COMP322 - Introduction to C++

Winter 2011

Lecture 07 - Overloading cont'd & The Basics of Inheritance

Milena Scaccia

School of Computer Science
McGill University

March 1, 2011

Last Time: Function Overloading

```
void print(int x, int radix = 10) {
  cout << setbase(radix) << x << endl;
}

void print(double x, int precision = 6) {
  cout << setprecision(precision) << x << endl;
}

int main() {
  print(10,16); // calls first print
  print(3.14159); // calls second print
}
```

Overloading operators

Almost all of the unary and binary operators in C++ can be redefined for a particular class.

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Non-Overloadable operators : `.` `::` `?:` `sizeof`

Operator overloading example

```
class complex {
private:
  float real, imag;
public:
  complex(float r=0, float i=0) {
    real = r;
    imag = i;
  }

  float getReal() const { return real; }
  float getImag() const { return imag; }

  complex operator+(const complex &y) const { //overload +
    return complex(real + y.real, imag + y.imag);
  }

  complex operator*(const complex &y) const { //overload *
    return complex(real * y.real - imag * y.imag,
                   real * y.imag + imag * y.real);
  }
};
```

Operator overloading example, continued

We can now use the '+' and '*' operators with our `complex` type, just as we can with "normal" types:

```
int main() {
  complex c1(2, 1);
  complex c2(3, 3);
  complex c3 = c1 * c2;
  complex c4 = c1 + 1; // Implicit promotion
}
```

The constant '1' is implicitly converted to a `float`, then our constructor generates a complex before performing the second addition.

Global operator functions

We can also define operator functions at the global level.

```
ostream & operator << (ostream &out, complex &r) {
  out << r.getReal() << " + " << r.getImag() << "i";
  return out;
}

int main() {
  complex c1(2, 1);
  complex c2(3, 3);
  complex c3 = c1 * c2;
  complex c4 = c1 + 1;

  cout << c3 << endl;
  cout << c4 << endl;
}
```

the output would be:

```
3 + 9i
3 + 1i
```

Overloading continued

Consider our example of matrix/vector multiplication:

```
vector operator*(const matrix& m, const vector& v)
{
  vector r;

  for (int i=0; i < m.rows(); i++) {
    r.elem(i) = 0;
    for (int j=0; j < m.cols(); j++) {
       r.elem(i) += m.elem(i,j) * v.elem(j);
    }
  }
  return r;
}
```

this permits us to write:

```
vector v, w;
matrix m;
//...
w = m * v;
```

Overloading conversion

Consider a simple string class:

```
class pstring {
  char *m_buf;
  int m_len;
public:
  pstring() { m_buf = 0; m_len = 0; } // Default Constructor
  pstring(char *pch); // Constructor
  pstring(const pstring &str); // Copy constructor
  int length() const { return m_len; }

  operator int() const { // Conversion to integer
    int result = 0;
    char *p = m_buf;
    for (int i = 0; (*p > '0' && *p < '9') && i < m_len; i++)
      result = (result * 10) + (*p++ - '0');
    return result;
  }
};
```

String conversion

```
#include <iostream>
#include "pstring.h"
using namespace std;
int main() {
  pstring y("12345");

  int n = (int)y; // Conversion
  cout << n << endl;
}
```

Overloading indexing

We can overload the idea of array indexing, and even check array bounds if desired:

```
// return reference so can use as lvalue
char & pstring::operator[](int i) {
  if (i < 0 || i >= m_len) {
    i = 0;
    cerr << "Index out of bounds\n";
  }
  return m_buf[i];
}

int main() {
  pstring y("12345");
  cout << y[4] << endl;

  cout<< y[10] <<endl; // Will print an error
}
```

Inheritance


McDuck

What is inheritance?

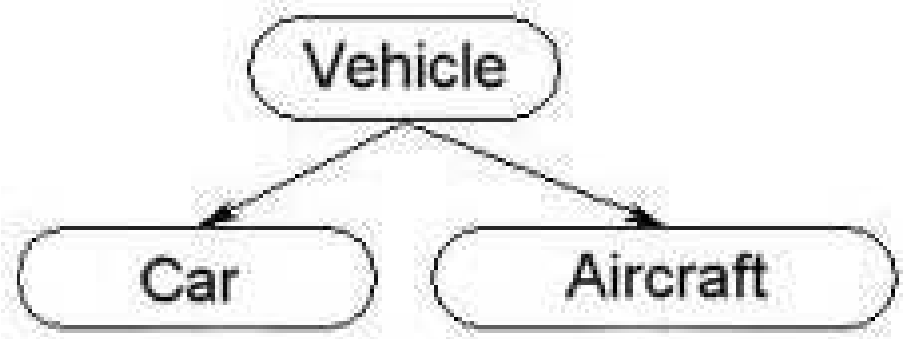
Problem: You write a class A and later discover a class B that is almost identical to A but has some extra attributes and operations.

Solution: Ask class B to re-use operations / attributes of A.

- *Inheritance* refers to our ability to create a hierarchy of classes, in which *derived* classes (subclass) automatically incorporate functionality from their *base* classes (superclass).
- A derived class inherits all of the data and functions from its base class.
- A derived class may add its own features, data, etc, and may also *override* one or more of the inherited functions.

Inheritance Example

- Inheritance allows to build software incrementally, allowing you to first build classes for the generic case and then build classes for special cases that inherit from the generic classes.



Inheritance syntax

```
class A {       // base class
private:
  int x;        // Visible only within 'A'
protected:
  int y;        // Visible to derived classes
public:
  int z;        // Visible globally
  A();          // Constructor
  ~A();         // Destructor
  void f();     // Example method
};

class B : public A { // B is derived from A
private:
  int w;        // Visible only within 'B'
public:
  B();          // Constructor
  ~B();         // Destructor
  void g() {
    w = z + y; // OK
    f();        // OK
    w = x + 1; // Error - 'x' is private to 'A'
   }
};
```

Notes

- No private members are inherited EVER with any of the keywords. Only public and protected members are inherited.
- `friends` are not inherited
- Types of Inheritance
  - Single Inheritance
  - Multiple Inheritance (Next Lecture)

```
class Foo {// ...};
class Bar {// ...};
class FooBar: public Foo, public Bar {// ...};
```

Benefits of Inheritance

- Maximize reuse and quality
  - Programmers reuse the base classes instead of writing new classes
  - Using well-tested base classes helps reduce bugs in applications that use them (e.g. Boost Library)
  - Reduce object code size

Overriding member functions

A derived class may *override* a function from its base class:

```
class A {
public:
  void f(int x) { cerr << "A::f(" << x << ")\n"; }
};

class B: public A {
public:
  void f(int x) { cerr << "B::f(" << x << ")\n"; }
};

int main() {
  A a;
  B b;
  a.f(1);
  b.f(2);
}
```

the `main()` program will print:

```
A::f(1)
B::f(2)
```

Do not confuse overriding with overloading!

Calling the base class

Overridden functions do not automatically invoke the base class implementation. We have to do this explicitly:

```
class B: public A {
public:
  void f(int x) {
    A::f(x);            // Call the base class
    cerr << "B::f(" << x << ")\n";
  }
};
```

the prior `main()` would now print:

```
A::f(1)
A::f(2)
B::f(2)
```

Because of *multiple inheritance*, C++ does not offer the Java `super()` construct.

Assignment compatibility

C++ considers objects of a derived class to be assignment compatible with objects of their base class. This just makes a copy, skipping members that aren't part of the base class.

```
class A {
protected:
  int x;
//...
};

class B: public A {
  int y;
//...
};

int main() {
  B b;
  A a;
  a = b;    // OK, but 'y' is not copied!
}
```

Assignment compatibility

However, we can't do the reverse and assign an object from a base class to a derived class. This could leave derived class members in an undefined state.

```
class A {
protected:
  int x;
//...
};

class B: public A {
  int y;
//...
};

int main() {
  B b;
  A a;
  b = a;    // Not OK - undefined value for 'y'
}
```

Assignment compatibility with pointers

The same rules apply with pointers. We can assign the address of an object of a derived class to an pointer to the base class, but not the opposite.

```
class A {
  // ...
};
class B: public A {
  // ...
};

int main() {
  A a, *pa;
  B b, *pb;
  pa = &b; // OK
  pb = &a; // Error!
}
```

However, since we are assigning pointers, the objects in these assignments *are not modified*, as opposed to the case when objects are copied. They retain their full contents.

Polymorphism

The ability to use base class pointers to refer to any of several derived objects is a key part of *polymorphism*.

Exploiting polymorphism requires additional effort:

```
class A {
public:
  void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f() { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();     // Which f() does this call?
}
```

This call invokes the base class, A::f()! This is known as *static binding*.

The choice of function to call is made at compile time.

Virtual functions - Dynamic Binding

The solution is to declare functions `virtual`.

```
class A {
public:
  virtual void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f() { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();     // Now this will call B::f()! Dynamic Binding
}
```

The virtual keyword indicates that the appropriate function is called based on the type of object that the pointer refers to, and not by the type of pointer. The choice of function to call is made at run time.

Virtual functions

A virtual function in the derived class will override the base class only if the type signatures match.

```
class A {
public:
  virtual void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f(int x) { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();     // Now this will call A::f()!
}
```

As with overloading, changing only the return type introduces an ambiguity and will trigger a compile-time error.

Virtual function details

- You do not need to use the `virtual` keyword in the derived classes, but it is legal.
- If you explicitly use the scope operator, you can override the natural choice of function.

```
class A {
public: virtual void f() { cerr << "A::f()\n"; }
};

class B : public A {
public: virtual void f() { cerr << "B::f()\n"; }
};

int main() {
  A *pa = new B();

  pa->A::f();    // Explicity invokes the base class
  pa->f();       // Invokes B::f()
}
```

Virtual constructors or destructors

- You *cannot* declare a constructor virtual.
- You can, and often *should*, declare a destructor virtual:

```
class A {
public:
  virtual ~A() {};
};

class B : public A {
private:
  int *mem;
public:
  B(int n=10) { mem = new int[n]; }
  ~B() { cerr << "~B()\n"; delete [] mem; }
};

int main() {
  A *pa = new B(100);

  delete pa; // What would happen if the destructor
             // was not declared as virtual?
}
```

Virtual destructor cont'd

- If the destructor was not declared virtual, the compiler would use the type of `pa` to decide which method to call.
- In our case, `pa` is of type A, so the A destructor would get called.
- This can cause a memory leak from `pa` (how?)
- *Always* declare destructors virtual!

Pure virtual functions and abstract classes

- Pure virtual function is a function declared with no definition (the base class contains no implementation at all)
- a class containing a pure virtual function is an abstract class (similar to Java interfaces)
- This enforces a design through inheritance hierarchy (inhertied classes must define implementation)

Example of Abstract Class

```
class A {
public:
  A();
  virtual void f() = 0; // pure virtual
};
class B: public A {
public:
  B();
  void f() { cout << "Class B" << endl; }
};
class C: public B {
public:
  C();
  void f() { cout << "Class C" << endl; }
};
```

Example continued

```
int main() {
B b; C c;
A* pa1 = &b;
A* pa2 = &c;
pa1->f();
pa2->f();
}

Outputs:
Class B
Class C
```

Abstract Class Example

Motivational Example: Imaging Device Objects (webcam, firewire, disk images, movies) need to acquire frames in their own way

```
class ImagingDevice {
protected:
    unsigned char *buffer;
    int width, height;
    ...

public:
    ImagingDevice();
    virtual ~ImagingDevice(); // virtual destructor
    ...
    virtual bool InitializeDevice() = 0;
    virtual bool GetImage()=0;
    virtual bool UninitializeDevice() = 0;
    virtual void SaveImage()=0;
    ...
};
```

Continued

```
class USBDevice: public ImagingDevice {
...

public:
   USBDevice();
   virtual ~USBDevice();
   ...
};

bool USBDevice::InitializeDevice(){ ... }
bool USBDevice::UninitializeDevice(){ ... }
bool USBDevice::GetImage(){ ... }
void USBDevice::SaveImage(){ ... }
```