Functions: Overloading and Default Parameters

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: The following 3 functions are considered different and distinguishable by the compiler, as they have different parameter lists

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

- 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)
```

- Try this example here

Avoiding Ambiguity

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

```
void DoTask(int x, double y);
void DoTask(double a, int b);
```

- These functions are legally overloaded. The first two calls below are fine. The third one is ambiguous:

```
DoTask(4, 5.9);       // calls function 1
DoTask(10.4, 3);      // calls function 2
DoTask(1, 2);         // ambiguous due to type conversion (int -> double)
```

Default parameters:

In C++, functions can be made more versatile by allowing **default values on parameters**. This allows some parameters to be *optional* for the caller

- To do this, assign the formal parameter a value when the function is first declared
- Such parameters are optional.
  - If the caller *does* use that argument slot, the parameter takes the value passed in by the caller (the normal way functions work)
  - If the caller chooses *not* to fill that argument slot, the parameter takes its default value
- Examples
  Declarations

```
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
```

  Legal Calls

```
int a = 2, b = 4, c = 10, r;
cout << Compute(a, b, c);      // all 3 parameters used (2, 4, 10)
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
```

- **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
```

- defaults.cpp -- Simple example illustrating default parameter value

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(12.3, 10);   // matches functions 3 and 4
cout << Process(13.5);       // matches functions 1 and 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!
