CSCI 123 Introduction to Programming Concepts in C++

Brad Rippe

Functions
Top Down Design
Applications
Top-Down Design

Browser
Top Down Design

Browser

- UI tabs
- UI L&F
- UI Menu
- Thread Pool Manager
- Threads
- Cache Manager
Top Down Design

• Stepwise Refinement
• Broad description of the product or application to a more detailed description of the individual parts needed to implement the application
• As the application is broken down into more parts the programmers learn more about the problem they are trying to solve
• Break the problem up into subtasks
• Divide and Conquer
Benefits of Top Down Design

• Subtasks, or functions in C++, make programs
  – Easier to understand
  – Easier to change
  – Easier to write
  – Easier to test
  – Easier to debug
  – Easier for teams to develop
C++ Functions

• Callable blocks of code
• May do work without a return value or do work and return a value
• Ability to pass in variables
• One of the building blocks for your applications

Function Call

reverse(input.begin(), input.end());
C++ Built-in Libraries

- `iostream` – provides the functionality to input from the keyboard and output to the screen
  ```cpp
  #include <iostream>
  ```
- `string` – provides string manipulation functions and a string class
  ```cpp
  #include <string>
  ```
C Built-in Libraries

• The elements of the C language library are also included as a subset of the C++ Standard library. These cover many aspects, from general utility functions and macros to input/output functions and dynamic memory management functions.

• They are divided in several files.
<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cassert</code></td>
<td>C Diagnostics Library</td>
</tr>
<tr>
<td><code>cctype</code></td>
<td>Character handling functions</td>
</tr>
<tr>
<td><code>cerrno</code></td>
<td>C Errors</td>
</tr>
<tr>
<td><code>cfloat</code></td>
<td>Characteristics of floating-point types</td>
</tr>
<tr>
<td><code>climits</code></td>
<td>Sizes of integral types</td>
</tr>
<tr>
<td><code>clocale</code></td>
<td>C localization library</td>
</tr>
<tr>
<td><code>cmath</code></td>
<td>C numerics library</td>
</tr>
<tr>
<td><code>csetjmp</code></td>
<td>Non local jumps</td>
</tr>
<tr>
<td><code>csignal</code></td>
<td>C library to handle signals</td>
</tr>
<tr>
<td><code>cstdarg</code></td>
<td>Variable arguments handling</td>
</tr>
<tr>
<td><code>cstddef</code></td>
<td>C Standard definitions</td>
</tr>
<tr>
<td><code>cstdio</code></td>
<td>C library to perform Input/Output operations</td>
</tr>
<tr>
<td><code>cstdlib</code></td>
<td>C Standard General Utilities Library</td>
</tr>
<tr>
<td><code>cstring</code></td>
<td>C Strings</td>
</tr>
<tr>
<td><code>ctime</code></td>
<td>C Time Library</td>
</tr>
</tbody>
</table>
Converting characters

- char upperChar = theChar - 32;

- With the C++ libraries we can now use from the cctype library:

```c
toupper(theChar);
```
<table>
<thead>
<tr>
<th>ASCII character set</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image12x3_to_702x545" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The digits at the left of the table are the left digits of the decimal equivalent (0127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for "F" is 70, and the character code for "a" is 38.
Testing CharacterCodes

• Testing the character for validity, test the boundaries (upper, lower, inside and outside):
• ` (96) outside the lower bound
• { (123) outside the upper bound
• a (97) inside the boundary
• z (122) inside the boundary
• A character in the range ‘i’ (108)
• and one out ‘+’ which is (43)
Revise characterCodes.cpp

• We must use the `#include <cctype>` directive

• The `cctype` refers to a header file. The header file provides basic information about the library.

• Allows the linker to find the object file for creating the executable
Preprocessor

- `#include` directive
- Inserts the contents of the file specified `< >`
- Runs before your code gets to the compiler
- Searches in the directory of the file then the include path
- More after we cover header files
using namespace std;

- Most standard libraries require you to qualify the objects with the std namespace
- You only have to use one using statement for all includes that use the std namespace

#include <iostream>
#include <cmath>
using namespace std;
Namespace

• If you don’t use the *using* statement, you will have to qualify your objects (like `cin` and `cout`) with the std namespace in each statement with the *scope resolution operator*, `::`

```cpp
std::cout << "Hi Class\n";
std::cout << "type a letter\n";
std::cin >> aLetter;
```
Namespace

• Why?
  – Avoids collisions between objects with the same name
  – If you define a type called Car and I define a type called Car, how does the compiler know which Car to use?
    • Namespaces

    mycar::Car
    yourcar::Car

    mycar::Car mycar;
    yourcar::Car yourcar;
    cout << mycar.info << endl;
    cout << yourcar.info << endl;

• More later in the course
Functions

• If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.

• Functions can have more than one argument
Is the following acceptable?

```cpp
string myStr = "Hello";

cout << toupper(myStr) << endl;

• We can’t do this because the function “toupper” is not defined for a string data type

int toupper( int c ) // signature
Is the following acceptable?

double myInt = 7.0;

cout << toupper(myInt) << endl;

• Like arithmetic operations, C++ will convert the double to an int with loss of data. So the above code is acceptable.

int toupper( int c )
Type Conversion

• If the function has integral data types as arguments, C++ will do automatic type conversion for you
• If the function has arguments of type double and the function call passes in a argument of type int, the int argument will be promoted to double

```cpp
double someFunction(double arg)
```

The function call

```cpp
int myInt = 7;
someFunction(myInt);
```
Type Casting

• What happens here?

```java
int operand1 = 18;
int operand2 = 4;
double result;
result = operand1/operand2;
```

• Result is 4
Type Casting

• What happens here?
  
  int operand1 = 18;
  int operand2 = 4;
  double result;
  result = static_cast<double>(operand1)/operand2;

• Result is 4.5
• Converts operand1 to a double
Type Casting

• What happens here?

```cpp
int operand1 = 18;
int operand2 = 4;
double result;
result = static_cast<double>(operand1/operand2);
```

• Result is 4
• Converts the result to a double after the operation
Improve CharacterCodes

• No need for the extra variable

```cpp
cout << "I can capitalize the char '"
    << static_cast<char>(toupper(theChar))
    << endl;
```
static_cast

• Useful when assigning a larger arithmetic type to a smaller type.
• Informs the compiler that you are aware of the loss data
• The compiler will no longer issue a warning message about the loss of data
double result;
int operand1 = 18;
int operand2 = 4;

result = (double)(operand1/operand2);
cout << "(double)(operand1/operand2) " << result;
cout << endl;

result = (double)(operand1)/operand2;
cout << "(double)(operand1)/operand2 " << result;
cout << endl;
return 0;
Programmer Defined Functions

• Requires two parts
  – Function Declaration (or prototype)
    • Provides the interface for the function
      
      ```
      returnType functionName(ParameterList);
      ```
  – Function Definition
    • Provides the instructions for the function
    • These instructions are executed when the function is called
      
      ```
      returnType functionName(ParameterList) {
        cout << “Function Instructions\n”;  
      }
      ```
CharacterCodes with Functions

- The code is more readable
- Easier to maintain
- Easier to manage
- Eliminates redundant code
int maximum(int digit1, int digit2, int digit3); // prototype

int main() {
    cout << "Finding the max from 23, 0, 88\n";
    int max = maximum(23, 0, 88);
    cout << "The max value is " << max << endl;
    return 0;
}

int maximum(int digit1, int digit2, int digit3) { // definition
    int max = 0;
    if(digit1 < digit2)
        max = digit2;
    else
        max = digit1;
    if(max < digit3)
        max = digit3;
    return max;
}
Last Time

• Top Down Design
  – (what is it?) (why does it help?)

• Predefined Libraries
  – <cctype>
  – islower()
  – toupper()
  – static_cast<dataType> ( )
  – Type Conversion Promotion of variables that are of smaller data storage to variables of a higher data storage

• User Defined Functions
  – maximum()
Programmer Defined Functions

• Requires two parts
  – Function Declaration *(or prototype)*
    • Provides the interface for the function
      
      ```
      returnType functionName(ParameterList);
      ```
  – Function Definition
    • Provides the instructions for the function
    • These instructions are executed when the function is called
      
      ```
      returnType functionName(ParameterList) {
        cout << "Function Instructions\n";
      }
      ```
The Return Statement

• Ends the function call
• Returns the value calculated by the function
• Syntax:
  
  \[
  \text{return expression;}
  \]
  
  – expression performs the calculation
  or
  – expression is a variable containing the calculated value
• Example:
  
  \[
  \text{return salary+ salary * PERCENT_INCREASE;}
  \]
The Return Statement

• The main function is allows to return without a return statement.
• If the return statement is left out, the compiler will insert a return 0
• The return in the main is treated as a status indicator
• Zero indicates success
• Nonzero has a machine dependent meaning
• More on this later
Global Functions

• Functions that don’t belong to a class are known as Global Functions

• int maximum(int digit1, int digit2, int digit3);

• From the `cmath` library:
  – `pow(int x)` – powers function
  – `sqrt(int x)` – square root function
  – `ceil(int x)` – rounds $x$ to the largest integer not greater than $x$
  – `floor(int x)` – rounds $x$ to the largest integer not greater than $x`
CharacterCodes Version 3

• Start with version 2 (improvedCharacterCodes)
• Ask what subtasks we might be able to divide the program into:
  - promptForLowerChar()
  - invalidCharMsg()
  - outputUpperCase(char charInput)
  - promptToContinue()
Main() CharacterCodes

char theChar = '1';
while (theChar != '0') {
    promptForLowerCase();
    cin >> theChar;

    if (!islower(theChar)) {
        invalidCharMsg();
        continue;
    }
}
outputUpperCase(theChar);
promptToContinue();
 cin >> theChar;
Stack – Data Structure

- **LIFO** (Last In First Out) Data Structure
- Think of a stack of dishes
- Put dishes onto the stack we use the term **push** dishes onto the stack
- Take a dish off of the stack, we use the term **pop** off the stack
Step 1: Operating system invokes `main` to execute application.

```
int main()
{
    int a = 10;
    cout << a << " squared: "
        << square(a) << endl;
    return 0;
}
```
Step 2: `main` invokes function `square` to perform calculation.

```cpp
int main()
{
    int a = 10;
    cout << a << " squared: "
         << square( a ) << endl;
    return 0;
}
```

Function call stack after Step 2:

1. Activation record for function `main`
   - Return location: R2
   - Automatic variables:
     - a 10

2. Activation record for function `square`
   - Return location: R1
   - Automatic variables:
     - x 10

Top of stack.
Step 3: square returns its result to main.

```cpp
int main()
{
    int a = 10;
    cout << a << " squared: " << square(a) << endl;
    return 0;
}
```

Return location **R2**

Function call stack after Step 3

Top of stack

Activation record for function `main`

Return location: **R1**

Automatic variables:

- `a` : 10
Sample 8 Maximum
Function Call Details

• The values of the arguments are plugged into the formal parameters (Call-by-value mechanism with call-by-value parameters)
  – The first argument is used for the first formal parameter, the second argument for the second formal parameter, and so forth.
  – The value plugged into the formal parameter is used in all instances of the formal parameter in the function body
Alternate Declarations

• Two forms for function declarations
  – List formal parameter names
  – List types of formal parameters, but not names
  – First aids description of the function in comments

• Examples:
  int maximum(int digit1, int digit2, int digit3);

  int maximum(int, int, int, int);

• Function headers must always list formal parameter names!
Function Definition Syntax

• Within a function definition
  – Variables **must** be declared before they are used
  – Variables are typically declared before the executable statements begin
  – At least one return statement must end the function
    • Each branch of an if-else statement might have its own return statement
Defining Functions

- C++ programs do not compile unless function prototypes are provided for every function or each function is defined before it is called.
Order of Arguments

• Compiler checks that the types of the arguments are correct and in the correct sequence.
• Compiler cannot check that arguments are in the correct logical order
• Example: Given the function declaration:
  `char grade(int received, int minScore);`

  ```
  int received = 95, minScore = 60;
  cout << grade( minScore, received);
  ```

  – Produces a faulty result because the arguments are not in the correct logical order. The compiler will not catch this!
Procedural Abstraction

• The Black Box Analogy
  – A black box refers to something that we know how to use, but the method of operation is unknown
  – A person using a program does not need to know how it is coded
  – A person using a program needs to know what the program does, not how it does it

• Functions and the Black Box Analogy
  – A programmer who uses a function needs to know what the function does, not how it does it
  – A programmer needs to know what will be produced if the proper arguments are put into the box

Use of islower, toupper()
Functions in the cctype library

• Function Signature or Interface (how we use these functions)

• int toupper(int ch);
• int tolower(int ch);
• int islower(int ch);
• etc

• Hiding the implementation is known as information hiding
  – One of the key concepts in OOP
Function Implementations and The Black Box

• Designing with the black box in mind allows us
  – To change or improve a function definition without forcing programmers using the function to change what they have done
  – To know how to use a function simply by reading the function declaration and its comment
Procedural Abstraction and C++

- Procedural Abstraction is writing and using functions as if they were black boxes
  - Procedure is a general term meaning a “function like” set of instructions
  - Abstraction implies that when you use a function as a black box, you abstract away the details of the code in the function body
Example of abstraction

// Example 1
double calculateRetroPay(double salary) {
    return salary*(1+PERCENT_INCREASE);
}

// Example 2
double calculateRetroPay(double salary) {
    double retroPay;
    retroPay = salary*PERCENT_INCREASE;
    retroPay += salary;
    return retroPay;
}
Defining Functions

• The Function declaration and the comments will aid the programmer in using the function

• Doxygen example

/**
* Function Description
* @param aNumberOfPeas contains the number of peas
* @param aNumberOfPods contains the number of pods
* @pre aNumberOfPeas and aNumberOfPods are initialized by the calling program.
* @post aNumberOfPeas and aNumberOfPods are multiplied to get the number of peas in a pod.
* @return the product of aNumberOfPeas and aNumberOfPods
*/
Formal Parameter Names

• Functions are designed as self-contained modules
• Different programmers may write each function
• Programmers choose meaningful names for formal parameters
  – Formal parameter names may or may not match variable names used in the main part of the program
  – It does not matter if formal parameter names match other variable names in the program
  – Remember that only the value of the argument is plugged into the formal parameter
Program Testing

• Programs that compile and run can still produce errors
• Testing increases confidence that the program works correctly
  – Run the program with data that has known output
    • You may have determined this output with pencil and paper or a calculator
  – Run the program on several different sets of data
    • Your first set of data may produce correct results in spite of a logical error in the code
      – Remember the integer division problem? If there is no fractional remainder, integer division will give apparently correct results
Use Pseudocode

• Pseudocode is a mixture of English and the programming language in use
• Pseudocode simplifies algorithm design by allowing you to ignore the specific syntax of the programming language as you work out the details of the algorithm
  – If the step is obvious, use C++
  – If the step is difficult to express in C++, use English
Local Variables

• Variables declared in a function:
  – Are local to that function, they cannot be used from outside the function
  – Have the function as their scope

• Variables declared in the main part of a program:
  – Are local to the main part of the program, they cannot be used from outside the main part
  – Have the main part as their scope
Global Constants

- Global Named Constant
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body
  - Declared outside the main function body
  - Declared before any function that uses it

- Example:
  ```
  const double PI = 3.14159;
  double volume(double);
  int main()
  {
  ...
  }
  ```
  - PI is available to the main function and to function volume
Global Variables

- Global Variable -- rarely used when more than one function must use a common variable
  - Declared just like a global constant except const is not used
  - Generally make programs more difficult to understand and maintain
Formal Parameters are Local Variables

• Formal Parameters are actually variables that are local to the function definition
  – They are used just as if they were declared in the function body
  – Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration

• The call-by-value mechanism
  – When a function is called the formal parameters are initialized to the values of the arguments in the function call
Namespaces Revisited

• The start of a file is not always the best place for
  
  using namespace std;

• Different functions may use different namespaces
  
  – Placing `using namespace std;` inside the starting brace of a function
    
    • Allows the use of different namespaces in different functions
    
    • Makes the “using” directive local to the function
Overloading Function Names

• C++ allows more than one definition for the same function name
  – Very convenient for situations in which the “same” function is needed for different numbers or types of arguments
• Overloading a function name means providing more than one declaration and definition using the same function name
Overloading Examples

- double ave(double n1, double n2)
  {
      return ((n1 + n2) / 2);
  }
- double ave(double n1, double n2, double n3)
  {
      return ((n1 + n2 + n3) / 3);
  }
  - Compiler checks the number and types of arguments in the function call to decide which function to use

    cout << ave(10, 20, 30);

    uses the second definition
Overloading Details

• Overloaded functions
  – Must have different numbers of formal parameters
    AND / OR
  – Must have at least one different type of parameter

– Must return a value of the same type
Overloading Example
int maximum(int digit1, int digit2); // prototype
int maximum(int digit1, int digit2, int digit3); // prototype
int main() {
    cout << "Finding the max from 23, 0, 88\n";
    int max = maximum(23, 0, 88);
    cout << "The max value is " << max << endl;
    return 0;
}

int maximum(int digit1, int digit2) {
    double max = 0;
    if(digit1 < digit2)
        return digit2;
    else
        return digit1;
}
int maximum(int digit1, int digit2, int digit3) {
    int max = 0;
    if(digit1 < digit2)
        max = digit2;
    else
        max = digit1;
    if(max < digit3)
        max = digit3;
    return max;
}
Which function gets called?
Automatic Type Conversion

- Given the definition
  ```
  double mpg(double miles, double gallons) {
    return (miles / gallons);
  }
  ```
  what will happen if mpg is called in this way?
  ```
  cout << mpg(45, 2) << “ miles per gallon”;
  ```
- The values of the arguments will automatically be converted to type double (45.0 and 2.0)
int maximum(int digit1, int digit2, int digit3) {
    int max = 0;
    if(digit1 < digit2)
        max = digit2;
    else
        max = digit1;
    if(max < digit3)
        max = digit3;
    return max;
}

double maximum(int digit1, int digit2, int digit3) {
    double max = 0.0;
    if(digit1 < digit2)
        max = static_cast<double>(digit2);
    else
        max = static_cast<double>(digit1);
    if(max < digit3)
        max = static_cast<double>(digit3);
    return max;
}
Type Conversion Problem

• Given the 1st mpg definition and the second definition in the same program
  
  ```cpp
  int mpg(double miles, double gallons)
  // returns the Miles per gallon
  {
    return (miles/gallons);
  }
  int mpg(int goals, int misses)
  // returns the Measure of Perfect Goals
  {
    return (goals – misses);
  }
  ```

what happens if mpg is called this way now?
  
  ```cpp
  cout << mpg(45, 2) << “ miles per gallon”;
  ```

– The compiler chooses the function that matches parameter types so the Measure of Perfect Goals will be calculated

**Do not use the same function name for unrelated functions**