today: functions

- what are functions and why to use them
- library and programmer-defined functions
- parameters and return values
- reading: textbook chapter 5, sections 1-4

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
  - and easier to *debug* (i.e., check and see if it works, and fix it if it doesn't work)
- *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
- i.e., these are functions that come with the C++ or C language
- we have used the *math* C library:
  - `sqrt`
  - `pow`
- we have used the *stdlib* C library:
  - `srand`
  - `rand`
- we have used the *iostream* C++ library:
  - `iostream.cout`
  - `iostream.cin`

- we have used the *fstream* C++ library:
  - `ifstream.open`
  - `ifstream.close`
  - `ifstream.eof`
  - `ofstream.open`
  - `ofstream.close`
  - `ofstream.eof`

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...
- first example:

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

int sayHello() { // define function
 cout << "hello\n";
 return 0;
}

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


- second example:

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

int sayHello(); // declare function (header only)

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

int sayHello() { // define function
  cout << "hello\n";
  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
- example:

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

int 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 example on the previous slide, you can define your own functions
- you are not limited just to those functions already defined in the C and C++ languages!
- now the real fun begins!
- of course, we have already created functions in the first homework, but maybe you didn't know exactly what you were doing...

local variables

- so far, all the variables we have declared have been called *global*
- this is because we declared them *outside* of the `main()` or any other function
- but actually, you can declare variables *locally*, within the body of a function
- and this is more efficient than declaring variables globally
- the reason has to do with *memory allocation*, i.e., how much memory the computer uses while your program is running
- when a function runs, the program allocates memory for that function; when the function finishes, the program releases that memory so that it can be used again for something else
- so, local variables are only used inside the function, and they "go away" when the function exits
- which is why *return values* are handy—because you can send a value from the function back to the part of the program that called it; and that value does not go away when the function finishes

- memory for global variables is allocated when the program starts, and the memory is not released until the whole program finishes; so it is better to use local variables for storing values that you only need to use some of the time that a program is running

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)
- but 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;
  }
} // end of sign()
```

- 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; they can even be different data types
- first example:

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


- 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;
  }
  else {
    result = -999;
  }
  return result;
} // end of doMath()
```