What you need to know about C++

Lecture 7

Zia Khan

Slides adapted from from Carl Kingsford at CMU.

C++ = C with a bunch of useful features and more

- Object oriented programming
- Abstract data types (templates/containers)
  - STL library
  - Qt library containers
- Operator Overloading
- You can still program in C-style using only a subset of features C++ provides. ;-)
- Useful code bases like the Qt library
- Many new modern programming features in

C++11 and C++14 (not covered today, worth looking into on your own)

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++

```cpp
class IntStack {

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

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

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

In C++ access specifies are not per declaration like in Java.

Java

```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;
};
```

Classes – function implementations

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

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

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

"delete [] X" frees the memory for the array pointed to by X.
To free a single object, omit the "[]".

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

Member functions can access class variables without any special syntax.

In C++ class declarations are usually in a separate file from member function implementations

Class declaration in **`IntStack.hpp`**.

```
class IntStack {

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

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

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

Member function implementation **`IntStack.cpp`**

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

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

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

You can include the implementation in the declaration like Java, but if you're not careful you will run into annoying dependency problems with your header files.

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 – 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;
}
```

Classes are not references in C++: Assignment Operator

```
IntStack A;
IntStack B;

B = A;
```

What does this do?

```
class IntStack {

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

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

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

Classes are not references in C++: Assignment Operator

```
IntStack A;
IntStack B;

B = A;
```

What does this do?

Calls default assignment operator that copies the pointer from A's stack variable to B's stack variable.

Also you've lost the reference to B's stack variable. Memory leak!!!!!

```
class IntStack {

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

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

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

Classes are not references in C++: Copy Constructor

```
IntStack A;
IntStack B = A;

A.push(10);
B.push(15);
```

The value of A.top() is 10, right?

```
class IntStack {

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

    void push(int);
    int pop();

  protected:
    int * stack;
    int top;

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

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

A “make it work” instruction.

```
#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
}
```

http://www.cplusplus.com/reference/string/

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 */ }
}
```

Operator Overloading == “syntactic sugar” and why your Java programs sometimes look ugly ;-)

```
struct Point {
    int x, y;
    Point(int xx, int yy) {
        x = xx; y = yy;
    }
    float operator += (const float a){
        x += a; y += a;
    }
};

Point  operator + (const Point &a, float s)
{
    Point r = a;
    r.x += s; r.y += s;
    return r;
}
```

```
{
    Point C (5,10)
    Point Q = C + 5;
    Q -= 5;
}
```

Can you guys tell me what is Q?

Same operator precedence rules apply to overloaded operators.

Mainly, you should know how to use overloaded operators to make your code easy to read.

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) { /* ... */ }.`

- Name header files with **.hpp** extension instead of **.h** extension.
- Name source files with **.cpp** extension instead of **.c** extension.

Including C code in C++ with extern “C”

Function and global declarations in **`Foobar.h`**.

```
#ifdef  __cplusplus
extern "C"
{
#endif
    void foo(int);
    void g(char);
    int i;
#ifdef __cplusplus
}
#endif
```

Function implementations and globals in **`Foobar.c`**

```
int i = 382; /* C global */
/* my C function g */
void g(char x) {
  if(x == 'Z')
      printf("Zia");
}
/* my cmsc427 C code */
void foo(int x) {
  if(x == 427) printf("foobar");
}
```

Will tell the linker that Foobar.c was compiled in C.
C++ mangles function names.

Initializing Member Classes (IMPORTANT!!!)

```
class MyApplication {
  MyApplication();
  void run();
protected:
  IntStack small_stack;
  IntStack big_stack;
  int first_val;
};
```

```
void MyApplication::run() {
  small_stack.push(first_val);
  big_stack.push(first_val);
  small_stack.push(427);
  first_val = small_stack.pop();
}
```

```
MyApplication::MyApplication() :
  small_stack(5),
  big_stack(10000)  {
  first_val = 330;
}
```

Initializer list.
Mandatory if there is no default constructor for the class.
Within member functions, can use as if defined on stack.

```
MyApplication::~MyApplication(){
  // destructor here
}
```

In the destructor, ~IntStack() is called for each automatically.

Member classes as pointers allocated on heap (IMPORTANT!!!)

```
class MyApplication {
  MyApplication();
  void run();
protected:
  IntStack *small_stack;
  IntStack *big_stack;
  int first_val;
};

MyApplication::MyApplication() {
  small_stack = new IntStack(5);
  big_stack = new IntStack(10000);
  first_val = 330;
}

MyApplication::~MyApplication(){
  delete small_stack;
  delete big_stack;
}
```

```
void MyApplication::run() {
  small_stack->push(first_val);
  big_stack->push(first_val);
  small_stack->push(427);
  first_val = small_stack->pop();
}
```

Now the classes are allocated on the heap.
Public member functions can be called using the "->" operator.

Note that now you need to call `delete` explicitly to free member class memory.

Using C++ templates (IMPORTANT, will make your life easier)

```
#include "testing.h"

#include <vector>
using namespace std;
class MyApplication {
    MyApplication();
    void run();
protected:
    vector<int> my_stack;
    int first_val;
};

MyApplication::MyApplication() {
    first_val = 330;
}

void MyApplication::run() {
    my_stack.push_back(first_val);
    cout << my_stack.back() << endl;
}
```

STL library provides many useful templates and containers e.g. vector, map, set, etc.

On the surface a bit like Java "generics" but totally different.

Your mental model should be a very capable pre-processor system (e.g. `#define`).

Gotcha associated with using C++ template functions

```
template <typename T>
T my_min(T A, T B) {
  if(A <= B) return A; else return B;
}
```

The compiler will try to infer types by itself for functions. Type-checker will complain about this since types don't match. 32.0 is a `double` by default.

```
float x = 3;
cout << my_min(x, 32.0) << endl;
```

Fix #1

```
float x = 3;
cout << my_min(x, (float)32.0) << endl;
```

Fix #2

```
float x = 3;
cout << my_min(x, 32.0f) << endl;
```

Familiar with Java packages, try C++ namespaces

Useful to avoid naming conflicts.

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

namespace foo
{
    int value() { return 5; }
}

namespace bar
{
    const double pi = 3.1416;
    double value() { return 2*pi; }
}

int main () {
    cout << foo::value() << '\n';
    cout << bar::value() << '\n';
    cout << bar::pi << '\n';
    return 0;
}
```

using keyword allows you to access all members without the :: syntax.

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

See resources on
https://www.cs.umd.edu/class/fall2014/cmsc427/