## cis1.5 spring2009 lecture III.2

today we are going to talk about writing your own functions...

- how functions work
- components of a function definition
- function parameters (aka arguments)
- programmer-defined (your own) functions
- local variables
- return values
- multiple function parameters
- prototypes and headers



## how functions work

- functions must be **declared** and **defined** before they can be **called** (or "invoked")
- the example below shows a function that is declared and defined together, in the file where it is invoked:

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

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

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



- OR you can declare a function **prototype** (or "header") first and then later list the function definition, for example:

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

int sayHello(); // declare function (prototype or header)

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

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



## components of a function definition

- *prototype* or *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;
 }
} // end of sayHello()
```



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

- 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!



## return values

- *return values* provide a way of sending a value from inside a function back to the part of a program that called that function
- today we have written functions that have a single `return` statement, typically

  ```
  return 0;
  ```

  (which means that the return value is $0$)
- the function can also return a number other than $0$ or other types of values, if the function's data type is something other than `int`
- NOTE that we have been lazy about writing the `main()` function, whose data type is `int`, by not specifying a `return` statement; technically, we should have been writing:

```
int main() {
  cout << "hello world\n";
  return 0;
} // end of main()
```

- but the return value for `main()` can be treated specially, because technically the value is returned to the operating system (rather than to another C++ function that calls it)

- you can write a function that has multiple `return` statments if the function contains branching statements
- for 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()
```



## function prototypes

- as with variables, functions have to be *declared* before they can be used
- a function *declaration* is where you specify the following:
  - name of the function
  - data type of the function (i.e., data type of return value)
  - any function parameters/arguments and their data types
- a function *definition* is where you specify the content of the function, or its *body* — i.e., what the function actually DOES
- in the example above, we *declared* and *defined* the function at once, because (1) we placed the function definition in the same file where the function is called, and (2) we placed the function definition in the file ahead of where it is called
- you can also use something called a *function prototype* which just contains the *declaration* information, and then you can put the *definition* somewhere else (e.g., later in the file or even in another file)
- in fact, the `#include` statements that you have been using are ways of including in your code the content of a *header* file—which contains function prototypes

- here is an example of a function prototype (i.e., function declaration):

```
int sayHello();
```

and here is the corresponding definition:

```
int sayHello() {
 cout << "hello\n";
 return 0;
}
```