Functions – Part IV

Parameter Passing Methods

- <u>Call-by-value</u> – a value is sent to the function. Changes to the function parameter aren’t seen outside of that module call.

Parameter Passing Methods

- Call-by-reference – a reference to a variable is sent to the function. Changes to the function parameter actually change the variable argument sent in the function call.

Relative Advantages

- Call-by-value – functions are completely independent
- Call-by-value – can send variables or values for the argument
- Call-by-reference – allows a function to update multiple variables
- Call-by-reference – memory

C++ How To

- To create a reference parameter, follow the parameter type with a & (in the prototype and implementation)

```
double ValueToCelcius(double x);
double RefToCelcius(double & x);

int main() {
  double temperature;
  cout << “Enter temperature in degrees Fahrenheit”;
  cin >> temperature;
  cout << ValueToCelcius(temperature) << endl;
  cout << RefToCelcius(temperature) << endl;
  cout << temperature << endl;
  return 0;
}

double ValueToCelcius(double x) {
  x = ( 5 / 9.0 )*( x – 32 ); // changes the copy
  return x;
}


double RefToCelcius(double & x) {
  x = ( 5 / 9.0 )*( x – 32 ); // changes the original
  return x;
}
```

Recursive Functions

- A recursive function is a function that calls itself.
- In order for a recursive function to terminate, it must call “simpler” versions of itself (the inductive step) until it reaches a base case

Recursive Functions

<u>Factorial</u>

- base case:
  - 0! = 1
- inductive step:
  - n! = n * (n-1)!, for n>=1

Recursive Functions

<u>The Fibonacci Sequence</u>

base cases:

$\text{Fib}(0) = 1$

$\text{Fib}(1) = 1$

inductive step

$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

for $n>=2$

Overloading Functions

- We can write multiple functions with the same name as long as their parameter lists are distinct (by type and/or number of parameters).
- This will keep the programmer that is using our functions from having to remember different function names for all of the different parameter lists.

```
int max( int num1, int num2 ) {
  if ( num1 > num2 )
    return num1;
  else
    return num2;
}

int max( int num1, int num2, int num3, int num4 ) {
  int largest = num1;
  if ( num2 > largest )
    largest = num2;
  if ( num3 > largest )
    largest = num3;
  if ( num4 > largest )
    largest = num4;
  return largest;
}
```

Default Arguments

- We can make our functions more versatile by providing default arguments for the parameters at the end of the parameter list.
- If the function is called with fewer than the number of parameters in the list, the default values will be assigned to the missing parameters.

Default Arguments

To provide default arguments for parameters in your function, simply place the assignment operator and default value with the parameter in the <u>function prototype</u> (do NOT place default values in the function implementation).

Default Arguments

Example:

The following function prototype is for a function that will return the maximum of 2, 3, or 4 positive integers.

```
int max(int, int, int = 0, int = 0);
```

Default Arguments

The default values must come from right to left. So, values that may be left out of your function call should come at the end of your parameter list.

For example, the following function prototype is invalid.

```
int max(int, int = 0, int, int = 0);
```

Default Argument

Your default values must not cause ambiguity with overloaded functions.

For example, the prototypes:

```
int max(int, int, int = 0, int = 0);
int max(int, int);
```

will cause confusion when a function call has two integer arguments.