Lecture 11: Functions

PIC 10A

Todd Wittman

Sec 4.1: Functions

- C++ is modular, which means the program can be built out of smaller components. Much like Swedish furniture.
- A function is a set of instructions separate from the main routine that can be called on at any time.
- Generally we pass the function parameters.
- The function often returns a value.
- We've already used several built-in functions

main ( )	Line.move (dx, dy)
sqrt (x)	cwin.coord (x1, y1, x2, y2)
getline(cin,s)	setw (columnWidth)

- Today we're going to learn how to write our own functions.

Sec 4.2: Writing Functions

- When we declare a function, we need to identify the data types of the parameters that are passed to the function and the type of variable that is returned by the function.

Parameters

```
type function_name ( type param1, type param2, ... ) {
    **FUNCTION CODE**
}
```

return type

Braces enclose the code of the function.

A dumb example...

```
int add2Numbers ( int num1, int num2 )  {
        int sum = num1 + num2;
        return sum;
}
```

return ends the function, but not the whole program. Sends back a value.

Calling Functions

- When we call the function, we shouldn't write out the data type.
- But we should pass variables of the corresponding type.
- A dumb example...

```
int main ( ) {
        int x = 2;
        int y = 3;
        cout << "x+y= " << add2Numbers (x,y);
}
```

- The C++ compiler reads top to bottom. Since main calls add2Numbers, we should put the function add2Numbers *above* the main routine.

Putting the Program Together

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

int add2Numbers ( int num1, int num2 )  {
        int sum = num1 + num2;
        return sum;
}

int main ( )  {
        int x = 2;
        int y = 3;
        cout << "x+y= " << add2Numbers (x,y);
        return 0;
}
```

First include our libraries.

Write the function(s).

The main routine is almost always last.

Sec 4.3: Commenting Functions

- When you write a function, comment on it right above the function declaration.
- Comment on the function's purpose, parameters passed, and return value.

```
/************************************************
**  add2Numbers: Adds two numbers.
**  Parameters:
**      num1 -- The first number
**      num2 -- The second number
**  Returns the sum of the two numbers.
************************************************/
int add2Numbers ( int num1, int num2 )  {
        int sum = num1 + num2;
        return sum;
}
```

Why Use Functions?

- OK, I admit adding 2 numbers is a dumb example.
- We use functions when...
  - there's a procedure to repeat, so then we don't have to repeat code.
  - a procedure is particularly complex, so it helps our coding/debugging to separate it out.
  - we anticipate possibly using that procedure in the future, perhaps in another program.
- Think about our previous homeworks:
  - HW1: convertSeconds(seconds),
    computeMPH (dist,time)
  - HW2: dropTheS (word)
  - HW3: drawX(p) , drawO(p)

An Example From HW2

- Write a function to drop the "s" off the end of a word, converting plural to singular.

```
string dropTheS (string word)  {
        int len = word.length( );
        if ( word.substr(len-1,1) == "s" )
                return word.substr(0,len-1);
        else
                return word;
}
```

void Functions

- If a function doesn't return a value, give it the blank data type void.
- End the function with simply return;
- Suppose we want our dropTheS function to print to the screen, rather than returning the singular word.

```
void printSingular (string word)  {
        int len = word.length( );
        if ( word.substr(len-1,1) == "s" )
                cout << word.substr(0,len-1);
        else
                cout << word;
        return;
}
```

Example: Least Common Multiple

- We want to find the Least Common Multiple (LCM) of two positive integers.

  cout << least_common_multiple(6,8); prints 24
- Pick either of the numbers and examine ascending multiples. If the other number divides that multiple, we have found the LCM.

```
int least_common_multiple (int num1, int num2) {
    int multiplier = 1;
    while ( (multiplier*num1)%num2 != 0)
        multiplier++;
    return multiplier*num1;
}
```

Factorization Example

- We want to write a program that decomposes a number into its prime factors, sorted in order from smallest to biggest.

```
Enter a number: 6
The factors of 6 are 2  3
Enter a number: 40
The factors of 40 are 2  2  2  5
```

- Notice we have to count repetitions.

Factorization Example

- It would help if we had a function that returned the smallest divisor (>1) of a number.
- Returns the original number if no divisor found: the smallest divisor of a prime number is itself.

```
int findSmallestDivisor (int num)   {
        int d = 2;
        while ( num%d != 0 && d < num )  {
                d++;
        }
        return d;
}
```

- What does it return if passed 15? 7? 1?

Factorization Example

- How would main call the findSmallestDivisor function?

```
int main ( )  {
      int x;
      cout << "Enter a number: ";
      cin >> x;
      cout << "The factors of " << x << " are ";
      while ( findSmallestDivisor(x) <= x )  {
            cout << findSmallestDivisor(x) << "  ";
            x = x / findSmallestDivisor(x);
      }
      return 0;
}
```

It's a little wasteful to call a function more than necessary and the stopping condition is rather convoluted. But I want you to see that the function call can appear in many different ways: while conditions, output, arithmetic.

A Graphics Example

- For HW4, you are required to make a function that draws an object at a given location and scale.
- The prototype might look like:

```
void drawObject (Point position, double scale)
```

- Then to make the animation you would repeatedly call this function, changing the position and scale a little bit each time.