6.088 Intro to C/C++

Day 6: Miscellaneous Topics

Eunsuk Kang & Jean Yang
In the last lecture...

Inheritance

Polymorphism

Abstract base classes
Today’s topics

Polymorphism (again)

Namespaces

Standard Template Library

Copying objects

Integer overflow
Polymorphism revisited
Polymorphism revisited

Recall: Ability of type A to appear as or be used like another type B.

Example:

```cpp
Rectangle* r = new Square(5.0); // length = 5.0
```

where Square is a subtype of Rectangle
Barbara Liskov
Winner, Turing Award 08’

If S is a subtype of T, then the behavior of a program P must remain unchanged when objects of type T are replaced with objects of type S.
Rectangle-Square example

class Rectangle {
    protected:
        float length;
        float width;
    public:
        Rectangle(float length, float width);
        void setLength();
        void setWidth();
        void getLength();
        void getWidth();
    }

class Square : public Rectangle {
    // representation invariant: length = width
    public:
        Square(float length);
        void setLength(); // ensures length = width
        void setWidth(); // does nothing
    }
Rectangle-Square example

class Rectangle {
    protected:
        float length;
        float width;
    public:
        Rectangle(float length, float width);
        void setLength();
        void setWidth();
        void getLength();
        void getWidth();
}

class Square : public Rectangle {
    // representation invariant: length = width
    public:
        Square(float length);
        void setLength();  // ensures length = width
        void setWidth();   // does nothing
}
Solutions

*Ugly:* Modify `setWidth` and `setLength` in `Rectangle` to return a boolean. Make them return “true” in `Rectangle`, and “false” in `Square`. Define a separate method “`setDimension`” in `Square`.

*Better:* Maybe `Square` shouldn’t really be a subtype of `Rectangle`? Change the type hierarchy!

Think about behaviors, not just characteristics of an object!
Solutions

Ugly: Modify setWidth and setLength in Rectangle to return a boolean. They always return “true” in Rectangle, and “false” in Square. Define a separate method “setDimension” in Square.

Better: Maybe Square shouldn’t really be a subtype of Rectangle? Re-think the type hierarchy!

Think about behaviors, not just characteristics of an object!
Solutions

**Ugly:** Modify `setWidth` and `setLength` in `Rectangle` to return a boolean. They always return “true” in `Rectangle`, and “false” in `Square`. Define a separate method “`setDimension`” in `Square`.

**Better:** Maybe `Square` shouldn’t really be a subtype of `Rectangle`? Re-think the type hierarchy!

Think about **behaviors**, not just characteristics of an object!
Namespaces
Namespaces

- an abstract space that contains a set of names
- useful for resolving naming conflicts

```cpp
namespace ford {
    class SUV {
        ...
    }
};
}
namespace dodge {
    class SUV {
        ...
    }
};

int main() {
    ford::SUV s1 = new ford::SUV();
    dodge::SUV s2 = new dodge::SUV();
    ...
}
```
Using namespaces

Use with caution!

```cpp
namespace ford {
    class SUV {
        ...
    };
    class Compact {
        ...
    };
}

int main() {
    using namespace ford;  // exposes SUV and Compact
    SUV s1 = new SUV();
    ...
}
```
Using namespaces

Expose only the things that you need to use!

namespace ford {
    class SUV {
        ...
    }
    class Compact {
        ...
    }
}

int main() {
    //using namespace ford;
    using ford::SUV;
    SUV s1 = new SUV();
    ...
}
C++ standard library & namespace

C++ standard library includes:

- string (std::string s = “Hello World!”)
- vector (std::vector<T> ...)
- iostream (std::cout << ...
- and many other things!

The library lives inside the namespace “std”
Using namespace std

#include <string>

class MITPerson {

protected:
    int id;
    std::string name;
    std::string address;

public:

    MITPerson(int id, std::string name, std::string address);

    void displayProfile();
    void changeAddress(std::string newAddress);

};
#include <string>

class MITPerson {

protected:
   int id;
   std::string name;
   std::string address;

public:
   MITPerson(int id, std::string name, std::string address);
   void displayProfile();
   void changeAddress(std::string newAddress);
};
Using namespace std

#include <string>

class MITPerson {

protected:
    int id;
    std::string name;
    std::string address;

public:

    MITPerson(int id, std::string name, std::string address);
    void displayProfile();
    void changeAddress(std::string newAddress);

};
Using namespace std

```cpp
#include <string>

using namespace std;

class MITPerson {

    protected:
        int id;
        string name;
        string address;

    public:

        MITPerson(int id, string name, string address);
        void displayProfile();
        void changeAddress(string newAddress);
};
```
Using namespace std - Be aware!

This is potentially dangerous. Why?

Rules of thumb:

- "using std::string" instead of "using namespace std"
- include "using..." only in the .cc file, not in the header (why?)

Simplest rule: Just type "std::". Sacrifice a few extra keystrokes in the name of good!
Standard Template Library
Standard Template Library (STL)

- a set of commonly used data structures & algorithms
- parameterized with types

Some useful ones include:
- vector
- map
- stack, queue, priority_queue
- sort

More available at:
http://www.cppreference.com/wiki/stl/start
Example using vectors

- an array with automatic resizing

```cpp
std::vector<T> v; // creates an empty vector of type T elements
std::vector<int> v2(100); // creates a vector with 100 ints
std::vector<T> v3(100); // creates a vector with 100 elements
```

primitives initialized to some default value

objects created using default constructor
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;        // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);
};
A list of classes as a vector

```cpp
#include <iostream>
#include <vector>
#include "MITPerson.h"
#include "Class.h"

class Student : public MITPerson {

    int course;
    int year;  // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;

public:
    Student(int id, std::string name, std::string address,
            int course, int year);
    void displayProfile();
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);
};
```

don’t forget!

declares an empty vector of pointers to Class objects
Vector operations

Class* c1 = new Class("6.01");
Class* c2 = new Class("6.005");

// inserting a new element at the back of the vector
classesTaken.push_back(c1);
classesTaken.push_back(c2);

// accessing an element
Class* c3 = classesTaken[0];
Class* c4 = classesTaken.at(1);
std::cout << c3.getName() << "\n";  // prints "6.01"
std::cout << c4.getName() << "\n";  // prints "6.005"

// removing elements from the back of the vector
classesTaken.pop_back();
classesTaken.pop_back();

// checking whether the vector is empty
if (classesTaken.empty()) std::cout << "Vector is empty!\n";
Vector operations

```cpp
Class* c1 = new Class("6.01");
Class* c2 = new Class("6.005");

// inserting a new element at the back of the vector
classesTaken.push_back(c1);
classesTaken.push_back(c2);

// accessing an element
Class* c3 = classesTaken[0];
Class* c4 = classesTaken.at(1);
std::cout << c3->getName() << "\n"; // prints "6.01"
std::cout << c4->getName() << "\n"; // prints "6.005"

// removing elements from the back of the vector
classesTaken.pop_back();
classesTaken.pop_back();

// checking whether the vector is empty
if (classesTaken.empty()) std::cout << "Vector is empty!\n";
```
Vector operations

Class* c1 = new Class("6.01");
Class* c2 = new Class("6.005");

// inserting a new element at the back of the vector
classesTaken.push_back(c1);
classesTaken.push_back(c2);

// accessing an element
Class* c3 = classesTaken[0];
Class* c4 = classesTaken.at(1);
std::cout << c3.getName() << "\n"; // prints “6.01”
std::cout << c4.getName() << "\n"; // prints “6.005”

// removing elements from the back of the vector
classesTaken.pop_back();
classesTaken.pop_back();

// checking whether the vector is empty
if (classesTaken.empty()) std::cout << “Vector is empty!\n”;
Vector operations

Class* c1 = new Class("6.01");
Class* c2 = new Class("6.005");

// inserting a new element at the back of the vector
classesTaken.push_back(c1);
classesTaken.push_back(c2);

// accessing an element
Class* c3 = classesTaken[0];
Class* c4 = classesTaken.at(1);
std::cout << c3.getName() << "\n"; // prints "6.01"
std::cout << c4.getName() << "\n"; // prints "6.005"

// removing elements from the back of the vector
classesTaken.pop_back();
classesTaken.pop_back();

// checking whether the vector is empty
if (classesTaken.empty()) std::cout << "Vector is empty!\n";
Traversing a vector using an iterator

```
// display a list of classes taken by the student

// create an iterator
std::vector<Class*>::iterator it;

std::cout << "Classes taken:\n";

// step through every element in the vector
for (it = classesTaken.begin(); it != classesTaken.end(); it++){
    Class* c = *it;
    std::cout << c->getName() << "\n";
}
```

Classes taken:
6.01
6.005
Traversing a vector using an iterator

```cpp
// display a list of classes taken by the student

// create an iterator
std::vector<Class*>::iterator it;
std::cout << "Classes taken:\n";

// step through every element in the vector
for (it = classesTaken.begin(); it != classesTaken.end(); it++){
    Class* c = *it;
    std::cout << c->getName() << "\n";
}
```
Traversing a vector using an iterator

```cpp
// display a list of classes taken by the student

// create an iterator
std::vector<Class*>::iterator it;

std::cout << "Classes taken:\n";

// step through every element in the vector
for (it = classesTaken.begin(); it != classesTaken.end(); it++){
    Class* c = *it;
    std::cout << c->getName() << "\n";
}
```

**it** a pointer to an element

***it** the element
There are many other functions

<table>
<thead>
<tr>
<th>Constructors</th>
<th>create vectors and initialize them with some data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>compare, assign, and access elements of a vector</td>
</tr>
<tr>
<td>assign</td>
<td>assign elements to a vector</td>
</tr>
<tr>
<td>at</td>
<td>return a reference to an element at a specific location</td>
</tr>
<tr>
<td>back</td>
<td>returns a reference to last element of a vector</td>
</tr>
<tr>
<td>begin</td>
<td>returns an iterator to the beginning of the vector</td>
</tr>
<tr>
<td>capacity</td>
<td>returns the number of elements that the vector can hold</td>
</tr>
<tr>
<td>clear</td>
<td>removes all elements from the vector</td>
</tr>
<tr>
<td>empty</td>
<td>true if the vector has no elements</td>
</tr>
<tr>
<td>end</td>
<td>returns an iterator just past the last element of a vector</td>
</tr>
<tr>
<td>erase</td>
<td>removes elements from a vector</td>
</tr>
<tr>
<td>front</td>
<td>returns a reference to the first element of a vector</td>
</tr>
<tr>
<td>insert</td>
<td>inserts elements into the vector</td>
</tr>
<tr>
<td>max_size</td>
<td>returns the maximum number of elements that the vector can hold</td>
</tr>
<tr>
<td>pop_back</td>
<td>removes the last element of a vector</td>
</tr>
<tr>
<td>push_back</td>
<td>add an element to the end of the vector</td>
</tr>
<tr>
<td>rbegin</td>
<td>returns a reverse_iterator to the end of the vector</td>
</tr>
<tr>
<td>rend</td>
<td>returns a reverse_iterator to the beginning of the vector</td>
</tr>
<tr>
<td>reserve</td>
<td>sets the minimum capacity of the vector</td>
</tr>
<tr>
<td>resize</td>
<td>change the size of the vector</td>
</tr>
<tr>
<td>size</td>
<td>returns the number of items in the vector</td>
</tr>
<tr>
<td>swap</td>
<td>swap the contents of this vector with another</td>
</tr>
</tbody>
</table>

Courtesy of C++ Reference. Used with permission.

Copying objects
Objects as values

So far we’ve dealt mostly with pointers to objects.

What if you want to pass around objects by value? For example,

```c
void print(MITPerson p){
    p.displayProfile();
}

int main() {
    MITPerson p1(921172, “James Lee”, “32 Vassar St.”);
    MITPerson p2 = p1;
    print(p2);
}
```
Creating an object from another

1. initialization by value, so make a copy

```java
void print(MITPerson p){
    p.displayProfile();
}
```

2. pass by value, so make a copy

```java
int main() {
    MITPerson p1(921172, "James Lee", "32 Vassar St.");
    MITPerson p2 = p1;
    print(p2);
}
```

(3. could also return an object as a return value)
void print(MITPerson p) {
    p.displayProfile();
}

int main() {
    MITPerson p1(921172, "James Lee", "32 Vassar St.");
    MITPerson p2 = p1;
    print(p2);
}

So how do objects get copied?
Copy constructors are called.

Object::Object(const Object& other) {
    ...
}
Copy constructor in MITPerson

```cpp
#include <string>

class MITPerson {

protected:
    int id;
    std::string name;
    std::string address;

public:
    MITPerson(int id, std::string name, std::string address);
    MITPerson(const MITPerson& other);
    void displayProfile();
    void changeAddress(std::string newAddress);
};
```
Defining the copy constructor

why const?  object that you are copying from

MITPerson::MITPerson(const MITPerson& other) {
    name = other.name;
    id = other.id;
    address = other.address;
}

Default copy constructor

- automatically generated by the compiler
- copies all non-static members (primitives & objects)
- invokes the copy constructor of member objects

```cpp
MITPerson::MITPerson(const MITPerson& other){
    name = other.name;
    id = other.id;
    address = other.address;
}
```
Assigning an object to another

```cpp
MITPerson p1(921172, "James Lee", "32 Vassar St.");
MITPerson p2(978123, "Alice Smith", "121 Ames St.");
p2 = p1; // assigns p2 to p1, does NOT create a new object
```

So how do objects get assigned to each other? **Copy assignment operator** is called.

```cpp
Object& Object::operator=(const Object& other) {
    ...
}
```
Copy assignment operator in MITPerson

```cpp
#include <string>

class MITPerson {

protected:
    int id;
    std::string name;
    std::string address;

public:

    MITPerson(int id, std::string name, std::string address);
    MITPerson(const MITPerson& other);
    MITPerson& operator=(const MITPerson& other);
    void displayProfile();
    void changeAddress(std::string newAddress);
};
```
Defining the copy assignment operator

MITPerson& MITPerson::operator=(const MITPerson& other){
    name = other.name;
    id = other.id;
    address = other.address;

    return *this; // returns a newly assigned MITPerson
}

MITPerson p1(921172, “James Lee”, “32 Vassar St.”);
MITPerson p2(978123, “Alice Smith”, “121 Ames St.”);
p2 = p1;
Defining the copy assignment operator

```cpp
MITPerson& MITPerson::operator=(const MITPerson& other){
    name = other.name;
    id = other.id;
    address = other.address;

    return *this;
}
```

Again, if you don’t define one, the compiler will automatically generate a copy assignment operator.

So why do we ever need to define copy constructors & copy assignment operators ourselves?
Default copy constructor - caution!

class B {
    public:
    void print() { std::cout << "Hello World!\n"; }
};

class A {
    B* pb;
    int x;
    public:
    A(int y) : x (y) { pb = new B(); }
    ~A() { delete pb; } // destructor
    void printB() { pb->print(); }
};

void foo(A a) {
    a.printB();
}

int main() {
    A a1(5);
    a1.printB();
    foo(a1);
    return 0;
}
Default copy constructor - caution!

```cpp
class B {
public:
    void print() { std::cout << "Hello World!\n"; }
};

class A {
    B* pb;
    int x;
public:
    A(int y) : x(y) { pb = new B(); }
    ~A() { delete pb; } // destructor
    void printB() { pb->print(); }
};

void foo(A a) {
    a.printB();
}

int main() {
    A a1(5);
    a1.printB();
    foo(a1);
    return 0;
}
```

Double free!
How do we fix this?
Default copy constructor - caution!

class B {
    public:
    void print() { std::cout << "Hello World!\n"; }
};

class A {
    B* pb;
    int x;
    public:
    A(int y) : x(y) { pb = new B(); }
    A(const A& other) { // copy constructor
        x = other.x;
        pb = new B();
    }
    ~A() { delete pb; } // destructor
    A& operator=(const A& other) { // copy assignment operator
        x = other.x;
        delete pb; // clean up the junk in the existing object!
        pb = new B();
        return *this;
    }
    void printB() { pb->print(); }
};
Rule of three in C++

If you define any one of the three in a class, then you should define all three (you will probably need them!)

- destructor
- copy constructor
- copy assignment operator
Integer overflow
int binarySearch(int a[], int key, int length) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found
}

Courtesy of Joshua Bloch. Used with permission.

based on from Joshua Bloch’s implementation in java.util
Binary search bug

int binarySearch(int a[], int key, int length) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found
}

Can you find the bug?

Courtesy of Joshua Bloch. Used with permission.
Binary search bug

```c
int binarySearch(int a[], int key, int length) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found
}
```

- If `(low + high) > MAX_INTEGER`, it will overflow!
- Dangerous array access! (Java will at least throw an exception)

Courtesy of Joshua Bloch. Used with permission.
One solution

Instead of:

```java
int mid = (low + high) / 2;
```

use:

```java
int mid = low + (high - low) / 2;
```

Surprisingly, most implementations of the binary search tree had this bug! (including java.util)

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
Conclusion
Useful links

Google C++ style guideline

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

C++ FAQ

There are many things we haven’t told you!

Thinking in C++ (B. Eckel) Free online edition!

Essential C++ (S. Lippman)

Effective C++ (S. Meyers)

C++ Programming Language (B. Stroustrup)

Design Patterns (Gamma, Helm, Johnson, Vlissides)

Object-Oriented Analysis and Design with Applications (G. Booch, et. al)
Congratulations!

Now you know enough about C/C++ to embark on your own journey!