What you should know about C++

Lecture 1.5: CMSC 420

C++

- Almost every C program is a C++ program.
- That was an explicit goal of the design of C++.
- Why are we covering C++?
  - Important language, nearly the *lingua franca* of computer science.
  - Languages like Python, Ruby, C#, Java are used more and more, but there's still a huge C++ code base.
  - Many features of C++ are designed to support abstract data types and general data structures.
- If you want, think of C++ as C with some features to make your life easier.

Java

```
class IntStack {

   public IntStack(int max) {
      stack = new int[max];
      top = -1;
   }

   protected void finalize() {
      // nothing to do here
   }

   public void push(int k) {
      stack[++top] = k;
   }

   // ...
   protected int [] stack;
   protected int top;
};
```

Constructor

Finalize() may or may not be called when instance is garbage collected

Major Addition to C: Classes

```
class IntStack {

  public:
    IntStack(int max=100);
    ~IntStack();

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

  public:
    int size();
};
```

Constructor has same name as class. Called when object is created.

Destructor called ~ClassName; Called when object is deleted.

Functions declared inside class are called member functions. They can access the data in the class.

protected things can only be seen by subclasses; **public** things can be seen by everyone; **private** things can only be seen by this class.

Comparison to Java

C++

```
class IntStack {

  public:
    IntStack(int max=100);
    ~IntStack();

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

  public:
    int size();
};
```

Java

```
class IntStack {

    public IntStack(int max) {
        stack = new int[max];
        top = -1;
    }

    protected void finalize() {
        // nothing to do here
    }

    public void push(int k) {
        stack[++top] = k;
    }

    // ...
    protected int [] stack;
    protected int top;
};
```

new and delete operators

- **new** operator similar to Java
- **new** and **delete** in C++ return explicit pointers.
  - Java: **int**[] A = **new int**[3];
  - C++: **int** * A = **new int**[3];
  - C++: Node * n = **new** Node;
- In Java, there's garbage collection. In C++, have to delete explicitly:
  - C++: **delete** [] A;
  - C++: **delete** myStack;

Classes – function implementations

```
IntStack::InStack(int max=100)
{
   stack = new int[max];
   top = -1;
}

IntStack::~IntStack()
{
   delete [] stack;
}

void IntStack::push(int k)
{
   top++;
   stack[top] = k
}
```

Syntax "new TYPE[SIZE]" creates a new array of length SIZE containing objects of type TYPE.

"delete [] X" frees the memory for the array pointed to by X.

To free a single object, omit the "[]".

Member functions can access class variables without any special syntax.

Classes – Example use

- Stored as a local variable:

```
{
    IntStack S(10000);
    S.push(10);
    S.push(12);
} // ~InStack automatically called
```

- Stored on heap:

```
{
    IntStack * S = new IntStack(10000);
    S->push(10);
    S->push(12);
    delete S;
}
```

Structures

- In C++ structures are just classes where everything is public by default:

```
struct Foo { ...};
class Foo { public: ...};
```

- Syntax a little nicer for C++ structures (e.g. can include constructors):

C

```
struct A {
    int key;
    struct A * next;
};

struct A myrecord;
myrecord.key = 10;
```

C++

```
struct A {
    int key;
    A * next;
    A(int k) {key = k;}
};

A myrecord(10);
```

I/O

- You can use all C functions for input or output.
- OR you can use C++ *streams* (but don't mix the two).
- Standard streams:
  - stdin is called cin.
  - stdout is called cout.

- Reading values from *stdin* (whitespace is ignored):

```
int i;
float f;
string s;
cin >> i >> f >> s;
```

- Writing same to *stdout*:

```
cout << i << " " << f << " " << s << endl;
```

Strings

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

int main() {
    string s = "abcdefg";
    string s2 = "cat";

    cout << s[0] << s[2] << endl;   // "ac"
    s.append(s2);
    cout << s << endl;              // "abcdefgcat"
    s.insert(2, s2);
    cout << s << endl;              // "abcatcdefgcat"
    cout << s.find("tcd");          // 4
}
```

A "make it work" instruction.

http://www.sgi.com/tech/stl/basic_string.html

References

- A way to give the same variable several names.
- Value of a reference must be specified when it is created and can never be changed.
- It's like a pointer that always points to the same variable, and the dereferencing operation (*x) is automatic.

```
int x = 10;
int & y = x;

cout << y;          // prints 10
x = 30;
cout << y;          // prints 30
y = 72;
cout << y;          // prints 72
cout << x;          // prints 72
```

References – Pass by Reference

- References are most commonly used to pass variables to functions so that the function can change them:

```
int add1(int * x) { (*x) += 1; }      /* C-style */

int add1(int & x) { x += 1; }         // C++-style
```

- Common case: want to pass a big object to a function, so don't want to copy, but want to be sure object isn't changed:

```
int foo(const Image & pict) {/*...*/}
```

Operator Overloading

```
struct Point {
    int x, y;
    Point(int xx, int yy) { x=xx; y=yy; }
};

bool operator==(const Point & A, const Point & B)
{
     return A.x == B.x && A.y == B.y;
}

int main() {
    Point p1(10, 4);
    Point p2(-12, -100);
    Point p3(10, 4);

    if(p1 == p2) { /* FALSE */ }
    if(p1 == p3) { /* TRUE */ }
}
```

Variable Declarations

- Can declare variables in the middle of blocks: e.g. put "**int** x;" any place you can have a statement:

```
{
    int i;
    // some code
    int j;
    // more code
}
```

- Also, can declare variables inside initialization section of for loops:

C

```
int i;
for(i=0; i < len; i++)
   total += X[i]
```

C++

```
for(int i=0; i < len; i++)
   total += X[i]
// i not visible after loop
```

Minor Differences From C

- Comments: // until the end of line (in addition to /* */)
- **bool** is a built-in type, with values **true** and **false**.
- **namespace**s: collect functions into groups. Probably you'll only use to say:

```
using namespace std;
```

- Doesn't support `int foo(a,b) int a, int b { /* ... */ }` syntax.
- Function arguments can have default values:
  `int foo(int a=0) { /* ... */ }.`
- "g++" instead of "gcc", .cc extension instead of .c

Other differences

- Templates: write code to work with any type of variables.
  - In practice, can be hard to get to work right.
  - Won't need for this class (but can use if you want).

- **const** modifier means variable cannot be changed.
  - Good idea in theory, except that const "infects" everything it touches
  - e.g. can't pass a const variable to any function that hasn't explicitly labeled the parameter const.
  - Just as well to avoid using it.

Resources

- SGI STL documentation:
  http://www.sgi.com/tech/stl/table_of_contents.html

- C++ Tutorial:
  http://www.otal.umd.edu/drweb/c++tutorial/