Programming in C/C++

Chapter 2: Data Types, Variables, and Numerical Output

Kristina Shroyer

Objectives

- You should be able to describe:
  - Variables and Data Types (differences Java/C++)
  - Arithmetic Operators
  - Formatting Numerical Output Using `cout`
  - Variables and Declarations
  - Common Programming Errors

Variables/Literals

- **Concept:** ***Variables*** represent storage locations in a computer's memory. The value of a variable can change while a program is running. ***Constants (literals)*** are data items whose value cannot change while a program is running (***I like the "acceptable value for a data type" definition better than this one – this one is from the book***).

- **A fundamental objective of all computer programs is to manipulate data to solve problems and get some sort of result (or answer)**
  - In order to do this there must be places in memory to store the data a program is manipulating

- All data used in a computer program are stored in and retrieved from the computer's memory unit

- It used to be that these memory locations were referenced by their memory address
  - You can think of a variable as an "interface" to RAM

Variables

- In high level programming, symbolic names called **variables** are used in place of memory addresses to reference stored data in memory

- A ***variable*** is a name a programmer uses to refer to a computer storage location (can be used to store or retrieve data from that location)
  - Symbolic names are used in place of memory addresses
    - These symbolic names are called variables
    - These variables refer to memory locations
    - **The value stored in the variable can be changed** (unlike constants/literals)
    - Simplifies programming effort

- **Each variable in a program will only store *one* particular type of data**
  - The type of data that can be stored in a variable is fixed once you define it in your program
    - Only the ***value*** each variable contains changes during the course of the program based on the statements in the program

Variables/Literals Example

- ***Part of the job of a programmer is to determine how many variables a program needs and what type of data/information each variable in a program will hold – do this before you start coding*** – declare variables at top of functions

- Example of a C++ program using a variable (Number.cpp – Example 1)

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

int main()
{
    int number;
    number = 5;

    cout << "The value of number is " << "number" << endl;
    cout << "The value of number is " << number << endl;

    number = 7;

    cout << "The value of number is " << number << endl;

    system("PAUSE");
    return 0;
}
```

Always declare all variables at the TOP of the function it is used in (this makes code reusable and readable and I WILL Be looking for it on exams). Do not just randomly declare them throughout the code as you need them – this makes code unreadable. I enforce these kind of things in this class when grading. We are in real programming now not just "making it work".

Reminder: Remember the code in a function is executed in order from top to bottom starting with the first programming statement after the `{`. The compiler cannot look ahead or jump back, it executes the statements in order one by one from top to bottom

Variables/Literals Example

- **The first line in the program is a *variable declaration*:**
  - `int number;`
  - A variable declaration tells the compiler the variable's ***name*** and the ***type of data*** it will hold
    - The compiler needs to know the name for the memory location you want to store data in and what amount of storage to set aside (the type of data determines this)
      - Remember a variable is just a name for a memory location to make it easier for the programmer to reference and use that memory location
  - This line indicates the variable's name is number and the word `int` stands for integer indicating that number will be used to hold integer values.
    - `int` is a data type (we'll talk more about them in a bit)
  - All variable declarations end with a semicolon (they are programming statements)
  - ***You must have a declaration for every variable you intend to use in a program.***
    - you must declare (allocate) a variable (memory location) before you can use it

- ***Assignment statement:*** assigns a value to a variable (this is NOT the same as MATH!..note the difference)
  - **Format**: *variable name = expression;*
  - **In the program**: `number = 5;  number = 7;`
  - The = sign in an assignment statement is an operator that copies the value on its right (5 or 7 in this case) into the variable (memory location) on the left
    - The right side of the assignment statement is an expression. Before the expression is copied into the variables on the left side of the equal sign it is evaluated.
  - An assignment statement does not print anything to the screen, it works silently behind the scenes storing a value in RAM

Review: What are the steps the computer takes when evaluating the assignment operator?

Question: What value is in `number` after we say `int number` but before we say `number = 5`?

Variables/Literals Example

- The next two lines of code in the Example Program:

```
cout << "The value of number is " << "number" << endl;
cout << "The value of number is " << number << endl;
```

- The output these two lines of code produce:

```
The value of number is number
The value of number is 5
```

  - In the first **cout** statement the string literal “number” is inserted into the output stream so the string number is printed
  - In the second **cout** statement because there are no quotation marks around it, it is the *variable* number that is inserted into the output string causing its value to be evaluated and printed
  - Remember we said the " " around a string were important, now you see why
    - "number" is what's called a string literal

- This part of the code illustrates both the string literal (constant) number and the variable number
- Let’s look at the formal definition for a literal

Remember you can't mix types of streams – so within a << use only one type

Literals/Constants

- ***Literal:*** Acceptable value for a ***data type***
  - Value explicitly identifies itself
  - Remember from your intro programming class that all data in a computer is represented by 1s and 0s, in order for data in a computer to have meaning in a program the TYPE of data MUST be identified
    - A combination of 1s and 0s for a string data type means something totally different for a numeric data type
    - Variables are going to be set up to be a certain data type and will only be able to hold data of that type….this type tells the compiler how to interpret the 1s and 0s

- A literal is a value of a specific type
  - ***Numeric Literals***
    - The numbers `2`, `0`, and `-20` are literals of type `int`
    - The numbers `2.1` and `-3.75` are literals of type `double`
    - The number `2.1f` is a literal of type `float`
      - Why the f? Because a fractional number is automatically considered to be of type double to the compiler so to make it of type float you need to add the f
  - ***String Literals***
    - The text `"Hello World!"` is a string literal
      - A string can be thought of as a sequence of characters (we'll discuss this more in a later lecture)
      - Notice the string literal is enclosed in double quotes
      - When you use a string literal in a `cout` statement the text itself is displayed
  - ***Character literals***
    - Notice the character literal is enclosed in single quotes
    - `'a'`, `'A'`, `'b'`, `'B'` are examples of character literals

Literals/Constants

- Literals are also known as **literal values** and **constants**

- An example of a non-literal value would be a value that does not display itself but that is stored and accessed using an identifier (a variable)
  - While literals are used in programs it is more common for a program to use a ***variable***
    - When you use a variable in a program you must decide what data type to associate with that variable
    - ***Variables can change during the course of the program while literals cannot***

- Example Program
  - Notice that the string literal "number" is not the only literal/constant used in the Example program
    - 5 and 7 are integer constants (literals)
      - Notice we're saving literal of type integer (int) into a variable of type int
    - "The value of the number is" is a string literal
    - "number" in the second `cout` statement is a string literal
    - 0 is an integer constant

Variables/Literals Example

- The next two lines of code in the Example Program:

```
number = 7;
cout << "The value of number is " << number << endl;
```

- The first line of code ***replaces*** the previous value stored in the variable number with a seven
  - A variable can only store one value of its declared type at a time

- The output this `cout` line of code produces:

```
The value of number is 7
```

- This part of the code illustrates that the value of a variable can be changed during the course of the program

- When put a VARIABLE in a `cout` stream the compiler finds the value in that variable and displays it as a string

- Review: So how does the compiler know we mean the variable number and not the string "number"?

Variable Declaration

- A ***variable declaration*** is a program statement that specifies the name of a variable of a given type in a program
  - `int number;`
    - is an example of a variable declaration that declares a variable with the name number that can store integers

- A variable declaration statement has the following general form
  - ***dataType variableName;***
    - A variable declaration statement always ends with a semicolon – in order to interpret data, the compiler must know where the data is (variable name) and what type it is (how to interpret it, is it an integer? a string? a character? etc.)

- A single variable declaration can specify the names of several variables:
  - `int number1, number2, number3;`
    - However for clarity in your programs it is generally better to declare each variable on a single line.
      - If I ever declare more than one variable on a line in this class it is only to save space, in a real program I would always declare each variable on a separate line

- Variables can be declared anywhere in your program but ***a variable must be declared before it can be used*** – the compiler reads statements sequentially and cannot skip ahead in the program.
  - ***It is standard practice to declare variables at the top of the function they are being used in whenever this is possible***
    - ***USE STANDARDS IN THIS CLASS – on tests not using standards means you lose points***

Variable Initialization

- When you declare a variable you can also assign an initial value to it at that time. A variable declaration that assigns an initial value to the variable is called an ***initialization***.
  - Remember as long as a variable doesn't have an initial value in it, it contains GARBAGE, NOT nothing (it is impossible for a variable to contain nothing)
- To initialize a variable when you declare it, write and equals sign after the variable declaration followed by the value you would like to initialize the variable to:
  - `int number = 1;`
  - `double value = 3.5;`
- ***IMPORTANT NOTE:*** **If you don't supply an initial value for a variable in C++, it will contain whatever garbage was in the variable's location before the program ran.**
  - ***This is not like Java where the compiler will give you an error if you fail to initialize a variable, in C++ the compiler will just use the garbage values.***
  - Whenever possible you should initialize your variables when you declare them
  - IF your variables start with initial values it will be easier to find errors when things go wrong in your program
    - The C++ compiler is much less generous in helping you find these kind of errors than the Java compiler is

Variable Declaration

- Let's look closer at the form of a ***variable declaration***
  - A variable declaration statement has the following general form
    - ***dataType variableName;***

- The second part of the variable declaration statement is the *variableName*. The variable name is a user selected variable name. **A variable name is an *identifier*** (a user defined name that represents some element of a program)
  - **To be legal a variable name must follow the rules of a legal identifier we discussed in the last chapter**
    1. An identifier must begin with a letter or underscore_
    2. Each character after the first character of an identifier must be a letter, digit, or underscore_
    3. The variable name cannot be a keyword

- The first part of the variable declaration statement is a **data type**
  - Variables are classified according to their data type which determines the type of information that may be stored in them
    - For example variables with integer data types can only hold whole numbers
  - In order to decide which data type to use for a variable you need to understand more about data types
    - We will go over this next

Data Types

- The objective of all programs is to process data
- It is necessary to classify data into specific types
  - Numerical
  - Alphabetical
  - Audio
  - Video
- C++ allows only certain operations to be performed on certain types of data
  - Prevents inappropriate programming operations
- ***Data Type:*** A set of values **AND** operations that can be applied to these values
  - ***You MUST KNOW THIS DEFINITION – it will be on the test***
  - **Example of Data Type**: Integers
    - The Values: Set of all Integer (whole) numbers
    - The Operations: Familiar mathematical and comparison operators

Data Types

- **C++ categorizes data types into two fundamental categories/groupings**

1. ***Class Data Types:***
   - Usually a programmer-created data type/requires external code defined by the programmer
     - But also can be a data type provided by the C++ library such as a `string`
   - Set of acceptable values and operations defined by a programmer using C++ code
   - The majority of the operations on class data types are user defined functions

2. ***Built-In Data Types***: Provided as an integral part of C++
   - Also known as a ***primitive*** type
   - Requires no external code
   - Consists of basic numerical types
   - Majority of operations are symbols (e.g. +,-,*,…) – Meaning most of the operations are basic mathematical operations
     - This is in contrast to class data types where the majority of the operations are provided as functions

- Again you MUST know this!

Primitive (Built in) Data Types

- In a very broad sense there is only one category of built in data type:
  Numerical Data Types
- There are two Categories of Numerical Data Types
  - **Integer Data Types**
  - **Floating Point Data Types**

TABLE 2.1 *Built-In Data Type Operations*

Built-in Data Types	Operations
Integer	+, –, *, /, %, =, ==, !=, <=, >=, sizeof(), and bit operations (see Sec. 17.4)
Floating Point	+, –, *, /, =, ==, !=, <=, >=, sizeof()

Primitive (Built in) Data Types

- **The primary considerations for *choosing which numeric data type to use* for a variable in a C++ program are**
  1. The largest and smallest numbers that may be stored by the variable
  2. How much memory the variable uses
  3. Whether the variable holds signed or unsigned numbers
  4. The number of decimal places of precision a variable has

Integer Data Types

- C++ provides eight built-in **Integer Data Types**
  - This is a lot more than in Java
- Three most important **Integer Data Types**
  - `int`
  - `char`
  - `bool` (this is similar to – but ***NOT*** exactly the same as the boolean in Java)
- Reason for remaining 5 Integer data types is historical
  - Originally provided for special situations
  - Difference among types based on storage requirements

Integer Data Types

- C++ provides eight built-in **Integer Data Types**
  - ***The values of the columns Size and Range depend on the system the program is compiled for.***
    - This is different than in Java

TABLE 2.4 *Integer Data Type Storage*

Name of Data Type	Storage Size (in bytes)	Range of Values
`char`	1	256 characters
`bool`	1	true (which is considered as any positive value) and false (which is a zero)
`short int`	2	–32,768 to +32,767
`unsigned short int`	2	0 to 65,535
`int`	4	–2,147,483,648 to +2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long int`	4	–2,147,483,648 to +2,147,483,647
`unsigned long int`	4	0 to 4,294,967,295

The `int` Data Type

- Set of values supported are whole numbers
  - Whole numbers mathematically known as integers
  - The value zero or any positive or negative numerical value without a decimal point
- An integer consists of digits only and can be optionally preceded by either a plus (+) or a minus (-) sign
- Commas, decimal points, and special signs not allowed
- **Examples of `int`:**
  - Valid: 0 5 -10 +25 1000 253 -26351 +36
  - Invalid: $255.62 2,523 3. 6,243,982 1,492.89
- Different compilers have different internal limits on the largest and smallest integer values that can be stored in each data type
  - The most common allocation for the `int`
    - 4 Bytes, which restricts the set of values represented by the `int` data type to the range -2,147,483,648 to 2,147,483,647

The `char` Data Type

- Used to store **individual (single)** characters
  - Letters of the alphabet (upper and lower case)
  - Digits 0 through 9
  - Special symbols such as + $. , - !

- **Single Character Value**: any **ONE** letter, digit or special character enclosed in single quotes
  - Examples ‘A’ ‘$’ ‘b’ ‘7’ ‘y’ ‘!’ ‘M’ ‘q’

- THE SINGLE QUOTES ARE ESSENTIAL
  - This the only way the compiler knows you are talking about a character literal and are saving the right type of data into a char type variable!

The `char` Data Type

- Character values are typically stored in a computer using either the ASCII or Unicode codes.
  - ASCII, pronounced AS-KEY, is an acronym for American Standard Code for Information Interchange.
  - The ASCII code provides codes for an English-language-based character set plus codes for printer and display control, such as new line and printer paper-eject codes. Each character code is contained within a single byte, which provides for 256 distinct codes.

- **ASCII**: American Standard Code for Information Exchange
  - Basically ASCII is just a table that tells which character is represented by which pattern of bits
  - Provides English-language based character set plus codes for printer and display control
  - Each character code contained in one byte
  - 256 distinct codes
  - Appendix B in your book shows the ASCII character codes
  - A char is a **numerical data type** because each of the 256 characters are represented by a binary code
    - These binary codes represent on thing if interpreted as a number and another if interpreted as a char
      - This can be used to the programmers advantage as we'll see later
  - Notice that each character in the ASCII table is represented by a numeric (decimal) value
    - So the character 'A' is represented by the decimal value 65
    - The following two statements both set the variable ch to the character 'A'
      - `char ch = 'A';`
      - `char ch = 65;`

- **Unicode**: Provides other language character sets
  - Each character contained in two bytes
  - Can represent 65,536 characters
  - First 256 Unicode codes have same numerical value as the 256 ASCII codes

The `char` Data Type

- Example (CharExample.cpp – Example 2)

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


int main()
{
    char ch1 = 65;  //The ASCII code 65 represents the character 'A'
    char ch2 = 'A';

    cout << "The value of ch1 is " << ch1 << endl;
    cout << "The value of ch2 is " << ch2 << endl;

    system("PAUSE");
   return 0;
}
```

The Escape Character

- **Backslash (\ ):** the escape character
  - Special meaning in C++
  - Placed before a select group of characters, it tells the compiler to escape from normal interpretation of these characters

- **Escape Sequence**: combination of a backslash and specific characters with no intervening white space which causes the compiler to create a single ASCII code
  - **Example**: newline escape sequence, `\n`

The Escape Character

- Both `'\n'` and `"\n"` represent the newline character
  - '\n' is a character literal
  - "\n" is a string literal
- Both cause the same thing to happen but are translated differently
  - A new line is forced on the output display
- In translating the '\n' the compiler translates it using the ASCII code
- In translating the "\n" the compiler translates the correct code but also adds a string termination character '\0'
- Good programming practice is to end the final output display with a newline escape sequence

Programming Tips

- It's important to under stand the difference between a character, a string, and a numerical literal

- 'A' and "A" both display the character A on an output device
  - 'A' is a character literal which represents a single character
  - "A" is a string literal. A string is an array of characters (if you don't know what an array is think of it as a sequence of characters) terminated with a null character '\0'
    - Think of an array for now as a data structure that can hold primitive data values of the same type
    - So while "A" prints the same thing to an input device as 'A' the two are not the same data type

- This also means that trying to print a string to an output device by enclosing it in single quotes instead of double quotes would cause an error 'A string literal should not go in single quotes'

- Another error would be to confuse the numeric value 5 with the string literal "5" or the character value '5'.
  - int number = '5' or int number = "5" are incorrect variable definitions because '5' and "5" are not integer literals/constants but are string and character literals/constants which are completely different data types

- You can only save data of the CORRECT TYPE into a variable
  - so an int variable can ONLY hold integer types
  - a char variable can NOT hold STRINGS
  - etc.
  - ***CAUTION:*** If you save the wrong type of data in a C++ variable, you won't always get an error – C++ will assume you know what you're doing and attempt to COVERT your data to the type of the variables – this can produce unwanted results

The bool Data Type

- Represents boolean (logical) data
- Represents true or false values
  - These true/false values are represented by integers in C++
- Often used when a program must examine a specific condition
  - If condition is true, the program takes one action, if false, it takes another action

- **The `bool` data type uses an integer storage code**
  - **Zero represents a false value and all other positive integer values represent a true value**
    - Note that this is different than in Java where the boolean data type only takes on the values of true and false
  - Note that bool can only take unsigned values

- Example (BoolExample.cpp – Example 3)

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

int main()
{
     bool boolF = 0;  //a bool data type with a value of zero is false
     bool boolT = 2;  //a bool data type with any value other than zero is true

     cout << "The value of boolF is " << boolF << endl;  //A zero value for bool represents false
     cout << "The value of boolT  " << boolT << endl;  //A 1 value for bool represents true

     system("PAUSE");
    return 0;
}
```

Note: You could replace the 0 and 2 above with the words true and false and the code "works" – HOWEVER, true is still represented behind the scenes by 1 and false by 0 – this can result in some errors later (in loops and selection)

Determining Storage Size

- C++ makes it possible to see how values are stored
  - Remember the number of bytes used to store a data type is compiler dependent
- **`sizeof()`:** provides the number of bytes required to store a value for any data type
  - Built-in operator that does not use an arithmetic symbol
- Example (SizeOf.cpp – Example 4)

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

int main()
{
  cout << "\nData Type  Bytes"
       << "\n---------  -----"
       << "\nint         " <<sizeof(int)
       << "\nchar        " <<sizeof(char)
       << "\nbool        " <<sizeof(bool)
       << '\n';

  system("PAUSE");
  return 0;
}
```

- The output of this program is compiler dependent. Each compiler will correctly report the amount of storage it provides for the data type under consideration

Signed and Unsigned Data Types

- **Signed Data Type**: stores negative, positive and zero values
- **Unsigned Data Type**: stores positive and zero values
  - Provides a range of positive values double that of unsigned counterparts
- `char` and `bool` are unsigned data types
  - No codes for storing negative values
- Some applications only use unsigned data types
  - **Example:** date applications in form *year month day*
  - For these type of applications an unsigned data type could be used
- All unsigned data types provide a range that is basically double the range for their signed counterpart
  - This extra positive range is made available by using the negative range of the data type's signed version for additional positive numbers

Signed and Unsigned Data Types

TABLE 2.4 *Integer Data Type Storage*

Name of Data Type	Storage Size (in bytes)	Range of Values
`char`	1	256 characters
`bool`	1	true (which is considered as any positive value) and false (which is a zero)
`short int`	2	–32,768 to +32,767
`unsigned short int`	2	0 to 65,535
`int`	4	–2,147,483,648 to +2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long int`	4	–2,147,483,648 to +2,147,483,647
`unsigned long int`	4	0 to 4,294,967,295

Floating-Point Types

- The other built in numerical data type (other than Integer types) are Floating-Point data types
  - Remember we are talking about BUILT IN DATA types, ones provided by the compiler rather than created and written by programmers (sometimes stored in libraries, sometimes not)

- A floating point number can be the number zero or any positive or negative number that contains a decimal point
  - Also called real number
  - Examples: +10.625 5. -6.2 3521.92 0.0
  - 5. and 0.0 are floating-point, but same values without a decimal (5, 0) would be integers
  - As with integer values special symbols such as the dollar sign or the comma are not permitted

- C++ supports three floating-point types:
  - `float, double, long double`
  - Different storage requirements for each

Floating-Point Types

- Most compilers use twice the amount of storage for a **`double`** as for a **`float`** which allows a double to have approximately twice the precision as a float
  - For this reason a float is often called a single precision number and a double a double precision number
  - The actual storage for each type does differ by compiler
    - Currently most C++ compilers allocate four bytes for a float and eight bytes for both the double and long double types
      - In compilers that allocate the same number of bytes for double and long double these types become identical (try sizeof())

- **Precision**: refers to the numerical accuracy or number of significant digits
  - **Significant digits** = number of correct digits + 1
    - Remember the last digit is rounded
- Significant digits in a number may not correspond to the number of digits displayed
  - **Example**: if 687.45678921 has five significant digits, it is only accurate to the value 687.46
    - So the significant digits are accurate to the value 687.46 and the last digit is rounded

Arithmetic Operators – Numeric Types

Operation	Operator
Addition	+
Subtraction	–
Multiplication	*
Division	/
Modulus	%

- **Operators for numerical data types (primitive/built in data types)**
  - **Remember data types are the set of values AND the operations that can be performed on those values**
- **Binary operators**: require two operands
  - The operands can be a literals or identifiers (or a literal and an identifier)
- **Binary Arithmetic Expression**: one consisting of an operator connecting two operands which can be variables or literals
  - **Format**:

```
literalValue operator literalValue
variable operator variable
variable operator literalValue
literalValue operator variable
```

Arithmetic Operators

- **Rules for evaluating arithmetic expressions**:
  1. **If both operands are integers**: result is integer
  2. **If one operand is floating-point**: result is floating-point

- **Mixed-mode Expression**: arithmetic expression containing integer and non-integer (real/floating point) operands
  - The result of a mixed mode expression is always a floating-point value (Rule 2)

- Note that the arithmetic operations for addition, subtraction, multiplication and division are implemented differently for integer and floating point arithmetic
  - This means that the type of arithmetic performed depends upon the type of operands in the expression
  - This means the arithmetic operators are overloaded
  - ***Overloaded Operator:*** a symbol that represents more than one operation – which operation is performed depends on the data type of the operands

Integer Division

- Division of two integers yields an integer
  - Remember the rule I just said: a mathematical operation on two integers results in an integer
  - Integers cannot contain a fractional part - results may seem strange
  - In C++ the fractional part of the result obtained when two integers are divided is simply dropped (**truncated**)
    - **Example**: integer 15 divided by integer 2 yields the integer result 7
    - Example (IntegerDiv.cpp – Example 5)

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

int main()
{
    int num1 = 15;
    int num2 = 2;

    cout << "Integer Division 15/2 = " << num1/num2 << endl;
    cout << "Integer Division 2/15 = " << num2/num1 << endl;

    system("PAUSE");
    return 0;
}
```

Remainder Division – Modulus %

- We may want to calculate the remainder of integer division. This is done with the modulus operator

- **Modulus Operator (%):** captures the remainder
  - Also called the **remainder operator**
  - **Example**: 9 % 4 is 1 (remainder of 9/4 is 1)

  - Example (ModulusEx.cpp – Example 6)

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

int main()
{
    int num1 = 9;
    int num2 = 4;

    cout << "Remainder Division 9%4 = " << num1%num2 << endl;
    cout << "Remainder Division 4%9 = " << num2%num1 << endl;

    system("PAUSE");
    return 0;
}
```

Question: Does modulus work on floating point numbers?

Negation

- A unary operation that negates (reverses the sign of) the operand
  - Unary refers to the fact that this arithmetic operation only takes one operand
- Uses same sign as binary subtraction (-)
- Table 2.6 in the book summarizes the arithmetic operators

Operator Precedence

- Frequently we will want to create complex arithmetic expressions in C++ that contain multiple operands and operators

- Rules for expressions with multiple operators
  - Two binary operators cannot be placed side by side
    - 5-+6
  - Parentheses may be used to form groupings
    - Expressions within parentheses are evaluated first
  - Sets of parentheses may be enclosed by other parentheses
    - Evaluate the expressions in the innermost parenthesis first and work your way outwards
    - The number of left parenthesis must always equal the number of right parenthesis
  - Parentheses cannot be used to indicate multiplication (multiplication operator (*) must be used)

Operator Precedence & Associativity

- Order of evaluation for complex arithmetic expressions
  - Evaluate expressions in parenthesis first from the innermost parenthesis to the outermost
  - What about parts of the expression not in parenthesis?
  - Evaluate parts of the expression not in parenthesis according to ***operator precedence*** (priority) rules

  - **Operator Precedence in C++:**
    1. All negations are done first
    2. Multiplication, division, and modulus operations are computed next. Expressions containing more than one multiplication, division, or modulus operator are evaluated from left to right as each operator is encountered.
    3. Addition and subtraction are computed last. Expressions containing more than one addition or subtraction are evaluated from left to right as each operator is encountered.

  - ***Operator Associativity*** refers to the order of which operators of the same precedence are evaluated. For rules 2 & 3 the associativity is left to right. For the unary – operator the associativity is right to left

Operator Precedence & Associativity

TABLE 2.7 *Operator Precedence and Associativity*

Operator	Associativity
unary –	right to left
* / %	left to right
+ –	left to right

Numerical Output Using `cout`

- `cout` allows for display of result of a numerical expression
  - Display is on standard output device

- **Example:**
  - `cout << "The total of 6 and 15 is " << (6 + 15);`
  - Statement sends two pieces of data: a string and a value to be sent to `cout`
    - **String**: `"The total of 6 and 15 is "`
    - **Value**: value of the expression 6 + 15
      - The parenthesis are not required to indicate that is the value of the expression that is being placed in the input string.
    - **Display produced**: The total of 6 and 15 is 21
  - Individually each set of data sent to `cout` is preceded by its own insertion symbol (`<<`)
  - Notice the space after the is in the string, without this the value 21 would come right after the “s”

Numerical Output Using cout

- The insertion of data onto the output stream can be made over multiple lines and is terminated only by a semicolon

- The rules for using multiple lines are that a string contained within double quotes **cannot** be split across lines and that the terminating semicolon must **only** appear on the last line
  - **Legal**

```
cout << "The total of 6 and 15 is "
     << (6 + 15);
```

  - **Illegal**

```
cout << "The total of 6 and
          15 is " << (6 + 15);
```

  - **Illegal**

```
cout << "The total of 6 and 15 is ";
     << (6 + 15);
```

  - Floating point values are displayed with significant digits to the right of the decimal place to accommodate the number
    - This is true if the number has six or fewer significant digits
    - More than six significant digits, fractional part is rounded to six significant digits
    - Zero significant digits displays no decimal point or significant digits
      - **Example:**

```
cout << "15.0 * 2.0 equals " << (15.0 * 2.0) << endl;
Output:  15.0 * 2.0 equals 30
```

      - **NOTE: In this example the output is 30 because cout formatted it that way….what data type is the 30?**
        - **ANSWER: It's still floating point even though it may appear C++ has converted it to an integer**

Formatted Output

- A program should present results attractively
- ***Field width manipulators:*** control format of numbers displayed by `cout`
  - Manipulators are included in the output stream
  - Field width manipulators are useful in printing columns so that the numbers in each column align correctly
- The `setw(n)` field manipulator sets the field width to n
  - The `setw(n)` manipulator should be place before the numeric field it is manipulating in the output stream
    - `cout << "The sum of 6 and 15 is" << setw(3) << 21;`
    - This field width setting causes the 21 to be printed in a field of three spaces which includes one blank space and the number 21

Example of Unformatted Output

Program 2.3

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

int main()
{
  cout << 6 << endl
       << 18 << endl
       << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

The output of Program 2.3 is

```
6
18
124
---
148
```

You would use the **`setw(n)`** manipulator to format this output to be right aligned

Formatted Output - Integers

- **Example (FieldWidth.cpp – Example 7)**

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

int main()
{
   cout << setw(3) << 6 << endl
        << setw(3) << 18 << endl
        << setw(3) << 124 << endl
        << "---\n"
        << 6+18+124 << endl;

   return 0;
}
```

  - To use the field width manipulators you must include the proper header file in the program which is:
    - `#include <iomanip>`
  - Output is right aligned as expected
  - **The field width manipulator must be included for each number inserted into the data stream sent to `cout` and a particular `setw` manipulator only applies to the *next* insertion of data immediately following it**
  - Could take out manipulator in front of 124 since 124 automatically already has a field width of 3

Formatted Output – Floating Point

- A formatted floating point number requires the use of three field width manipulators
  - The `setw(n)` manipulator to set the total width of the display
  - The `fixed` manipulator to force the display of the decimal point (otherwise you get scientific notation)
  - The `setprecision(n)` manipulator to determine how many digits will be displayed to the right of the decimal point
    - Numbers are rounded to this number of decimal places
  - Unlike `setw(n)`, the fixed and `setprecision(n)` field width manipulators can be used once in a `cout` statement and then will be applied to all other floating point output following their use
  - In a floating point number the decimal point counts as one field place when using `setw(n)`
  - With both floating point and integer numbers if the field set in your `setw(n)` manipulator is not large enough to hold your number the field width is automatically expanded to accommodate your number
  - Example:
    - ```
      cout << setw(10) << fixed
           << setprecision(3) << 25.67;
      ```
    - `Output: 25.670 (right aligned, spaces fill up 4 unused fields in the total field width of 10)`
  - These are the main field width manipulators we will be using but there are more you can use to make your output even more formatted or formatted in different ways (see table 2.10)
      ```

Formatted Output – Floating Point

- Example (FieldWidth2.cpp – Example 8)

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

int main()
{
   cout << fixed << setprecision(2)
        << setw(6) << 6.1 << endl
        << setw(6) << 18.2 << endl
        << setw(6) << 124.0 << endl
        << "------\n"
        << 6.1+18.2+124<< endl;

   system("PAUSE");
   return 0;
}
```

Formatted Output – Floating Point

TABLE 2.9 *Effect of Format Manipulators*

Manipulators	Number	Display	Comments
`setw(2)`	3	`| 3|`	Number fits in field
`setw(2)`	43	`|43|`	Number fits in field
`setw(2)`	143	`|143|`	Field width ignored
`setw(2)`	2.3	`|2.3|`	Field width ignored

`setw(5)`
`fixed`
`setprecision(2)`
	2.366	`| 2.37|`	Field width of 5 with 2 decimal digits

`setw(5)`
`fixed`
`setprecision(2)`
	42.3	`|42.30|`	Number fits in field with specified precision

`setw(5)` `setprecision(2)`	142.364	`|1.4e+002|`	Field width ignored and scientific notation used with the `setprecision` manipulator specifying the total number of significant digits (integer plus fractional)

Variable Declaration Statements - Review

- Names a variable, specifies its data type
  - **General form**: ***dataType variableName;***
  - **Example**: `int sum;` Declares `sum` as variable which stores an integer value

- Variable declaration statements can be placed anywhere in a function
  - Typically grouped together and placed immediately after the function's opening brace (better readability)
    - This is the best programming practice so someone reading the code can immediately see all the variables being used in the program
  - **In addition, a variable must be declared before it can be used**
  - **Also in C++ variables that are not initialized do not cause compiler errors, instead your program will run with the variable containing garbage and produce incorrect results**

Multiple Variable Declarations

- Variables with the same data type can be grouped together and declared in one statement
  - **Format**: *dataType variableList*;
  - **Example**: `double grade1, grade2, total, average;`
  - **Generally it's better to declares each variable on a separate line**

- **Initialization**: using a declaration statement to store a value in a variable
  - Good programming practice is to declare each initialized variable on a line by itself
  - **Example:** `double grade2 = 93.5;`

Common Programming Errors

- Forgetting to declare all variables used in a program
- **Attempting to store one data type in a variable declared for a different type**
- **Using a variable in an expression before the variable is assigned a value**
- Dividing integer values incorrectly Mixing data types in the same expression without clearly understanding the effect produced
  - It is best not to mix data types in an expression unless a specific result is desired
- Forgetting to separate individual data streams passed to `cout` with an insertion (<<) symbol

Summary

- Four basic types of data recognized by C++:
  - Integer, floating-point, character, boolean
- `cout` object can be used to display all data types
- Every variable in a C++ program must be declared as the type of variable it can store
- Reference variables can be declared that associate a second name to an existing variable
- A simple C++ program containing declaration statements has the format:

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

int main()
{
  declaration statements;

  other statements;

  system("PAUSE");
  return 0;
}
```

Summary (continued)

- **Declaration Statements**: inform the compiler of function's valid variable names
- **Definition Statements**: declaration statements that also cause computer to set aside memory locations for a variable
- **`sizeof()` operator**: determines the amount of storage reserved for a variable