More on Functions

Lecture 7
COP 3014 Fall 2021

October 6, 2021

Function Overloading

The term **function overloading** refers to the way C++ allows more than one function in the same scope to share the same name – as long as they have different parameter lists

- The rationale is that the compiler must be able to look at any function call and decide exactly which function is being invoked
- Overloading allows intuitive function names to be used in multiple contexts
- The parameter list can differ in number of parameters, or types of parameters, or both
- Example:

```
int Process(double num);          // function 1
int Process(char letter);        // function 2
int Process(double num, int position);
                                  // function 3
```

Function Overloading

Sample calls, based on the above declarations

```
int x;
float y = 12.34;

x = Process(3.45, 12);          // invokes function 3
x = Process('f');              // invokes function 2
x = Process(y);               // invokes function 1
              //(automatic type conversion applies)
```

Avoiding Ambiguity

- Even with legally overloaded functions, it's possible to make ambiguous function calls, largely due to automatic type conversions.
- Consider these functions

```
int Compute(int x, int y, int z = 5);
            // z has a default value
void RunAround(char x, int r=7, double f=0.5);
            // r and f have default values
```

- **Important Rule:** Since the compiler processes a function call by filling arguments into the parameter list left to right, any default parameters **MUST** be at the end of the list

```
void Jump(int a, int b = 2, int c);
            // This is illegal
```

Avoiding Ambiguity

Legal Calls

```
int a = 2, b = 4, c = 10, r;

cout << Compute(a, b, c); // all 3 parameters used

r = Compute(b, 3); // z takes its default value of 5
          // (only 2 arguments passed in)

RunAround('a', 4, 6.5);      // all 3 arguments sent
RunAround('a', 4);           // 2 arguments sent
                     // f takes default value
RunAround('a');              // 1 argument sent
                     // r and f take defaults
```

Default parameters and overloading

- A function that uses default parameters can count as a function with different numbers of parameters. Recall the three functions in the overloading example:

```
int Process(double num);                    // function 1
int Process(char letter);                   // function 2
int Process(double num, int position); //
function 3
```

- Now suppose we declare the following function:

```
int Process(double x, int y = 5); // function 4
```

- This function conflicts with function 3, obviously. It ALSO conflicts with function 1. Consider these calls:

```
cout<<Process(1.3,10); //matches functions 3 & 4
cout << Process(13.5); // matches functions 1 & 4
```

- So, function 4 cannot exist along with function 1 or function 3
- BE CAREFUL to take default parameters into account when using function overloading!

Reference Variables

- A reference variable is a nickname, or alias, for some other variable
- To delare a reference variable, we use the unary operator &

```
int n = 5; // this declares a variable, n
int & r = n;//this declares r as a reference to n
```

- In this example, r is now a reference to n. (They are both referring to the SAME storage location in memory).
- To declare a reference variable, add the & operator after the type
- Note: The notation can become confusing when different sources place the & differently. The following three declarations are equivalent:

```
int &r = n;
int& r = n;
int & r = n;
```

- The spacing between the "int" and the "r" is irrelevant. All three of these declare r as a reference variable that refers to n.

WHY???!

- While the above code example shows what a reference variable is, you will not likely use it this way!
- In this example, the regular variable and the reference are in the same scope, so it seems silly. ("Why do I need to call it r when I can call it x ?")
- So when are references useful? When the two variables are in different scopes (this means functions!)

Pass By Value

- Recall that the variables in the formal parameter list are always local variables of a function
- This is known as Pass By Value - function parameters receive copies of the data sent in.

```
void Func1(int x, double y)
{
      x=12; // these won't affect the caller
      y=20.5; // they change LOCAL variables x & y
}
```

- In the function above, any int and double r-values may be sent in

```
int num = 5;
double avg = 10.7;
Func1(num, avg);                  // legal
Func1(4, 10.6);                   // legal
Func1(num + 6, avg - 10.6);       // legal
```

Pass By Reference

- Consider the following function

```
void Twice(int& a, int& b)
{
      a *= 2;
      b *= 2;
}
```

- Note that when it is run, the variables passed into Twice from the main() function DO get changed by the function
- The parameters a and b are still local to the function, but they are reference variables (i.e. nicknames to the original variables passed in (x and y))

Pass by Reference

- When reference variables are used as formal parameters, this is known as **Pass By Reference**

```
void Func2(int& x, double& y)
{
      x = 12; // these WILL affect
      y = 20.5; //the original arguments
}
```

- When a function expects strict reference types in the parameter list, an L-value (i.e. a variable, or storage location) must be passed in

```
int num;
double avg;
Func2(num, avg);              // legal
Func2(4, 10.6);               // NOT legal
Func2(num + 6, avg - 10.6);       // NOT legal
```

Pass by Reference

- Note: This also works the same for return types. A return by value means a copy will be made. A reference return type sends back a reference to the original.

```
int Task1(int x, double y);
                    // uses return by value
int& Task2(int x, double y);
                   // uses return by reference
```

- This is a trickier situation than reference parameters (which we will not see in detail right now).

Comparing: Value vs. Reference

- Pass By Value
  - The local parameters are copies of the original arguments passed in
  - Changes made in the function to these variables do not affect originals
- Pass By Reference
  - The local parameters are references to the storage locations of the original arguments passed in.
  - Changes to these variables in the function will affect the originals
  - No copy is made, so overhead of copying (time, storage) is saved

const Reference Parameters

- The keyword const can be used on reference parameters.
  `void Func3(const int& x);`
- This will prevent x from being changed in the function body
- General Format:
  `const typeName & variableName`
- This establishes variableName as a reference to a location that cannot be changed through the use of variableName.
- This would be used to avoid the overhead of making a copy, but still prevent the data from being changed
- Since the compiler will guarantee that the parameter value cannot change, it IS legal to pass in any R-value in this case:

```
int num = 5;
Func3(num); // legal
Func3(10); // legal
Func3(num + 50); // legal
```