FUNCTIONS

Today

- What are functions and why to use them
  - I will use the terms "function" and "method" interchangeably.
- Library and programmer-defined functions
- Parameters and return values
- Reading: textbook Chapter 4

Advantages of functions

- *Modularity*
  - We can divide up a program into small, understandable pieces (kind of like steps in a recipe)
  - This makes the program easier to read
  - This makes the program easier to *debug*.
- *Write once, use many times*
  - If we have a task that will be performed many times, we only have to *define* a function once; then we can *call* (or *invoke*) the function as many times as we need it
  - Also, we can use *parameters* (or *arguments*) to use the function to perform the same task on or with different data values

Library functions

- We have already talked about *built-in*, or *library*, functions
  - These are functions that come with the C++ language
- We have used the *iostream* C++ library:
  - `iostream.cout`
  - `iostream.cin`
- We have used the *fstream* C++ library:
  - `ifstream.open`
  - `ifstream.close`
  - `ofstream.open`
  - `ofstream.close`

- We have also mentioned the *stdlib* C library:
  - `srand`
  - `rand`
- Soon we will come across the *math* C library:
  - `sqrt`
  - `pow`

How functions work

- Functions must be *defined* (or "declared") and then they can be *called* (or "invoked")
- In the file that contains a program, a function must be declared before it can be invoked.
- You can declare a function "header" (see the next slide) first and then later list the function definition; or you can simply put the function definition in the file before the function is called...

A first example

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

void sayHello() // define function
{
 cout << "hello" << endl;
 return 0;
}

int main()
{
  sayHello(); // call function
  return 0;
}
```


A second example

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

void sayHello(); // function header only

int main()
{
  sayHello(); // call function
  return 0;
}

void sayHello() // define function
{
  cout << "hello" << endl;
  return 0;
}
```

Components of a function definition

- *header*
  - Data type or `void`
  - Identifier
  - Argument list— contains *formal parameters* (also sometimes called *dummy* parameters)
- *body*
  - Starts with `{`
  - Contains statements that execute the task(s) of the function
  - Uses a `return` statement to return a value corresponding to the function's data type (unless the function is `void`, in which case there is no `return` statement or return value)
  - Ends with `}`

Function parameters

- *Call by value*: this means that when a function is called, the *value* of any function parameters are transferred to the inside of the function and used in there.
- The name of the dummy parameter is what is used inside the function, and its initial value is set to the value of the argument that is used when the function is called.


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

void sayHello( int n ) // n is a dummy parameter
{
 int i;
 for ( i=0; i<n; i++ )
 {
   cout << "hello\n";
   return 0;
 }
}

int main()
{
  sayHello( 3 ); // 3 is the value of the argument
  return 0;
}
```


- When the example runs, the dummy parameter `n` inside the function `sayHello` will be set to the value `3`, because that is the value of the argument when the function is called from the main program

Programmer-defined functions

- As in the previous example, you can define your own functions.
- You are not limited just to those functions already defined in the C++ language!
- Now the real fun begins!
- Note that we'll be using functions more or less continuously from here until the end of the course.
- If you don't understand them, then it is time to go back and make sure that they make sense.

Local variables

- In the last version of `sayHello` we declared a variable `i`.
- This variable is *local* to the function.
- That means it cannot be used outside of `sayHello`.
- We say that its *scope* is the function `sayHello`.
- Similarly, the scope of any variable defined in `main` is just the function `main`.
- (That's right, `main` is just a function.
- This is why we need arguments and return values — to pass information around programs.

Return values

- As in the previous slide, *return values* are good because they are a way of sending a value from inside a function back to the part of a program that called that function.
- Up until now, we have written functions that have a single `return` statement, typically `return 0`
- (Which means that the return value is 0).
- You can actually write a function that has multiple `return` statments if the function contains branching statements.

Example

```
int sign( double x )
{
  if ( x == 0 )
  {
    return 0;
  }
  else if ( x > 0 )
  {
    return 1;
  }
  else // x < 0
  {
    return -1;
  }
}
```

Example

- This example returns:
  - 0 if the function argument is equal to zero,
  - 1 if the function argument is positive, and
  - -1 if the function argument is negative

Multiple function parameters

- You can write functions that have more than one parameter.
- The parameters can be of any data type.
- The parameters can even be different data types.

First example

```
int add( int A, int B )
{
  int sum;
  sum = A + B;
  return sum;
}
```


Second example

```
int doMath( int A, int B, char op )
{
  int result;
  if ( op=='+' )
  {
    result = A + B;
  }
  else if ( op=='-' )
  {
    result = A - B;
  }
  else if ( op=='*' )
  {
    result = A * B;
  }
  return result;
} // end of doMath()
```

Summary

- This lecture has introduced functions.
- We briefly recapped library functions.
- Then considered user-defined functions.
- In our consideration of user-defined functions, we looked at:
  - Parameters.
  - Local variables
  - Return values