.6 Treatment of Text

In C++, text (input and output) is treated as a stream of characters, divided into lines. Each line consists of zero or more characters, followed by the new line ('\n') character. Each character string is automatically terminated by the null ('\0') character.

The functions **getchar**, **putchar**, **gets, scanf, scanf, printf, sprintf**are useful for handling text, but as illustrated in the forgoing sections, there are alternate ways.

1.8 Arithmetic Expressions and Assignments

An expression is a literal or combination of variables (identifiers), operators, and literals. There are three kinds of expressions — arithmetic expressions, expressions using bit-wise operators, and Boolean expressions. We will deal with the first two here; a discussion of Boolean expressions will come later in the course (in lecture 3).

1.8.1 Arithmetic Expressions

Figure 1-16 provides the syntax for an arithmetic expression, along with some accompanying clarifications.

Figure 1-16: Arithmetic Expression

ArithmeticExpression ::= <Literal> | <Variable> | <ShortcutExpression> | [<IncDecOpr> <Variable>]
| [<Variable> <IncDecOpr>] | [<ArithExpression> <ArithOperator> <ArithExpression>]

ShortcutExpression ::= <Variable> <Operator> = <Expression>

ArithOperator ::= + | - | * | / | % | = **IncDecOper** ::= ++ | --

Meaning of the Basic Arithmetic Operations

+	Addition
-	Subtraction or negation
*	Multiplication
/	Division
%	Modulus (remainder)
++	Increment by 1 (may be prefix or postfix); e.g. counter++ (increment after use); ++counter (increment before use)
--	Decrement by 1 (may be prefix or postfix); e.g. counter-- (decrement after use); --counter (decrement before use)
=	Assignment. Thus counter = counter+1 is equivalent to ++ counter or counter++

As in other imperative and/or object-oriented programming languages, expressions are the building blocks for other statements. As such, the following general rules apply:

1. An assignment can appear as part of a larger expression. This strategy is often used to shorten your program code.
2. Parentheses when used, take higher precedence than any other operator.

C++ is at the forefront of many programming languages when it comes to the matter of shortcut expressions; this issue deserves a bit of clarification; here is the general rule:

<Variable> = <Variable> <Operator> <Expression>
may be shortened to
<Variable> <Operator> = <Expression>

Example 1-21: Following are some illustrations of shortcut expressions.

```
x = x + y;    /* is equivalent to */    x += y;
x = x – y;    /* is equivalent to */    x -= y;
x = x * y;    /* is equivalent to */    x *= y;
x = x % y;    /* is equivalent to */    x %= y;
x = x / y;    /* is equivalent to */    x /= y;
```

1.8.2 Expressions Using Bit-wise Operators

Bit-wise operators can only be applied to integral operands (i.e. data of type **int**, **char**, **short** or **long**) whether signed or unsigned. Bit-wise operators are useful for low-level programming such as device or signal control and data encryption. Figure 1-17 provides a list of common bit-wise C++ operators.

Figure 1-17: Bit-wise Operators

&	Bitwise AND
|	Bit-wise OR (inclusive)
^	Bit-wise XOR
<<	Shift left
>>	Shift right
~	One's compliment

The shift operators shift bits to the left or right and replace the positions with zeroes. They are used a lot in matters relating to device control. The shift operators are binary; the compliment operator is unary.

Example 1-22: The following code snippet illustrates how the shift and compliment operators work.

```
char thisString[8] = "00100101";  // Let thisString have the value 00100101

// Assuming the above, the following operations apply:
```

Operation	Result in value of ThisChar
thisString <<4	01010000
thisString >>4	00000010
~ thisString	11011010

Note: Do not confuse the shift operators with the BNF convention re the use of angular braces, or the requirement for specification of header files. Also, do not confuse the OR operator with the BNF convention re choice-related specifications.

The operators AND, OR, and XOR (&, |, and ^) operators perform on corresponding bits in each operand; they are binary operators.

Example 1-23: The following code snippet illustrates how the AND, OR, and XOR operators work.

```
char thisString[8] = "00010010";  // Let thisString have the value 00010010
char otherString[8] = "10010001";  // Let otherString have the value 10010001

// Assuming the above, the following operations apply:
```

Operation	Result
thisString & otherString	00010000
thisString | otherString	10010011
thisString ^ otherString	10000011

Several of the commonly used operators of C++ have been introduced so far. There are several others, many of which will be covered as we advance through the course. All C++ operators follow an operator precedence protocol as summarized in figure 1-18 (see [ECUP]).

Figure 1-18: C++ Operator Precedence Schedule

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ --	Suffix/postfix increment and decrement	
	type() type{}	Function-style type cast	
	()	Function call	
	[ ]	Array subscripting	
	.	Element selection by reference	
	→	Element selection through pointer	
3	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	C-style type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	new, new[ ]	Dynamic memory allocation	
	delete, delete[ ]	Dynamic memory deallocation	
4	.* →*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?:	Ternary conditional	Right-to-left
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

1.9 Mathematical Functions

The header file **<math.h>** contains a number of useful mathematical functions. Figure 1-19 shows some of the more commonly used functions from this header file. In most cases the argument is a floating point variable or expression.

1-19: Commonly Used Components of the <math.h> Header File

```
// Commonly Used Functions of the <math.h> Header File
cos (x)         returns the cosine of x
sin (x)         returns the sine of x
tan(x)          returns the tangent of  x
cosh (x)        returns the cosh of x (hyperbolic cosine function)
sinh (x)        returns the sinh of x (hyperbolic)
tanh (x)        returns the tanh of x (hyperbolic)
asin (x)        returns sin-1 (x)
atan (x)        returns tan-1 (x)
atanz (y,x)     returns tan-1 (y/x)
log (x)         returns ln (x)
log10 (x)       returns log10 (x)
exp (x)         returns ex
pow (x,y)       returns xy
sqrt (x)        returns square root of x
fabs (x)        returns absolute value of the floating point x
abs (i)         returns absolute value of integer i
labs (l)        returns absolute value of long integer l
```

```
// Commonly used Constants of the <math.h> Header File
M_E:            Base of natural logarithms (e)
M_LOG2E         Base 2 logarithm of e
M_LOG10E:       Base 10 logarithm of e
M_LN2:          Natural logarithm of 2
M_LN10:         Natural logarithm of 10
M_PI:           Pi, the ratio of the circumference of a circle to its diameter
M_PI_2:         Value of pi divided by 2
M_PI_4:         Value of pi divided by 4
M_1_PI:         Value of 1 divided by pi
M_2_PI:         Value of 2 divided by pi
M_2_SQRTPI:     Value of 2 divided by the positive square root of pi
M_SQRT2:        Positive square root of 2
M_SQRT1_2:      Positive square root of 1/2
```

1.10 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture:

- A typical C++ programming environment consists of an editor, a preprocessor, the C++ compiler, the linker, and a library of enhancement facilities that the programmer may find useful.
- The main advantages that C++ provides are portability, flexibility, and efficiency.
- A C++ program consists of functions. There is usually a special function called **main**.
- A typical C++ program makes use of several header files. These files contain important functions, and are shipped with the language.
- C++ is a primitive but very powerful language. The primitive data types supported are **long int**, **short int**, **int**, **signed int**, **unsigned int**, **signed char**, **unsigned char**, **char**, **float**, **double**, and **enum**.
- C++ relies heavily on the use of arrays and pointers.
- C++ provides a number of functions for handing input and output. The more commonly used ones are **cout**, **cin**, **gets**, **getchar**, **putchar**, **printf**, **sprintf**, **scanf**, and **sscanf**.
- C++ Supports three types of expressions: arithmetic, bitwise, and Boolean. An arithmetic expressions relates to manipulation of variables to yield a final result. A bit-wise expression is similar to an arithmetic expression, but it acts on bits contained in the operand(s). A Boolean expression evaluates to true of false.
- The **<math.h>** header file contains a number of useful mathematical functions.

The upcoming lecture focuses some more on declaration and manipulation of variables in C++.

1.11 Recommended Readings

[EPUP] École Polytechnique University Paris-Saclay. "C++ Operator Precedence." Accessed July 2017. http://www.enseignement.polytechnique.fr/informatique/INF478/docs/Cpp/en/cpp/language/operator_precedence.html

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapters 0 – 2.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapters 1 – 3.

[Kernighan & Richie 1988] Kernighan, Brian W. & Dennis M. Richie. 1988. *The C Programming Language*. Boston: Prentice Hall. See chapters 1 & 2.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 1.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapters 1 & 2.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapters 1 & 2.