Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 18

- RTTI
- Static Members
- Namespaces
- Operator Overloading

RTTI : Runtime Type Information

Type information attached to objects that lets the C++ runtime system

- dispatch virtual functions (VFTP)
- decide whether a down-cast is legal

Other languages such as Java and C# can do more at runtime: program reflection – look at data types — and even create new types.

One possible implementation of `dynamic_cast<T*>(p)`: at compile time create a directed graph that mirrors the type hierarchy. Then determine at runtime whether `*p` or an (indirect) super-class of `*p` is of type `T`.

Static Members

- Sometimes it is useful if all objects of a class have access to the same variable e.g. a "global" class option or a counter that keeps track of how many objects have been created
- Can also save space e.g. shared pointer to error-handling routine
- Advantages:
  - Information hiding can be enforced. Static members can be private - global variables cannot
  - Static members are not entered in global namespace, limiting accidental name conflicts
- Syntax: `static` qualifier before var/func decl.

Example

```
struct X
{
  X() { ++count; ... }

  // static member function
  static int get_count() { return count; }

private:
  static int count;     // #of X objects...
};

int X::count = 0; // ... must be defined in .C file

int main()
{
  X x;
  cout << X::get_count() << endl; // 1
  return 0;
}
```

Namespaces

Symbol collections (types, variables, functions) qualified by name. Avoids name conflicts

`using namespace X;`

- introduces all symbols of namespace `X` into the currenct context (no need for qualification)
- e.g. `using namespace std;` introduces `cin,cout,...`

`using X::y;`

- symbol `y` is introduced as being an abbrevation for `X::y`
- e.g. `using std::iostream;` only introduces `iostream`

Namespaces continued

You can create your own namespaces. E.g.

- `namespace foo { void bar(); }`
- call with: `foo::bar()` or
- `using namespace foo;  bar();`

No namespace: symbols are put in global namespace (empty prefix). E.g.

```
::strlen(s)   // defined in <cstring>
```


Example

```
#include <iostream>

namespace My
{
  int cout;
};

int main()
{
  cout = 0;             // illegal: undeclared
  My::cout = 0;         // OK

  std::cout = 0;        // illegal: stream!
  std::cout << "foo";  // OK
}
```


Operator Overloading

Goal: no difference between built-in types and class types

Would like to write:

```
class Matrix { ... } a, b, c, d;
class Vector { ... } v;

a = b + c * d;

cout << a; cin >> b;

v[0] = 0;
```

C++ allows users to overload/redefine global operators such as << and class operators such as `[ ]`

Limits: arity, associativity, and priority are fixed!

Complex Number Example

```
#include "Complex.H"

int main()
{
  Complex a(1.0);
  Complex b(0.0, 1.0);
  Complex c;

  c = (a + b) * (a - b);
  c += Complex(4, 3);
  c = c + 3.0;
  ++c;
  std::cout << c << std::endl;
};
```


Global Operators

Example: C++ I/O streams

How to define global operators such as input/output operators << >> ?

- `ostream &operator<< (ostream &os, const X &x);`
- `istream &operator>> (istream &is, X &x);`
- Reference to streams returned to allow cascading such as `cout << x << y;` and `cin >> x >> y;`

Example

```
class Complex { // Complex number class
  ...
private:
  float re, im; // real and imaginary parts
};

// write complex number to output stream
ostream &operator<< (ostream &os, const Complex &x) {
  os << x.re << ' ' << x.im;
  return os;
}

// read complex number from input stream
istream &operator>> (istream &is, Complex &x) {
  is >> x.re >> x.im;
  return is;
}
// doesn't work:  re,im are private!
```


Solution: friends or getters/setters

```
class Complex
{
public: ...
  friend ostream &operator<<(ostream &os, const Complex &x);
  friend istream &operator>>(istream &is, Complex &x);
private:
  float re, im;
};

ostream &operator<< (ostream &os, const Complex &x) {
  os << x.re << ' ' << x.im; return os;
  // Alternative: os << x.get_re() << ' ' << x.get_im(); return os;
}

istream &operator>> (istream &is, Complex &x) {
  is >> x.re >> x.im; return is;
}

// application

Complex a;
cin >> a; cout << a;
```

Friends

- Syntax (in class definition):

  ```
  friend <function-declaration> ;
  friend <class-name> ;
  ```

- Functions or entire classes now have access to all data/function members, even to those that are private!

- Avoid — usually indicates a class design that can be improved

Class Operators

No parameters:

```
+  -  *  !  &  ~  ++  --  (prefix/suffix)
```

One or more more parameters:

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

DON'T OVERLOAD: prefix-& && || ,

Int-Vector Revisited

```
class V {
public:
  V(int n_=1) { ... }
  ~V() { ... }

  int &operator[](int i) { check(i); return p[i]; }

  const int &operator[](int i) const {
    check(i); return p[i];
  }
  ...
private:
  void check(int i) const { assert(i >= 0 && i < n); }
  int *p, n; ...
};

V v(100);
v[3] = 0; cout << v[0];
```


Why Two Definitions of operator[]?

```
class Foo {
public:

  V a;
  ...

  int bar() const {
    return a[0];  // only works if const definition
                  // is provided for V[]
                  // otherwise, the compiler complains
                  // that bar() may change a
  }
};
```

Class Operator Syntax

Unary prefix operator ++x (or --x):

```
X& operator++() { ... }
```

Unary postfix operator x++ (or x--):

```
X operator++(int) { ... }
```

Binary infix operator x @ x : (@ = + - * ...)

```
Y operator@(const X &x) { ... }
```

```
[]: Y operator[](T i) {...}
```

```
(): Y operator()(<params>) {...}
```

```
->: Y* operator->() {...}
```

has to return pointer because e.g. a->foo accesses member foo

Calling Class Operators

Class operator applications are transformed into regular member function calls. E.g.

```
v[i]     ->  v.operator[](i)

f(x, y) ->  f.operator()(x, y)  // f is a functor

a + b    ->  a.operator+(b)

++x      ->  x.operator++()

x++      ->  x.operator++(0)
```

So, class operators are actually member functions

They can even be virtual!


```
class Complex { // Complex Number class
public:
  Complex(float r=0, float i=0) : re(r), im(i) {}
  // default destructor, assignment, copy OK

  Complex operator+(const Complex &x) const;
  Complex operator+(float x) const;     // special case
  ...
  Complex &operator+=(const Complex &x);
  Complex &operator+=(float x);         // special case
  ...
  Complex &operator++();                // prefix ++
  Complex  operator++(int);             // postfix ++
  Complex  operator-() const;           // unary operator
  ...
  float real() const { return re; }     // gives environment
  float imag() const { return im; }     // access to data

private:
  float re, im;  // real & imaginary part
};
```


Complex Class Implementation

```
#include "Complex.H"

Complex Complex::operator+(const Complex &x) const {
  // computes new coordinates, copy-constructs a new
  // object and returns it to the environment
  return Complex(re + x.real(), im + x.imag());
}

// faster implementation in special case
Complex Complex::operator+(float x) const {
  return Complex(re + x, im);
}

Complex &Complex::operator+=(const Complex &x) {
  re += x.real();
  im += x.imag();
  return *this;
}

Complex Complex::operator-() const {
  return Complex(-re, -im);
}
```