Operator Overloading

- Operators and Functions
- friends
- unary, binary
- +, *, ++, etc.
- <<, >>
- =, []

Introduction

- Just as functions can be overloaded using different *signatures*, operators can be overloaded using different *operands*

- Operator overloading is not essential – we could do the job just as well with functions (as we'll see).

- Operator overloading can be dangerous – C++ gives us *lots* of rope with which to hang ourselves. (This is why there is no operator overloading in Java.)

- If we are careful, operator overloading is useful and elegant – but we shouldn't go overboard.

Operators and Functions

The key to operator overloading is:

operators and functions are really the same thing

E.g. consider:

```
c = a + b
```

"$+$" is the *operator*, with (a,b) the *operands*.

If the compiler didn't provide the "$+$" operator, we could live just as happily with a `plus` function:

```
c = plus(a,b)
```

(altho we would have to do tedious bit manipulations in `plus`)

Let's code this in the "obvious" way:

```
#include <iostream.h>   // operator1.cc

int plus(int a, int b) { return a+b; }

int main() {
  const int a(17);
  const int b(28);
  cout << "plus(" << a << "," << b << ") = " << plus(a,b) << endl;
  return 0;
}
```

1. OK, we're still using "+" but patience!

2. We've only done this for `int`, but we know how to use templates

3. For objects bigger than `int`, we should really use *references* – so as not to pass big objects on the stack.

```
#include <iostream.h>    // operator2.cc

int plus(const int& a, const int& b) { return a+b; }

int main() {
  const int a(17);
  const int b(28);
  cout << "plus(" << a << "," << b << ") = " << plus(a,b) << endl;
  return 0;
}
```

- This `plus` function will now work with other objects. E.g. suppose we want to add (in the vector sense) two `Point` objects. We probably *don't* want to use templates here, because different objects will likely have different algebras.

- First add a `plus` declaration to the header file (we've removed the virtual functions for simplicity). If we try to do the "obvious" we'll hit a problem:

```
class Point {
public:
  ...
  Point plus(const Point& a, const Point& b);
};
```

- We think we want `plus` to be a member function of class `Point`. But if this were the case, we would have to use it as part of an object:

```
Point c;
c.plus(a,b);
```

- We don't want to do this, because we want it to look like "normal" algebra:

```
Point c=plus(a,b);    // Point c=a+b
```

- `plus` is really a global function – it exists without an object.
- `Point`'s data are `private`, so a non-member function can't get at them.
- We get round this by introducing a new C++ construct, the `friend`.

Friends

A friend is a way of letting a different object or function have access to a class's private data.

As with all friends, you should pick them carefully, and not be too friendly to too many people.

Only use friends when there is no other "clean" way

- An object has to grant friendship – a function cannot just decide it wants to be a friend. That would break encapsulation.
- An object only grants friendship to *specific* classes or functions – not just anyone who comes along.

- In the `plus` example, class `Point` needs to grant access to function `plus`
- It's simply done with the `friend` keyword.
- Now `plus` is a friend of class `Point`, `plus` can use `Point`'s private data.
- We are then able (finally) to implement our `plus` function:

```
#ifndef __POINT_HH   // Point3.hh
#define __POINT_HH
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rMoveTo(const Point&);
  void moveTo(const Point&);
  friend Point plus(const Point&, const Point&);
private:
  int m_x, m_y;
};
#endif // __POINT_HH
```

```
#include "Point3.hh"   // operator3.cc

Point plus(const Point& a, const Point& b) {
  Point tmp;
  tmp.m_x = a.m_x + b.m_x;
  tmp.m_y = a.m_y + b.m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  const Point b(5,-2);
  Point c=plus(a,b);
  cout << "plus(a,b) = ";
  c.print();
  cout << endl;
  return 0;
}
```

Points to note:

- The *return* type of `plus` is a `Point`, but `plus` is *not* a member of `class Point`
- `plus` returns an object, and not a reference? Why?
  1. To return a reference, the object must already exist
  2. A temporary object exists in `plus`, but that goes out of scope when `plus` returns.
- But — we set out to overload "+", not write a function "`plus`".
- All we have to do is define (with the right syntax) our operator:

```
#ifndef __POINT_HH   // Point4.hh
#define __POINT_HH
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rMoveTo(const Point&);
  void moveTo(const Point&);
  friend Point operator+(const Point&, const Point&);
private:
  int m_x, m_y;
};
#endif // __POINT_HH
```

```
#include "Point4.hh"   // operator4.cc

Point operator+(const Point& a, const Point& b) {
  Point tmp;
  tmp.m_x = a.m_x + b.m_x;
  tmp.m_y = a.m_y + b.m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  const Point b(5,-2);
  Point c = a + b;
  cout << "a+b = ";
  c.print();
  cout << endl;
  return 0;
}
```

- Now we can use "`+`" with `Point` objects.
- It looks like the operator is really just a function, called `operator+`. Is it? — Yes!
- We could equally well do:

  ```
  Point c = operator+(a,b);
  ```

  Of course, having gone to the trouble of defining "+" we wouldn't – but it shows the equivalence of operators and functions.
- Even tho `plus` is not a *member* of `Point`, it's probably sensible to keep the implementation code in `Point.cc`
- We are using the `=` operator for class `Point` – see later.

Member or friend?

Now we're starting to feel secure, let's do it a different way. C++ gives us yet more rope with which to hang ourselves.

This can get *really* confusing. Fasten your seat belts.

In the function signature, we know that the signature is specified by:

1. the argument types
2. the function name
3. the class. i.e. `A::f()` is different from `B::f()`.

- Usually, this match is done with code such as:

```
A a;
a.f();
```

- To maintain (a kind of) symmetry, the match can also be done with the *first operand* of an overloaded operator.

- So we have another way of defining an overloaded operator – this time as a true member function.

```
class Point {
public:
  ...
  Point operator+(const Point&) const;
};
```

- But wait a minute – what's that final `const`?
- We want our operator to take `const` operands, but usually a member function can modify the current (`this`) object. We not only don't want that – we will also get a compilation error.

> **Rule** — **A member function cannot change the `this` object if it is declared `const`.**

- So now – finally – we can implement our member function overloaded operator:

```
#include "Point5.hh"   // operator5.cc

Point Point::operator+(const Point& a) const {
  Point tmp;
  tmp.m_x = m_x + a.m_x;
  tmp.m_y = m_y + a.m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  const Point b(5,-2);
  Point c = a + b;
  cout << "a+b = ";
  c.print();
  cout << endl;
  return 0;
}
```

- So which should we use? Friend or member?
- Other things being equal, it's probably best to stick with member functions, rather than friends.
- But that's not always possible:

Suppose we overload the "*" operator, to scale a Point. We could do it just as for +

```
class Point {
public:
  ...
  Point operator*(const int&) const;
};
```

and implement in the obvious way:

```
#include "Point6.hh"   // operator6.cc

Point Point::operator*(const int& a) const {
  Point tmp;
  tmp.m_x = a*m_x;
  tmp.m_y = a*m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  const int scale(3);
  Point b = a*scale;
  cout << "a*" << scale << " = ";
  b.print();
  cout << endl;
  return 0;
}
```

- But there's a problem: we'd like scalar multiplication to commute: `Point b = a*scale;` and `Point c = scale*a;` should do the same thing.
- But they don't and they can't: for a member function (operator), the first operand is the object itself, so `Point b = a*scale;` works.
- However, for `Point c = scale*a;` we'd have to write a member function for class `int` – and there is no such class. It's a built-in type.
- It would be ugly to have one operator a member, and the other global.
- In this case, we should make them both global, and define them both accordingly.

```
class Point {
public:
  ...
  friend Point operator*(const Point&, const int&);
  friend Point operator*(const int&, const Point&);
};
```

---

once we've defined `Point*int` we can define `int*Point` in terms of `Point*int`, rather than all over again.

```
#include "Point7.hh"   // operator7.cc

Point operator*(const Point& a, const int& s) {
  Point tmp;
  tmp.m_x = s*a.m_x;  tmp.m_y = s*a.m_y;
  return tmp;
}
Point operator*(const int& s, const Point& a) { return a*s; }

int main() {
  const Point a(3,4);
  Point b(a*6);
  cout << "a*6 = "; b.print(); cout << endl;
  b = 3*a;
  cout << "3*a = "; b.print(); cout << endl;
  return 0;
}
```

Unary and Binary operators

- The "+" operator is called a *binary* operator – because it has 2 operands.
- Operators can also be *unary* – with just 1 operand.
- Let's write a "-" operator to change the sign of a `Point` object. Again, we have the choice of friend or member. Let's do it first with a friend:

```
class Point {
public:
  ...
  friend Point operator-(const Point&);
};
```

```
#include "Point8.hh"   // operator8.cc

Point operator-(const Point& a) {
  Point tmp;
  tmp.m_x = -a.m_x;
  tmp.m_y = -a.m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  Point b(-a);
  cout << "-a = ";
  b.print();
  cout << endl;
  return 0;
}
```

That was easy!

Now we can do it as a member: since the first operand is the object itself – and there is only one operand, we don't give `operator-()` any arguments.

```
class Point {
public:
  ...
  Point operator-() const;
};
```

```
#include "Point9.hh"   // operator9.cc

Point Point::operator-() const {
  Point tmp;
  tmp.m_x = -m_x;
  tmp.m_y = -m_y;
  return tmp;
}

int main() {
  const Point a(3,4);
  Point b(-a);
  cout << "-a = ";
  b.print();
  cout << endl;
  return 0;
}
```

- `operator-()` is a *prefix* operator – the operator comes before the operand.

- Most unary operators are prefix, but `++` and `--` can be *either* prefix *or* postfix. How will that work?

> **Rule**
>
> **The normal operator rules apply to *prefix* operators.**

- If we *want* a prefix operator, we don't have to do anything differently. Let's do the `++` operator.

- If `p` is a `Point`, let `++p` increment `p`'s (x, y) coordinates.

- We know how to do this – let's use a friend.

```
#include "Point11.hh"   // operator11.cc

Point operator++(Point& a) {
  ++a.m_x;
  ++a.m_y;
  return a;
}

int main() {
  Point a(3,4);
  Point b = ++a;
  cout << "++a = ";
  b.print();
  cout << endl;
  return 0;
}
```

- How do we do the postfix ++ operator?
- There is no obvious way: C++ gives us a rule and a trick:

Rule	**If a default operator is unary (binary) the overloaded operator must also be unary (binary).**

- Since ++ starts out *unary*, it must *always* be unary.
- The trick is if we provide a *second* (dummy) operand, C++ knows that there can't be 2 operands, so interprets this as a *postfix* operator – ignoring the second operand.
- But it can't be any old operand:

Rule	**The second (dummy) operand of a unary operator must be an `int`**

Now we can do the postfix ++ operator:

```
class Point {
public:
  ...
  friend Point operator++(Point&);       // prefix
  friend Point operator++(Point&, int);  // postfix
};
```

```
#include "Point12.hh"   // operator12.cc

Point operator++(Point& a) {
  ++a.m_x;  ++a.m_y;
  return a;
}
Point operator++(Point& a, int) {
  Point tmp=a;
  ++a;
  return tmp;
}
int main() {
  Point a(3,4);
  cout << "a++ = ";
  a++.print();
  a.print();  cout << endl;
  return 0;
}
```

Note:

1. we define the postfix in terms of the prefix
2. for the postfix, we have to make a local copy to return before we increment

For this second reason, postfix operators are *always* more expensive – this can be important.

We will return to this when we cover STL.

Can we do the same using a member function?

Yes – by using the `this` pointer:

```
class Point {
public:
  ...
  Point operator++();        // prefix
  Point operator++(int);     // postfix
};
```

We'll be using `this` a lot.

```
#include "Point13.hh"   // operator13.cc

Point Point::operator++() {
  ++m_x;  ++m_y;
  return *this;
}
Point Point::operator++(int) {
  Point tmp=*this;
  ++*this;
  return tmp;
}
int main() {
  Point a(3,4);
  cout << "a++ = ";
  a++.print();
  a.print();  cout << endl;
  return 0;
}
```

Overloading iostream operators

We now overload the iostream extraction/insertion operators, `>>` and `<<`.

Let's remind ourselves how they are used:

```
cout << foo;
```

- The operand to the *right* of `<<` is inserted into the stream
- The return type of `<<` is the stream itself – which has been modified
- The returned stream is not a *new* stream, but the stream on the LHS of the operator.
- There's an extra subtlety – but we'll hold off for a while

To declare `<<`, we add to `Point.hh`

```
class Point {
public:
  ...
  friend ostream& operator<<(ostream& os, const Point& p);
};
```

- The *first* argument of `<<` is the `ostream` (or `istream`) instance. `<<` and `>>` can *never* be member functions.
- `iostream.h` defines `<<` and `>>` as member functions for all the *built-in* types.

The implementation appears straightforward:

```
#include "Point14.hh"   // operator14.cc

ostream& operator<<(ostream& os, const Point& p) {
  os << "(" << p.m_x << ", " << p.m_y << ")";
  return os;
}

int main() {
  Point a(3,4);
  cout << "a = " << a << endl;
  return 0;
}
```

But there are some traps:

- Why do we declare `<<` type `ostream&`?
- Why do we return `os`?

Suppose we write: `cout << foo << bar;`
This is really shorthand for: `(cout << foo) << bar;`
The sequence is:

1. First do: `cout << foo;` and return the *same* stream `cout`
2. Then do: `cout << bar;`

- For this to work, not only must `<<` return an `ostream` object, but it must return a *Reference* to the first argument.
- Without a reference, we would have to create a *new* ostream object.

Now it is easy to “chain” `<<` operations:

```
#include "Point15.hh"   // operator15.cc

ostream& operator<<(ostream& os, const Point& p) {
  os << "(" << p.m_x << ", " << p.m_y << ")";
  return os;
}

int main() {
  Point a(3,4);
  cout << "a = " << a << "\n" << Point(5,12) << endl;
  return 0;
}
```

Overloading assignment operator

- C++ gives us certain default functions: “bare” constructor, copy constructor, destructor
- it also gives us a default assignment operator, `=` which does a *member by member* assignment.

```
#include "Point16.hh"   // operator16.cc

int main() {
  Point a(3,4);
  Point b=a;
  cout << "a = " << a << "\nb = " << b << endl;
  return 0;
}
```

- For class `Point` there is no problem.
- What if the class contained a *pointer* to data outside the class?
- Let's define a class `Array` with a dynamically-created array. (Exercise for the student: make `Array` a template class.)
- But we already know the trap: we've seen it in the copy constructor example:
  - the default `=` only copies the *pointer*, not the contents
  - we already have to write a destructor and copy constructor
  - we also have to provide an assignment operator

```
#ifndef __ARRAY_HH   // Array1.hh
#define __ARRAY_HH
#include <iostream.h>

class Array {
public:
  Array(unsigned int size=0, const int* array=0);
  Array(const Array&);      // copy constructor
  ~Array();                 // destructor
  Array& operator=(const Array&);
  friend ostream& operator<<(ostream& os, const Array&);
private:
  unsigned int m_size;
  int* m_array;
};

#endif // __ARRAY_HH
```

`Array::operator=()` needs some work and explanation:

```
Array& Array::operator=(const Array& a) {
  if (this!=&a) {
    delete [] m_array;
    m_array = new int[m_size=a.m_size];
    for (unsigned int i=0; i<m_size; i++) {
      m_array[i]=a.m_array[i];
    }
  }
  return *this;
}
```

```
#include <stdlib.h>   // operator17.cc
#include "Array1.hh"

int main() {
  int x[] = { rand(), rand(), rand(), rand() };
  int y[] = { rand(), rand(), rand(), rand(), rand(), rand() };
  Array a(sizeof(x)/sizeof(int), x);
  Array b(sizeof(y)/sizeof(int), y);
  Array c(a);
  cout <<"&a, a: "<<hex<<(unsigned long)&a<<",  "<<dec<< a<<endl;
  cout <<"&b, b: "<<hex<<(unsigned long)&b<<",  "<<dec<< b<<endl;
  cout <<"&c, c: "<<hex<<(unsigned long)&c<<",  "<<dec<< c<<endl;
  c = b;
  cout <<"&c, c: "<<hex<<(unsigned long)&c<<",  "<<dec<< c<<endl;
  return 0;
}
```

Points to note:

- the return type for `Foo::operator=()` is a `Foo&`
- the return object is `*this`
- the operand for `Foo::operator=()` is a `const Foo&`
- `operator=()` first tests that the argument is not `this`
- `operator=()` then deletes the old array before allocating a new one. This is *not* the constructor, so there is *always* a pre-existing array.
- it then does a copy of all the elements of the array

Why return `Array&`, rather than make `Array::operator=()` void?

assignments can be chained:

```
c = b = a;
```

This does the following:

1. first assign `a` to `b`
2. then return the result `b`
3. then assign this result to `c`

to do this, `operator=()` must return a reference to its argument object.

Be careful whenever you use `new`

Whenever memory is allocated outside the object, you should *always*

- Provide a copy constructor
- Provide a destructor
- Provide an assignment operator

Be careful to define the whole algebra

Suppose we:

- Define an overloaded `operator+()`
- Define an overloaded `operator=()`

What happens if we write:

```
Foo a;
Foo b;
b += a;
```

Fortunately, the compiler saves us! Altho there is a default `operator=()`, there is *not* a default `operator+=()`.

The implementation looks simpler than `operator+()`

```
Array& Array::operator+=(const Array& a) {
  for (unsigned int i=0; i<m_size; i++) {
    m_array[i] += a.m_array[i];
  }
  return *this;
}
```

So maybe we should define `operator+()` in terms of `operator+=()`:

```
Array Array::operator+(const Array& a) const {
  Array tmp=*this;
  tmp += a;
  return tmp;
}
```

Overloading [] operator

- The subscript operator, `[]` can also be overloaded.
- Why should we do this?
- Let's make `Array` test that an index is within range when it is accessed.
- We can do this by overloading `[]`.

```
class Array {
public:
  ...
  int& operator[](int);
};
```

The implementation of `Array::operator[]()` is straightforward:

```
int& Array::operator[](int i) {
  if ( (i<0) || (i>=(int)m_size) ) {
    cout <<"index "<< i<<" out of bounds.  Should be 0<=i<"
 << m_size << endl;
    return m_array[0];   // no elegance here
  }
  return m_array[i];
}
```

as too is the use:

```
#include <stdlib.h>   // operator18.cc
#include "Array2.hh"

int main() {
  int x[] = { rand(), rand(), rand(), rand(), rand(), rand() };
  Array a(sizeof(x)/sizeof(int), x);
  cout <<"a: " << a <<endl;
  cout << "a[-3] = " << a[-3] << endl;
  cout << "a[4] = " << a[4] << endl;
  cout << "a[17] = " << a[17] << endl;
  return 0;
}
```

Points to note:

- `operator[]()` looks strange – the argument is between the `[` and `]`
- The return type is an `int&`. This allows the function to be on the LHS of an expression.

```
#include "Array2.hh"   // operator19.cc

int main() {
  const int kArraySize(6);
  Array a(kArraySize);
  for (int i=0; i<kArraySize; a[i++]=i*i) {}
  cout <<"a: " << a <<endl;
  return 0;
}
```

Whoa! The function (operator) is on the LHS of an expression?
This can be done with a Reference.

Some terminology:

- Usually, a function, such as $f(x)$, returns a result on the RHS of an expression.
  - This is called an `rvalue`
- If the result is on the LHS, as with `operator[]()`,
  - This is called an `lvalue`
  - The return type *must* be a reference
- C++ prevents `rvalues` and `lvalues` getting mixed up.

Rules for Operator Overloading

The following operators *can* be overloaded:

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

The following operators *cannot* be overloaded:

.	.*	::	?:	sizeof

1. Only built-in C++ operators can be overloaded
2. Operators for built-in types cannot be overloaded
3. ++ and -- come in both prefix and postfix versions. Use a dummy `int` argument to signify postfix.
4. Operator precedence rules cannot be changed by overloading
5. Default parameters cannot be used
6. A unary (binary) operator must also be unary (binary) when overloaded.
7. The number of operands cannot be changed

Style Guidelines

The default constructors, destructor, and `operator=` are often a source of error.

1. For class `Foo`, make the declarations: `Foo::Foo(const Foo&)`, `Foo::~Foo()`, and `Foo::operator=(const Foo&)`, `private:`. This will cause a compile error if they are used.

2. *Or:* make them `public:` and define them as:

   ```
   Foo::Foo(const Foo&) { assert(0); }
   ```

3. *Or:* define them correctly!

4. For a base class, make the destructor virtual. (The constructor cannot be virtual). This forces the destructor for the derived class to be called.

```
#ifndef __FOOBASE_HH   // FooBase.hh
#define __FOOBASE_HH
#include <iostream.h>

class FooBase {
public:
  FooBase(int x=0) : m_x(x) {}
  friend ostream& operator<<(ostream&, const FooBase&);
  virtual ~FooBase() { cout << "FooBase destructor" << endl; }
private:
  int m_x;
};

ostream& operator<<(ostream& os, const FooBase& f) {
  os << f.m_x;   return os;
}
#endif // __FOOBASE_HH
```

We create a pointer of type `FooBase*` for an object of type `Foo*`. (This is not as silly as it seems).

```
#include "FooBase.hh"   // FooBase.cc

class Foo : public FooBase {
public:
  Foo(unsigned int i) : FooBase(i) {}
  ~Foo() { cout << "Foo destructor" << endl; }
};

int main() {
  FooBase* f = new Foo(5);
  cout << *f << endl;
  delete f;
  return 0;
}
```