Numeric Data Types

- **Integral Types** - Refer to integer values (or whole numbers)
  - `char`, `short`, `int` and `long`
  - Integer values can be negative or positive unless they are explicitly defined as `unsigned`, in which case they can only be zero or positive
  - The above DataTypes represent the different sizes of integers in terms of the amount of memory they require
    - Typically *(results are from the lab machine)*
      - `char` is 8 bits (1 byte)
      - `short` is 16 bits (2 bytes) (-32,768 through 32,767)
      - `int` is 32 bits (4 bytes) (-2,147,483,648 through 2,147,483,647)
      - `long` is 64 bits (8 bytes) (-2^{63} + 1 through 2^{63})
More Numeric Types

- **Floating-Point Types** – refers to decimal numbers
  - `float`, `double` and `long double`
  - Floating-point types are used to represent real numbers and have both an integer part and a fractional part. They can also have exponents.
  - The above DataTypes represent the different sizes of floating-point types in terms of the amount of memory they require
    Typically *(results are from blackhawk)*
    - `float` is 4 bytes
    - `double` is 8 bytes
    - `long double` is 16 bytes

Declarations for Numeric Types

- **Named Constant Declarations** - using named constants makes it easier to modify programs in the future. It is also a way to shorten typing required and provide meaning (for example PI)
  - `const float PI = 3.14159;`
  - `const int MINIMUM = -459; // Absolute zero`
  - `const double E = 2.71828; // The constant e`

- **Variable Declarations**
  - `int class_size = 100; // # of students in a class`
  - `float miles; // number of miles driven`
  - `double time; // amount of time to solve the problem`
  - Appropriate assignment statements:
    - `miles = 345.6; class_size = 120;`
  - **See Program DataTypes2_01.cpp for some examples**
Arithmetic Operators

• To indicate the sign of a number - positive or negative
  • + Unary plus (one operand, rarely used):  +456  +56.77
  • - Unary minus (one operand):  -54   -90.383

• Math Operators
  • + Addition (two operands):  5 + 8 \(\Rightarrow\) 13  6.3 + 7.8 \(\Rightarrow\) 14.1
  • - Subtraction (two operands): 5 - 8 \(\Rightarrow\) -3  6.3 - 7.8 \(\Rightarrow\) -1.5
  • * Multiplication (two operands): 4*5 \(\Rightarrow\) 20  5.6*8.3 \(\Rightarrow\) 46.48
  • / Floating point division (floating point result) 16.0/5.0 \(\Rightarrow\) 3.2
  • Integer division (no fractional part) 16/4 \(\Rightarrow\) 4  16/5 \(\Rightarrow\) 3
  • % Modulus (remainder from integer division)
    \[ \begin{array}{ccc}
    8/2 & \Rightarrow & 4 \text{ (remainder 0)} \\
    8\%2 & \Rightarrow & 0 \\
    8/5 & \Rightarrow & 1 \text{ (remainder 3)} \\
    8\%5 & \Rightarrow & 3 \\
    \end{array} \]

1 Here the symbol “\(\Rightarrow\)” is meant to be read as “gives a result of” or “has a value of”, and it is not a C++ operation.

Assignment Statements

• Given:

```c
float alpha, beta;
int num, count;
```

• Valid Assignments:

```c
beta = -6.2;
alpha = 6.5*beta;
alpha = alpha + beta*alpha;
um = count - 5;
um = num - count;
alpha = beta = 3.14159;
```

• Invalid Assignment: `alpha + beta = 34.5;`
Increment and decrement operators

- ++ designates increment
- -- designates decrement
- Can be either prefix or postfix operators.
  - As a prefix operator, the increment or decrement occurs BEFORE assignment
  - As a postfix operator, the increment or decrement occurs AFTER assignment

SYNTAX TEMPLATES:

<table>
<thead>
<tr>
<th>IncrementStatement</th>
<th>DecrementStatement</th>
</tr>
</thead>
<tbody>
<tr>
<td>postfix</td>
<td>prefix</td>
</tr>
<tr>
<td>Variable++;</td>
<td>++Variable;</td>
</tr>
<tr>
<td>Variable--;</td>
<td>--Variable;</td>
</tr>
</tbody>
</table>

num++; is equivalent to num = num + 1;
num--; is equivalent to num = num - 1;

See Program DataTypes2_02.cpp for some examples

Additional C++ Operators (from Chapter 7)

- Look at pages 323
- Combined assignment operators: +=, -=, *= and /= are add and assign, subtract and assign, multiply and assign and divide and assign respectively

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>j *= 5;</td>
<td>j = j * 5;</td>
</tr>
<tr>
<td>k -= 3;</td>
<td>k = k - 3;</td>
</tr>
</tbody>
</table>
Compound Arithmetic Expressions

- **Arithmetic expressions** can contain several **constants, variables, operators and parentheses**.
- **Have a precedence order**:
  - Highest level (performed first): `( )`
  - Upper level: Unary `+`, Unary `-`
  - Middle level: `*`, `/`, `%`
  - Low level: `+`, `-`
  - Lowest level: `=`, `+=`, `-=`, `*=`, `/=`
  - Last: `=`
- Execution is performed from highest level to lowest level
- Can use parentheses to change the order of evaluation
- Expressions involving several binary operators with the same precedence have a grouping order from left to right

Precedence Examples – Example 1

<table>
<thead>
<tr>
<th><strong>Arithmetic Expression</strong></th>
<th><strong>Value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>10 + 5 / 5 * 8</code></td>
<td>18</td>
</tr>
<tr>
<td><code>14 % 4 * 5 - 6 / 2</code></td>
<td>7</td>
</tr>
<tr>
<td><code>3 * 4 / 2 * 3</code></td>
<td>18</td>
</tr>
<tr>
<td><code>3 * 4 / (2 * 3)</code></td>
<td>2</td>
</tr>
<tr>
<td><code>(5 + 10) / 3 % 5</code></td>
<td>0</td>
</tr>
</tbody>
</table>

Can always use parentheses to make sure the desired evaluation is made
Type Coercion (Implicit Conversion)

- **Implicit (automatic) type coercion** is the automatic conversion of a value from one DataType to another.

- Most times entire expressions are involved in type coercion
  - Conversion from an `int` to a `float` **USUALLY DOES NOT** result in a loss of information
  - However, conversion from a `float` to an `int` **USUALLY DOES** result in a loss of information due to truncation

Type Casting (Explicit Coercion)

- **Type casting** explicitly converts one DataType to another

- Example of a C++ *cast operation* (using functional notation):
  
given: `float real; int num;` then
  
  `num = int (3.14159*2.0*real);` // converts float expression to int

- Type casting provides clarity to the program - **shows the intent of the programmer**

- Another way for type casting - put the DataType in parentheses before the number or expression (prefix notation)
  
  `(float)(num + num - num*2);` // takes an extra pair of parentheses
  
  `real*(float)num/2.0;` // converts num to a float first
  
  `real*float(num)/2.0;` // Provides same result as the previous line
Mixed Data Type Expressions

- It is possible to mix DataTypes within an expression - operands are of a different DataType
- Whenever an integer value and floating-point value are joined by an operator, *implicit type coercion occurs as follows:*
  1) The integer is temporarily coerced to a floating point value
  2) The operation is performed
  3) The result of the expression is a floating-point value

See Program DataTypes2_03.cpp for some examples

Formatting the Output

- *Manipulators* are used to control the horizontal spacing of the output.
- Manipulators of interest are defined in the following header files:
  - `#include <iostream>`
  - `#include <iomanip>`
  - `endl` sets the end of a line
  - `setw` sets the width of the output field
  - `fixed` sets fixed point notation
  - `setprecision` sets the precision of floating-point output
  - `showpoint` shows or hides the decimal point
  - `left` left justify the output – Output *starts* on the left in the field specified by `setw`
  - `right` right justify the output – Output *ends* at the right in the field specified by `setw`

- **SYNTAX TEMPLATE:** `OutputStatement`
  - `cout << ExpressionOrManipulator << ExpressionOrManipulator;`
setw

- **setw** (set width) controls how many character positions the next data item should occupy when it is output.
- The argument to setw is an integer expression called the *field width specification*.
  - The data item to be printed is right justified within the field width.
  - If the number specified is too small, the field width is expanded to the exact number of positions necessary.

```
cout << setw(4) << "hello" << setw(12) << "My name is";
Output:   hello$$My name is   ( where $ indicates a space)
cout << setw(2*4) << "hello" << setw(12) << "My name is";
Output:  $$$hello$$My name is
```

```
cout << setw(8) << left << "Hello" << setw(12) << "My name is" << "End";
Output:  Hello$$$My name is$$End
```

**Floating-Point Numbers**

- `setw(#)` also works with floats. The decimal point requires a character position in the output, so 4.85 is 4 characters not 3. *Also, setw(#) applies to the next number output only*.

See Program DataTypes2_04.cpp for examples
Fixed and Showpoint

- To prevent numbers from being output in scientific notation, use the `fixed` manipulator. **Stays true for all future numbers**
  ```cpp
cout << fixed << 4784839.98;  // output is 4784839.98
  ```

- Can toggle back to scientific notation using the output manipulator `scientific`:
  ```cpp
cout << scientific;
  ```

- To have floating point whole numbers printed out with a .0, use `showpoint` - **holds for all subsequent outputs**. To remove this functionality, use the manipulator `noshowpoint`
  ```cpp
cout << showpoint << x << noshowpoint;
  ```

  **See Program DataTypes2_04.cpp for examples**

setprecision

- `setprecision(#)` is used to control the number of decimal places. **Setprecison stays in effect for all future outputs**
  - If in fixed mode, then setprecision determines the number of decimal places
  - If in scientific mode, setprecision determines the number of digits

- `setw, setprecision, fixed and showpoint` can all be used together.
  ```cpp
  float x = 4.353;
  cout << fixed << setprecision(2) << setw(4) << x; // output is: 4.35
  cout << setw(6) << x; // output is: $4.35
  ```

- **Read matters of style on pages 117-118**
  **See Program DataTypes2_04.cpp for examples**
left and right

- *left and right justification*: these manipulators specify the justification to use within the field width specified by `setw`.
  - Stay in effect until changed from one to the other

```cpp
cout << left << setw(8) << "Hello" << right << setw(8) << "Hello";
```

Output:
```
12345678901234567890
Hello      Hello
```

See Program DataTypes2_04.cpp for examples

Additional *string* Operations

- *length*, *size*, *find* and *substr* - all of these functions operate on strings

  - The *length* and *size* function return an unsigned integer value that equals the number of characters currently in the string.
    ```cpp
    string name;
    unsigned int len;
    len = name.length();
    ```

  - To keep from guessing if the compiler requires unsigned int or unsigned long for the return value of length or size, the string class defines a DataType *size_type*
    ```cpp
    string::size_type len; // length is of DataType string::size_type
    len = name.size();
    ```
The **find** Function

- Positions in a string are numbered starting at 0 - the first character in a string has a position value of 0

- The **find** function searches a string to find the **first occurrence** of a particular substring and returns an unsigned integer.
  - **If found**, it returns the position where the substring begins. This value is returned as Data Type `string::size_type`
  - **If not found**, it returns a special number given by `string::npos`
    - `string::npos` is the maximum possible value of `string::size_type`
    - `string` operations do not allow strings to reach this size

The **find** Function

- The substring passed as an argument to the **find** function can be either a literal string or a string expression. The following are all valid **find** function calls:
  ```cpp
  string str1,str2 = "test";
  string::size_type position;
  position = str1.find("when");
  position = str1.find(str2);
  position = str1.find(str2+"when");
  ```
More find Function

\[ \text{str1} = \text{“This class is CPE112”; str2 = “cla”;} \]

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1.find(“is”)</td>
<td>2 (starting counts at 0)</td>
</tr>
<tr>
<td>str1.find(“ss”)</td>
<td>8</td>
</tr>
<tr>
<td>str1.find(‘ ‘)</td>
<td>4</td>
</tr>
<tr>
<td>str1.find(“cpe”)</td>
<td>string::npos</td>
</tr>
<tr>
<td>str1.find(str2+”ss”)</td>
<td>5</td>
</tr>
</tbody>
</table>

The `find(string_to_find)` function matches identical substrings only and determines the first occurrence of the substring.

See Program DataTypes2_05.cpp for examples

The `substr` Function

- The `substr` function returns a particular substring of a string, and the arguments to it are of type `string::size_type`.

  ```
  string str1, str2; str2 = str1.substr(4,10); 
  assigns position 4 through 13 of str1 to str2.
  The first argument in the function call is the starting position.
  The second argument specifies the length of the desired substring.
  ```

- A length of 0 produces the null string as the result.
- **An error occurs** if the starting position is **beyond** the length of the string.
- If the second argument specifies more characters than are present from the starting position, `substr` returns the characters from the starting point to the end of the string.
- **REMEMBER:** counting starts at 0 for the position of a character in a string.
The `substr` Function

- Another form for calling the `substr` function uses a single argument of type `string::size_type`.
  
  ```
  string str1 = “This is the string that is processed”;
  string str2;
  str2 = str1.substr(15);
  str2 contains the phrase “ing that is processed”
  ```

In this case, the number of characters to pull for the substring is not specified, so the `substr` function returns all remaining characters in the string.

**substr Examples**

```
string str1,str2,str3;
string::size_type start = 5;
str1 = “This class is CPE112”;

**Substring function call** | **String returned**
--- | ---
str2 = str1.substr(start,4); | “clas”
str2 = str1.substr(10,10); | “ is CPE112”
str2 = str1.substr(40,2); | Error
str2 = str1.substr(5,0); | “” // null string
str1.length() returns | 20
```

See Program DataTypes2_06.cpp for examples
Function Calls

- **Value-returning functions return a value to the main program upon completion**
- The statement `cout << "square of 9 is: " << Square(9) << endl;` contains the *function call* (or function invocation) `Square(9)`
- When a function is called, the main function is temporarily put on hold, and the called function is started.
- When the called function finishes doing its work, the computer goes back to main and continues where it left off.

- Above, the number 9 in `Square(9)` is known as the *argument* to the function `Square`. Arguments make it possible for functions to work on many different values
- **Syntax Template:** `FunctionCall FunctionName( ArgumentList )`

Function Calls (continued)

- The argument list is a way for functions to communicate with each other, and there can be zero to many arguments in the list
- **Value returning functions** are used in expressions like variables and constants are. *The value returned by the function takes the place of the function call in the expression*
  - `area = square(9.0)*4.0; // all floating point numbers`
  - For value returning functions, *function calls are used within an expression; they do not appear as a separate statement*
  - The function computes a value (result) that is available for use in the expression
  - The function returns *Exactly One function value* (result)
  - The argument to a value-returning function can be any expression of the appropriate type
  - Compiler applies implicit type coercion for mismatched types
Function Calls(continued)

- Arguments in a function can be expressions that evaluate to the appropriate DataType
- Expressions in a function's argument list can even include calls to functions. Note, comments here are used to explain what each line is not what the line does

    // sample program showing function calls as arguments to other functions
    #include <iostream>
    using namespace std;
    float Square(float); // function declaration/prototype
    float Volume(float,float); // function declaration/prototype

Example Continued

int main()
{
    float cylinder_vol, height, radius; // variable declaration
    height = 3.0; radius = 3.0
    cylinder_vol = Volume(height, 3.14159*radius*radius);
    cylinder_vol = Volume(height, 3.14159*Square(radius));
    return 0;
}
float Square (float radius) // function heading
{
    return radius*radius; // value returned by the function
}
float Volume(float cyl_height, float base_area) // function heading
{  
    return cyl_height * base_area;
}
Void Functions

• **Void functions** do not return a function value. They perform some action and then quit (return control to the caller)

• Unlike value returning functions, which are used in expressions, a void function call is a separate, stand-alone statement

• **Function Call** example:
  
  CalcPay(payRate, hours, wages);

• **Void function declaration (Prototype):**
  
  void CalcPay(float, float, float&);  // function prototype

Library Functions

• Every C++ system includes a standard library - a large collection of prewritten functions, DataTypes and other items that any C++ programmer may use

• To use these functions, a #include preprocessor directive specifying the appropriate header file is required near the top of the program

  ```
  #include <iostream>
  #include <cmath>  // Some compilers also require the -lm option when using the math functions CC -lm -o ...
  #include <iomanip>
  ```

• **Appendix A:** lists the identifiers which are reserved words

• **Appendix B:** Shows the operator precedence - several yet to talk about