cisc1110 fall 2010 lecture VI.1

- functions
- built-in/library functions:
  - cmath
  - cctype
- writing your own functions
- return values
- function parameters
- value parameters and reference parameters

library functions

- we have already used some library functions
- iostream C++ library:
  - iostream.cout
  - iostream.cin
- string C++ library:
  - string.length()
  - string.find()
  - string.replace()
  - string.insert()
- stdlib C library:
  - srand()
  - rand()

cmath library

- there is a standard library of useful math functions defined in C that is commonly used in C++
- the header file:
  #include <cmath>
  using namespace std;
- these include, for example:
  - double sqrt( double x )
  - double pow( double x, double y )
  - double sin( double x )
- these take arguments and return values, e.g.:
  double f = sqrt( 4.0 );
- many other functions are defined in math.h, including trigonometry functions (like sin, cos, tan), and constants like MATH_PI
- on-line reference for cmath:
  http://www.cplusplus.com/reference/clibrary/cmath/

example:

```cpp
class C { public:   
    C() {   
        if ( is_correct )    
            return;   
        else    
            throw std::runtime_error( "bad construction in C" );   
    } 

private: 
   int m_data; 
}; 

int main() {
    C c; 
    return 0; 
}
```

```cpp
class C { public:   
    C() {   
        if ( is_correct )    
            return;   
        else    
            throw std::runtime_error( "bad construction in C" );   
    } 

private: 
   int m_data; 
}; 

int main() {
    C c; 
    return 0; 
}
```
cctype

- There is a standard library of useful character functions defined in C that is commonly used in C++
- The header file:
  ```
  #include <cctype>
  using namespace std;
  ```
- `int isalnum(int c)` checks if character argument is alphanumeric
- `int isalpha(int c)` checks if character argument is alphabetic
- `int isdigit(int c)` checks if character argument is a decimal digit
- `int islower(int c)` checks if character argument is a lowercase letter
- `int ispunct(int c)` checks if character argument is a punctuation character
- `int isupper(int c)` checks if character argument is an uppercase letter
- `int tolower(int c)` converts uppercase letter argument to lowercase
- `int toupper(int c)` converts lowercase letter argument to uppercase

On-line reference for cctype:
http://www.cplusplus.com/reference/clibrary/cctype/

Example:
```
#include <iostream>
#include <cctype>
using namespace std;

int main() {
  bool q = false;
  char c;
  while (!q) {
    cout << "Enter a character (q to quit): ";
    cin >> c;
    cout << "You entered: " << c << endl;
    if (islower(c)) {  // convert lowercase character to uppercase
      c = toupper(c);
    }
    cout << "Upper case = " << c << endl;
    q = (c == 'Q');
  }  // end while
}  // end of main()
```

Writing your own 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

How functions work

- Functions must be declared and defined before they can be called (or “invoked”); first you can declare a function prototype (or “header”) and then later list the function definition; you can invoke the function anywhere in the code after you have listed the prototype
- Example:
  ```
  #include <iostream>
  using namespace std;

  // Declare function (prototype or header)
  void sayHello();

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

  // Define function
do {
    cout << "Hello, world!";
  }  // end while
}
```
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 }

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
  
  ```c
  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:
  ```c
  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)

functions and parameters

- you can write a function that has multiple *return* statements IF the function contains *branching* statements
- for example:
  ```c
  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
- **the data type of the return value has to match the data type in the function definition!**
value parameters example

- **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.

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

int sayHello( int );

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

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

- When the above 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.

another call by value example:

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

int add3( int );

int main() {
    int p = 7, sum;
    sum = add3( p );
    cout << "sum=\n" << sum << endl;
}

int add3( int a ) {
    int ret;
    ret = a + 3;
    return( ret );
}
```

reference parameters example

- **call by reference**—this means that when a function is called, the reference to any function parameters is transferred to the inside of the function and used in there.
- The `&` preceding `sum` in the function header—this indicates that it is a reference parameter.

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

void add3( int & );

int main() {
    int sum = 7;
    add3( sum );
    cout << "sum=\n" << sum << endl;
}

void add3( int &sum ) {
    sum = sum + 3;
}
```