Functions

CMSC 202

Topics Covered

- Function review
- More on variable scope
- Call-by-reference parameters
- Call-by-value parameters
- Function overloading
- Default parameters

Function Review

We've already introduced:

- Using predefined functions, including describing some popular library functions
- The basic parts of defining a new function:
  - Function declaration
  - Function definition
  - Function call

Function Review

We also covered:

- Declaring the return type of a function, including using “void”
- Declaring the parameters of a function, including, again, using “void”

Function Review

Lastly, we introduced:

- Variable scope
- Local vs. global variables

Functions

- New stuff
  - More on pre-/post-conditions
  - Parameters
    - Call by value
    - Call by reference
    - Constant
    - Default
  - Function return types
  - Function overloading

Functions – Pre-/Post-Conditions

- Functional abstraction
  - Hiding the details…
  - Do not need to know how a function works to use it
- Pre-conditions
  - Comment describing constraints on parameter values to ensure proper function execution
- Post-conditions
  - Comment describing state of the program after function execution
- VERY IMPORTANT!!!

Pre/Post-condition Example

```
//-------------------------------------------------------
// Function: ShowInterest
// PreCondition:
//      balance is a nonnegative savings account balance
//      rate is the interest rate expressed as a percent
//              such as 5 for 5%
// PostCondition:
//      the amount of interest for the given balance at the
//              given rate is displayed to cout.
//      if the parameters are invalid, "No Interest" is
   displayed
//-------------------------------------------------------
void ShowInterest( double balance, double rate )
{
    if (balance >= 0 && rate >=0)
    {
      // code to calculate and display interest }
    else
    {
       cout << "No Interest\n";
    }
}
```

Preconditions

- How to write a good precondition?
  - Describe assumptions/limitations of each parameter
    - Ex: denominator cannot be equal to zero
  - Describe assumptions/limitations of the program state
    - Ex: global array "students" must have at least one student in it
- What must the function do?
  - Test every precondition!!!
    - Ex:

```
if (denominator == 0)
      cerr << "Error: Denom == 0" << endl;
```

    - Ex:

```
if (NbrOfStudents < 1)
      cerr << "Error: NbrOfStud < 1" << endl;
```

Preconditions

- How to deal with unmet preconditions?
  - Handle the error by returning a "safe" value or printing an error
    - Prefer NOT to print errors from functions!
  - Return a status value
  - Throw an exception (later…)
  - Last resort: Abort the program (exit or assert)

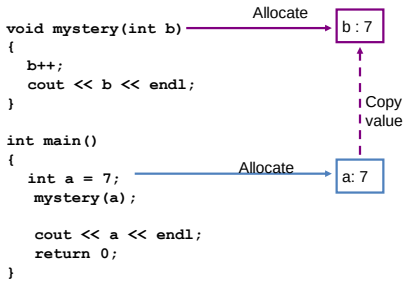
Postconditions

- How to write a good postcondition?
  - Describe all possible message from the function
    - Ex: Error message is printed if preconditions are violated
  - Describe all possible return values
    - Ex: Return value is 0 if an error is encountered, otherwise, a positive value representing the current rate calculated is returned
- What must the function do?
  - Functionality must match postcondition claims!

Function Parameters

- Argument
  - Value/variable passed IN to a function
- Parameter (or Formal Parameter)
  - Variable name INSIDE the function
- Call-by-Value
  - We've already seen this
  - The parameter is a local variable—contains a *copy* of the argument passed in by caller
  - Changes to the parameter do not affect the argument

Call by Value Example




```
void mystery(int b)
{
  b++;
  cout << b << endl;
}

int main()
{
  int a = 7;
   mystery(a);

   cout << a << endl;
   return 0;
}
```

Old C-style "Call by Reference"

- "Call by reference" is a parameter-passing scheme where a reference to the original caller's argument is passed in to a function
  - This allowed caller's variable to be modified by called function
- Originally, C (and earliest versions of C++) implemented this with pointers
  - So we pass in the address of, i.e., a usable reference to, the caller's variable

Call by Reference

- C++ has true *call by reference*
  - Changes to the parameter change the argument
    - Function declares that it will change argument
  - Share memory
    - Essentially a pointer
  - Syntax:

```
retType funcName( type &varName, … ){ … }
```

Look familiar?
Works "backwards"

Call by Reference Example

```
void mystery(int &b)
{
   b++;
   cout << b << endl;
}

int main()
{
   int a = 7;
    mystery(a);

    cout << a << endl;
    return 0;
}
```

Value versus Reference?

- Why choose value or reference?
  - Value
    - Data going in, nothing coming out
    - Only one piece of data coming out (return it!)
  - Reference
    - Need to modify a value
    - Need to return more than one piece of data
    - Passed an array (by default are by reference, no '&' needed)

Call-by-Reference – Issue!

- What happens in the following?

```
void mystery(int &b)
{
   b++;
   cout << b << endl;
}

int main()
{
    mystery(6);

    return 0;
}
```

Practice 1

- What is printed in the following code block?

```
void mystery(int a, int &b) {
   a++;
   b++;
   cout << a << " " << b << endl;
}

int main() {
   int a = 1;
   int b = 1;

    mystery(a, b);
    mystery(b, a);
    mystery(a, a);

   cout << a << " " << b << endl;
    return 0;
}
```

Practice 2

- Correctly implement a swap function such that the following code will work:

```
int a = 7;
int b = 8;
Swap(a, b);
cout << a << " " << b << endl;
// We want the above to print out "8 7"
```

Recap

- Pass by value:
  - Changes to Parameter do NOT affect Argument
  - Syntax:

```
retType funcName( type variable, …) { }
```

- Pass by reference:
  - Changes to Parameter DO affect Argument
  - Syntax:

```
retType funcName( type &variable, …) {
  }
```

Parameter Passing Guidelines

- Pass class-type objects by reference
  - string, vector, Car, Customer, etc.
    (more on this later…)
- Pass primitive objects by value
  - int, double, float, bool, etc.
    (some use for returning "extra values", but…)
- & can be any where between type and reference, personal choice
  - type& variable
  - type & variable
  - type &variable

Constant Parameters

- Don't want a function to change class-type objects?
  - Use 'const' instead of pass-by-value
    - Declares the parameter as constant
    - No changes are allowed to the parameter
    - Prevents copy of entire object
- Syntax:
  - retType funcName(const type &variable, …) { }
- Example:
  - int findItem(const vector<int> &myVec, int key) { }

Const Parameters

- Example

```
void AddOne (const int& n) {
  n++;                    // compiler error
}

int main ( ) {
  int x = 42;
  AddOne ( x )
}
```

Const Parameter Rules

- Bottom Line
  - Primitive/Built-in types (int, double, float, …)
    - Pass by value
      - Function is NOT changing argument
    - Pass by reference
      - Function IS changing argument
  - Class/User-defined types (string, vector, Car, …)
    - Pass by const reference
      - Function is NOT changing argument
    - Pass by reference
      - Function IS changing argument

Function Return Types

- Return by value?
  - Yes, you usually do this – makes a copy of the value
- Return by reference?
  - Yes, does not make a copy of the value
  - DANGER – the value/memory MUST be dynamically allocated (and not go out-of-scope when function ends)
- Return by const value?
  - No, almost never use this
- Return by const reference?
  - Yes, return class-type objects this way to prevent copy
  - A reference to the original unchangeable object is returned

Default Parameters

- Allow us to define functions with optional parameters
  - Optional parameter gets a default value
  - Must be right-most parameters, why?
- Syntax:

```
retType funcName( type variable = defValue )
```

- Example
  - Function that adds between 2 and 5 integers

```
int Add( int a, int b, int c = 0,
         int d = 0, int e = 0)
{
  return a + b + c + d + e;
}
```

Default Parameters Example

```
int Add(int a, int b, int c = 0,
        int d = 0, int e = 0)
{
  return a + b + c + d + e;
}

int main()
{
  cout << Add (1, 2) << endl;
  cout << Add (1, 2, 3) << endl;
  cout << Add (1, 2, 3, 4) << endl;
  cout << Add (1, 2, 3, 4, 5) << endl;
}
```

Overloading Functions

- C limitation
  - Functions are unique based on name
- C++ extention
  - Functions are unique based on name AND parameter list (type and number)
- Overloading
  - Declaring two or more functions with same name
  - Must have different parameter lists
    - Return types are NOT used to differentiate functions

Overloading Example

```
int AddTwo( int a, int b)
{
   return a + b;
}

double AddTwo( double a, double b)
{
   return a + b;
}

int main()
{
   cout << AddTwo(3, 4) << endl;
   cout << AddTwo(3.0, 4.0) << endl;
   cout << AddTwo(3, 4.0) << endl;
   cout << AddTwo(3.0, 4) << endl;
   return 0;
}
```

Interesting…

- What happens with this?

```
float AddTwo( float a, float b)
{
  return a + b;
}

int main()
{
  cout << AddTwo(3.0, 4.0) << endl;

  return 0;
}
```

Function Prototypes

- C++ allows us to define a function above main OR below it…
  - If function is defined below, we must *prototype* it
    - Declare the function above main, define below
    - Prototype must match function EXACTLY
- Syntax:

```
retType funcName( parameter_list );
```

Semi-colon!!!

- Why?
  - Easier to find main
  - Easier to read

Prototype Example

```
int AddTwo( int a, int b);
double AddTwo( double a, double b);

int main()
{
   cout << AddTwo(3, 4) << endl;
   cout << AddTwo(3.0, 4.0) << endl;
   return 0;
}

int AddTwo( int a, int b)
{
   return a + b;
}

double AddTwo( double a, double b)
{
   return a + b;
}
```

Common Errors…

- What's wrong with the following?

```
void swap (int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

Common Errors…

```
bool IsOld (int age, int oldAge)
{
  if (age > oldAge)
     return true;
}
```

```
Main.cpp: In function `bool IsOld(int,
  int)':
Main.cpp:42: warning: control reaches
  end of non-void function
```

Common Errors…

```
int AddOne( int n );

int main ( )
{
   int x = 42;
   cout << Addone ( x ) << endl;
   return 0;
}

int AddOne (int n)
{
   return n + 1;
}
```

```
Main.cpp: In function `int main()':
Main.cpp:6 : `Addone' undeclared (first use this function)
Main.cpp:6: (Each undeclared identifier is reported only once
   for each function it appears in.)
```

Common Errors…

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

int main ( )
{
    int x = 42;
    int y = Add ( 66, x);
    cout << y << endl;
    return 0;
}

int Add (int n, int m)
{
    return n + m;
}
```

```
Main.cpp: In function `int main()':
Main.cpp :8: `Add' undeclared (first use this function)
Main.cpp :8: (Each undeclared identifier is reported only once for each function
    it appears in.)
Main.cpp : In function `int Add(int, int)':
Main.cpp :16: `int Add(int, int)' used prior to declaration
```

Parameter Selection…