C++ Basics

Lecture 3
COP 3014 Spring 2017

January 12, 2017

Structure of a C++ Program

- Sequence of statements, typically grouped into functions.
  - function: a subprogram. a section of a program performing a specific task.
  - Every function body is defined inside a block.
- For a C++ executable, exactly one function called main()
- Can consist of multiple files and typically use libraries.
- **Statement**: smallest complete executable unit of a program.
  - Declaration statement
  - Execution statement
  - Compound statement – any set of statements enclosed in set braces { } (often called a block)
  - Simple C++ statments end with a semi-colon. (A block does not typically need a semi-colon after it, except in special circumstances).

Libraries

- Usually pre-compiled code available to the programmer to perform common tasks
- Compilers come with many libraries. Some are standard for all compilers, and some may be system specific.
- Two parts
  - Interface: header file, which contains names and declarations of items available for use
  - Implementation: pre-compiled definitions, or implementation code. In a separate file, location known to compiler
- Use the #include directive to make a library part of a program (satisfies declare-before-use rule)

Building and Running a C++ Program

- Starts with source code, like the first sample program
- Pre-processing
  - The #include directive is an example of a pre-processor directive (anything starting with #).
  - #include <iostream>tells the preprocessor to copy the standard I/O stream library header file into the program
- Compiling
  - Syntax checking, translation of source code into object code (i.e. machine language). Not yet an executable program.
- Linking
  - Puts together any object code files that make up a program, as well as attaching pre-compiled library implementation code (like the standard I/O library implementation, in this example)
  - End result is a final target – like an executable program
- Run it!

Typical Code Elements

- Comments - Ignored by the Compiler
- Directives - For preprocessing
- Literals - Hardcoded values. g: 10
- Keywords - Words with special meaning to the compiler. Eg: int
- Identifiers - Names for variables, functions, etc.
- Operators - Symbols that perform certain operations. eg: +

Comments

- Comments are for documenting programs. They are ignored by the compiler.
- Block style (like C)

```
/* This is a comment.
It can span multiple lines */
```

- Line comments – use the double-slash //

```
int x; // This is a comment
x = 3; // This is a comment
```

Data Types

Atomic data types are the built-in types defined by the C++ language.

- **bool**: has two possible values, `true` or `false`
- integer types
  - **char** - 1 byte on most systems.
    - Typically used for representing characters
    - Stored with an integer code underneath (ASCII on most computers today)
  - **short** - (usually at least 2 bytes)
  - **int** - (4 bytes on most systems)
  - **long** - (usually 4 or more bytes)
  - The integer types have regular and unsigned versions
- floating point types - for storage of decimal numbers (i.e. a fractional part after the decimal)
  - **float**
  - **double**
  - **long double**

Identifiers

Identifiers are the names for things (variables, functions, etc) in the language. Some identifiers are built-in, and others can be created by the programmer.

- User-defined identifiers can consist of letters, digits, and underscores
- Must start with a non-digit
- Identifiers are case sensitive (count and Count are different variables)
- Reserved words (keywords) cannot be used as identifiers

Style Conventions for Identifiers

- Don't re-use common identifiers from standard libraries (like cout, cin)
- Start names with a letter, not an underscore. System identifiers and symbols in preprocessor directives often start with the underscore.
- Pick meaningful identifiers – self-documenting

```
numStudents, firstName        // good
a, ns, fn                     // bad
```

- a couple common conventions for multiple word identifiers
  - `numberOfMathStudents`
  - `number_of_math_students`

Declaring Variables

- **Declare Before Use**: Variables must be declared before they can be used in any other statements
- Declaration format:

```
typeName variableName1, variableName2, ...;

int numStudents;      // variable of type integer
double weight;        // variable of type double
char letter;          // variable of type character

//Examples of multiple variables of the same type
//in single declaration statements

int test1, test2, finalExam;
double average, gpa;
```

Initializing Variables

- To **declare** a variable is to tell the compiler it exists, and to reserve memory for it
- To **initialize** a variable is to load a value into it for the first time
- If a variable has not been initialized, it contains whatever bits are already in memory at the variable's location (i.e. a garbage value)
- One common way to initialize variables is with an assignment statement. Examples:

```
int numStudents;
double weight;
char letter;

numStudents = 10;
weight = 160.35;
letter = 'A';
```

Initializing Variables

- Variables of built-in types can be declared and initialized on the same line, as well

```
int numStudents = 10;
double weight = 160.35;
char letter = 'A';

int test1 = 96, test2 = 83, finalExam = 91;
double x = 1.2, y = 2.4, z = 12.9;
```

Initializing Variables

An alternate form of initializing and declaring at once:

```
// these are equivalent to the ones above
int numStudents(10);
double weight(160.35);
char letter('A');

int test1(96), test2(83), finalExam(91);
double x(1.2), y(2.4), z(12.9);
```

Constants

- A variable can be declared to be **constant**. This means it cannot change once it's declared and initialized
- Use the keyword `const`
- MUST declare and initialize on the same line

```
const int SIZE = 10;
const double PI = 3.1415;

// this one is illegal, because it's not
// initialized on the same line
const int LIMIT; // BAD!!!
LIMIT = 20;
```

- A common convention is to name constants with all-caps (not required)

Symbolic Constants (an alternative)

- A symbolic constant is created with a preprocessor directive, #define. (This directive is also used to create macros).
- Examples:

```
#define PI 3.14159
#define DOLLAR '$'
#define MAXSTUDENTS 100
```

- The preprocessor replaces all occurrences of the symbol in code with the value following it. (like find/replace in MS Word).
- This happens before the actual compilation stage begins

Literals

- Literals are also constants. They are literal *values* written in code.
- integer literal – an actual integer number written in code (4, -10, 18)
  - If an integer literal is written with a leading 0, it's interpreted as an **octal** value (base 8).
  - If an integer literal is written with a leading 0x, it's interpreted as a **hexadecimal** value (base 16)
  - Example:

```
int x = 26;       // integer value 26
int y = 032;      // octal 32 = decimal value 26
int z = 0x1A;     // hex 1A = decimal value 26
```

More Literals

- floating point literal – an actual decimal number written in code (4.5, -12.9, 5.0)
  - These are interpreted as type *double* by standard C++ compilers
  - Can also be written in exponential (scientific) notation: (3.12e5, 1.23e-10)
- character literal – a character in single quotes: ('F', 'a', '\n')
- string literal – a string in double quotes: ("Hello", "Bye", "Wow!\n")
- boolean literals - true or false

Escape Sequences

- String and character literals can contain special escape sequences
- They represent single characters that cannot be represented with a single character from the keyboard in your code
- The backslash \is the indicator of an escape sequence. The backslash and the next character are together considered ONE item (one char)
- Some common escape sequences are listed in the table below

Escape Sequence	Meaning
\n	newline
\t	tab
\"	double quote
\'	single quote
\\	backslash

Input and Output Streams

- In C++ we use do I/O with "stream objects", which are tied to various input/output devices.
- These stream objects are predefined in the iostream library.
- **cout** – standard output stream
  - Of class type ostream (to be discussed later)
  - Usually defaults to the monitor
- **cin** – standard input stream
  - Of class type istream (to be discussed later)
  - Usually defaults to the keyboard
- **cerr** – standard error stream
  - Of class type ostream
  - Usually defaults to the monitor, but allows error messages to be directed elsewhere (like a log file) than normal output

Using Streams

- To use these streams, we need to include the **iostream** library into our programs.

```
#include <iostream>
using namespace std;
```

- The using statement tells the compiler that all uses of these names (cout, cin, etc) will come from the "standard" namespace.

Using the Output Stream

- output streams are frequently used with the **insertion operator** <<
- Format:
  `outputStreamDestination <<itemToBePrinted`
- The right side of the insertion operator can be a variable, a constant, a value, or the result of a computation or operation
- Examples:

```
cout <<‘‘Hello World"; // string literal
cout <<‘a’; // character literal
cout <<numStudents; // contents of a variable
cout <<x + y - z; // result of a computation
cerr <<‘‘Error occurred"; // string literal
printed to standard error
```

Cascading Output

- When printing multiple items, the insertion operator can be "cascaded".
- Cascading is placing another operator after an output item to insert a new output item.

```
cout <<‘‘Average = " <<avg <<‘\n’;
cout <<var1 <<‘\t’ <<var2 <<‘\t’ <<var3;
```

- We won't utilize cerr in this course. It's less common than cout esp. in intro programming, but here for completeness.

Input Streams

- input streams are frequently used with the **extraction operator** $>>$
- Format:
  `inputStreamSource >>locationToStoreData`
- The right side of the extraction operator MUST be a memory location. For now, this means a single variable!
- By default, all built-in versions of the extraction operator will ignore any leading "white-space" characters (spaces, tabs, newlines, etc)
- In case if strings, the extraction operator will keep reading until it encounters a white space character.

Examples

```
int numStudents;
cin >>numStudents; // read an integer

double weight;
cin >>weight; // read a double

cin >>'\n'; // ILLEGAL. Right side must be a
variable
cin >>x + y; // ILLEGAL. x + y is a computation, not
a variable
```

The extraction operator can be cascaded, as well:

```
int x, y;
double a;
cin >>x >>y >>a; // read two integers and a double
from input
```

Some special formatting for decimal numbers

You will need the `iomanip` library for this.

- By default, decimal (floating-point) numbers will print in standard notation while possible, using scientific notation only when the numbers are too small or too large.
- Usually, cout prints out floats only as far as needed, up to a certain preset number of decimal places (before rounding the printed result).

```
double x = 4.5, y = 12.666666666666, z = 5.0;

cout <<x;       // will likely print 4.5
cout <<y;       // will likely print 12.6667
cout <<z;        // will likely print 5
```

Magic Formula

- A special "magic formula" for controlling how many decimal places are printed:

```
cout.setf(ios::fixed); //fixed point notation

cout.setf(ios::showpoint);
// so that decimal point will always be shown

cout.precision(2);
// sets floating point types to print to 2
decimal places (or use your desired number)

cout.setf(ios::scientific);
// float types formatted in exponential notation
```

Magic Formula

- Any cout statements following these will output floating-point values in the usual notation, to 2 decimal places.

```
double x = 4.5, y = 12.666666666666, z = 5.0;

cout <<x;       // prints 4.50
cout <<y;        // prints 12.67
cout <<z;        // prints 5.00
```

- These statements use what are called stream manipulators, which are symbols defined in the iostream library as shortcuts for setting those particular formatting flags

Alternate Method

- Here's an alternate way to set the "fixed" and "showpoint" flags

```
cout <<fixed;
// uses the "fixed" stream manipulator

cout <<showpoint;
// uses the "showpoint" stream manipulator

cout <<setprecision(3); // uses the set
precision stream manipulator (you'll need the
iomanip library for this)

//The above sets precision of the value to 3
numbers.  You can change this value based on what
you need.
```