COMP6771
Advanced C++ Programming

Week 4
Part Two: Operator Overloading

2017

www.cse.unsw.edu.au/~cs6771

Move Semantics Revisited

- Consider the code below:

```
1  UB_stack(const UB_stack &s);
2  UB_stack(const UB_stack &&s);
3  UB_stack& operator=(const UB_stack &s);
4  UB_stack& operator=(const UB_stack &&s);
```

- Which ones belong to move semantics? Which ones are copy semantics?
- Is there any problem with the move semantics code?

Move Semantics Revisited

- Do not make the object that you are moving from const!

```
1 UB_stack(const UB_stack &s);          // copy constructor
2 UB_stack(UB_stack &&s);          // move constructor
3 UB_stack& operator=(const UB_stack &s);          // copy assignment
4 UB_stack& operator=(UB_stack &&s);                // move assignm
```

- What about mixing auto and uniform initialisation?

```
1 UB_stack a;
2 UB_stack b = a;  // copy
3 auto c = a;
4 auto d {a};
```

- What is the type of c? What is the type of d? Is it the same as b?

Copy Control Gotchas

- What is the type of c? What is the type of d?

```
1 UB_stack a;
2 UB_stack b = a;  // copy
3 auto c = a;
4 auto d {a}; // (or auto d = {a})
```

- c is an UB_Stack object as expected.
- d is an initializer list, it is not a UB_Stack object.
- See:
  http://scottmeyers.blogspot.com.au/2014/03/if-braced-i

What is Operator Overloading?

- Convenient shorthand
- What's more intuitive:

```
1  print(std::cout, add(d1, d2));
2  vs.
3  std::cout << d1 + d2;
```

- C++ supports a rich set of operators
- Programmer may define their own
- Powerful and subtle feature of C++
- Advantages:
  - Easier to remember
  - Reuse of existing code
  - Concise description of the task
- Disadvantage:
  - Lack of context may disguise meaning

Rules and Exceptions

Overloaded operators must have at least one operand of class or enumeration type.

Exceptions:

- the order of evaluation guarantees of `||`, `&&` and `,` are not preserved when these operators are overloaded

- Can't change precedence, associativity, or arity
- Can't invent new symbols

Some Advise

",", "&", "&&" and "||" should usually not be overloaded

Overloaded Operator Design

Some operators must be overloaded as class members: assignment (=), subscript ([]), call (()) and member access arrow (->).

Some operators must be defined as friend functions: I/O operators (<< and >>).

Otherwise we have a choice, but here are some guidelines:

- operators that change the state of the class or are closely linked to the class should also be defined as members (e.g. compound-assignment operators)
- symmetric operators, such as arithmetic, equality, relational and bitwise operators are best defined as friend functions

Careful

Do not be tempted to overload operators in ways that change their normal meaning and behaviour.

I/O Operators

- The output operator *must* be defined as a friend function and is usually a nonmember function.
- The output operator takes as arguments a reference to an output stream and a `const` reference to the class type and returns a reference to the supplied output stream:
- The output operator << for `Sales_data`:

```
1  Sales_data.h:
2    friend std::ostream& operator<<(std::ostream&,
3                                    const Sales_data &);
4  Sales_data.cpp:
5  std::ostream& operator<<(std::ostream &os, const Sales_data &item) {
6    os << item.isbn() << " " << item.units_sold << " "
7       << item.revenue <<  " " << item.avg_price();
8   return os;
9  }
```

I/O Operators

- The input operator >> is similar, it takes as arguments a reference to an input stream and a *non*-`const` reference to the class type and returns a reference to the supplied >>.
- The input operator for `Sales_data`:

```
1  Sales_data.h:
2  friend std::istream& operator>>(std::istream&,
3                                  Sales_data&);
4  Sales_data.cpp:
5  std::istream& operator>>(std::istream &is, Sales_data &item) {
6    double price = 0;
7    is >> item.bookNo >> item.units_sold >> price;
8    if (is) // check that the inputs succeeded
9      item.revenue = price * item.units_sold;
10   else
11     item = Sales_data();
12           // // failed: give it a default state
13   return is;
14 }
```

Compound Assignment Operators

- Usually implemented as a member function
- Here is how `combine` in `Sales_data` should have been implemented (more nicely with += overloaded):

```
1 Sales_data.h:
2 Sales_data& operator+=(const Sales_data &rhs);
3
4 Sales_data.cpp:
5 Sales_data& Sales_data::operator+=(const Sales_data &rhs) {
6   units_sold += rhs.units_sold;
7   revenue += rhs.revenue;
8   return *this;
9 }
```

Arithmetic Operators

- Arithmetic operators are usually nonmember functions
- Here is how add in `Sales_data` should be implemented

```
1  Sales_data.h:
2  Sales_data operator+(const Sales_data&, const Sales_data&);
3  // inside Sales_data class definition:
4  friend Sales_data operator+(const Sales_data&, const Sales_data&);
5
6  Sales_data.cpp:
7  Sales_data operator+(const Sales_data &lhs, const Sales_data &rhs) {
8    Sales_data sum = lhs;
9    sum += rhs;
10   return sum;
11 }
```

NB

Note that the + operator returns a new value and not a reference. This is consistent with the built-in operator. Further if both are present, the arithmetic operator should be defined in terms of the related compound assignment operator.

Relational Operators

- Relational operators are usually nonmember functions
- Classes with == and != often (but not always) have relational operators such as < and <=
- The associative containers and some STL algorithms use <
- Requirement for associative containers: if two objects are !=, then one should be < the other
- For `Sales_data`, < is not defined. Why? There is no single logical definition for <.

Equality Operators

- Equality operators are usually nonmember functions

```
1  Sales_data.h:
2  friend bool operator== (const Sales_data&, const Sales_data&);
3  friend bool operator!=(const Sales_data&, const Sales_data&);
4
5  Sales_data.cpp:
6  bool operator==(const Sales_data &lhs, const Sales_data &rhs) {
7    return lhs.isbn() == rhs.isbn() &&
8           lhs.units_sold ==  rhs.units_sold &&
9           lhs.revenue ==  rhs.revenue;
10 }
11
12 bool operator!=(const Sales_data &lhs, const Sales_data &rhs) {
13   return !(lhs ==  rhs);
14 }
```

SmallInt Class

```
1  // SmallInt.h:
2
3  #include<iostream>
4
5  class SmallInt { // range checking omitted
6  public:
7    SmallInt(int v) : value_{v} { }
8  private:
9    int value_;
10 };
```

- Designed to represent values in a subrange of int
- Can be used as follows:

```
SmallInt s{1};
SmallInt t{2};
```

Using the STL Sort and Copy Algorithms

```
1  // smallIntTest.cpp
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  #include <algorithm>
6  #include <iterator>
7  #include "SmallInt.h"
8
9  int main() {
10   std::vector<SmallInt> vec{SmallInt{3}, SmallInt{1}, SmallInt{2}};
11
12   std::sort(vec.begin(), vec.end());
13   std::copy(vec.begin(), vec.end(),
14        std::ostream_iterator<SmallInt>(std::cout, " "));
15 }
```

SmallInt: A Correct Implementation

```
1  // SmallInt.h:
2  #include<iostream>
3
4  class SmallInt {
5    friend std::ostream& operator<<(std::ostream &os,
6           const SmallInt &s);
7    friend bool operator<(const SmallInt&,const SmallInt&);
8  public:
9    SmallInt(int v) : value_{v} { }
10 private:
11   int value_;
12 };
13
14 std::ostream& operator<<(std::ostream &os, const SmallInt &s);
15 bool operator<(const SmallInt&,const SmallInt&);
```

SmallInt: A Correct Implementation

```
1  // SmallInt.cpp:
2  #include "SmallInt.h"
3
4  bool operator<(const SmallInt &rhs,const SmallInt &lhs) {
5      return rhs.value_ <= lhs.value_;
6  }
7
8  std::ostream& operator<<(std::ostream &os, const SmallInt &s) {
9    os << s.value_;
10   return os;
11 }
```

- sort requires operator<
- ostream_iterator requires operator<<

The Assignment Operator =

- We covered assignment of objects of the same type already. Recall that assignment must be defined as a member.
- We can define additional assignment operators to handle assignment from different types:

```
1  Sales_data.h:
2  Sales_data &Sales_data::operator=(std::istream &is);
3  Sales_data.cpp:
4  Sales_data &Sales_data::operator=(std::istream &is) {
5    double price = 0;
6    is >> bookNo >> units_sold >> price;
7    if (is)
8     revenue = price * units_sold;
9    else
10    *this = Sales_data();
11   return *this;
12 }
```

NB

For consistency with built-in operators, assignment and compound assignment should return a reference to the left-hand operand.

The Subscript Operator []

- Must be defined as a member function
- Often defined on an indexable container (vector or map)
- Explicit argument is the index
- Result is the value stored at the given position
- Return a reference to the value to get an lvalue
- operator[][] does not exist

The Subscript Operator []: An Example

- Define an array that does bound-checks at run time
- The subscript operator [] is overloaded

```
1  SafeArray.h:
2
3  class SafeArray {
4  public:
5    SafeArray(int s);
6    SafeArray(const int v[], int s);
7    ~SafeArray() { delete[] values; }
8    int& operator[](int i);        // for setting via []
9    int operator[](int i) const;   // for getting via []
10 private:
11   int size;
12   int* values;
13 };
```

- `delete[]` called on an array (covered in Week 5)

The Subscript Operator []: An Example

SafeArray.cpp:

```
#include <cassert>
#include "SafeArray.h"

SafeArray::SafeArray(int s) : size{s}, values{new int[size]} {}
SafeArray::SafeArray(const int v[], int s) : size{s}
{
  values = new int[size];
  for (int i = 0; i < size; i++)
    values[i] = v[i];
}

int& SafeArray::operator[](int index) {
  assert((index >= 0) && (index < size));
  return values[index];
}

int SafeArray::operator[](int index) const {
  assert((index >= 0) && (index < size));
  return values[index];
}
```

User Code

```
1  safeArrayTest.cpp
2  #include<iostream>
3  #include<cassert>
4  #include "SafeArray.h"
5
6  int main() {
7    SafeArray s{10};
8    s[12] = 2;
9    std::cout << s[2] << std::endl;
10 }
11
12 % g++ -std=c++11 -Wall -Werror -O2 -o safeArrayTest
13 SafeArray.cpp safeArrayTest.cpp
14
15 % ./safeArrayTest
16 safeArrayTest: SafeArray.cpp:13: int&
17 SafeArray::operator[](int): Assertion `(index >= 0)
18 && (index < size)' failed.
19 Aborted (core dumped)
```

Increment and Decrement Operators

- Generally defined as member functions
- Two versions of each:
  - prefix: `++x` and `--x` (lvalues)
  - postfix: `x++` and `x--` (rvalues)
- C++ uses an extra argument to distinguish the postfix form

Return Types

- Prefix returns a reference
- Postfix returns a value (it does more work)

- Prefix preferred over postfix for iterators:

```
1 vector<int> v;
2 for (auto it = v.begin(); it != v.end(); ++it)
3   ...
```

Increment and Decrement Operators: An Example

```
1  Clock.h:
2
3  #include<iostream>
4
5  class Clock {
6  friend std::ostream& operator<<(std::ostream&, const Clock&);
7  public:
8    Clock(int h=12, int m=0) : hour{h}, min{m} { }
9                                  // noon by default
10   Clock & operator++();          // prefix
11   Clock operator++(int);         // postfix
12 private:
13   void tick();                   // advance clock by 1 min
14   int hour, min;
15 };
```

Increment and Decrement Operators: An Example

```
1  Clock.cpp:
2  #include "Clock.h"
3
4  Clock & Clock::operator++() {   // prefix
5    tick();             // increment
6    return *this;       // return itself for assignment
7  }
8
9  Clock Clock::operator++(int) { // postfix
10   Clock c = *this;   // Save/copy the object.
11   tick();            // Increment the object.
12   return c;          // Return the original object
13 }
14
15 void Clock::tick() {  // Advance the object's min by 1
16   min = (min + 1) % 60;
17   if (min == 0)
18     hour = (hour + 1) % 24;
19 }
20
21 std::ostream& operator<<(std::ostream& os, const Clock& c) {
22   if (c.hour < 10)
23      os << '0';
24   os << c.hour << ":";
25   if (c.min < 10)
26     os << '0';
27   os << c.min;
28   return os;
29 }
```

The Arrow and Dereferencing Operators

- `->` and `*` are overloaded to provide pointer-like behavior
- Smart pointers:
  - Classes that exhibit pointer-like behavior when `->` is overloaded
  - Examples: `unique_ptr`, `shared_ptr` and `weak_ptr` in the C++ library and `HasPtr` in the textbook (§13.2.2)
- `->` must be a member function and `*` is usually a member
- The semantics of ptr->ident:
  1. If `ptr` is an (ordinary) pointer to an object with a member named `ident`, then the compiler will access the `ident` member
  2. else, if `ptr` is an object that defines `->`, then the compiler will access `ptr.operator->()->ident`
  3. else, there is an error

`->` to work it *must* return either a pointer to a class type or an object of a class type that defines its own `->` operator.

Using (Named) Objects to Prevent Memory Leaks

- Place a pointer inside a named object and have the object manage the raw pointer for you

```
1 class StringPtr {
2 public:
3   ~StringPtr() { delete ptr; }
4 private:
5   std::string *ptr;
6 };
```

- User code:

```
1 {
2   StringPtr p(new std::string{"smart pointer"});
3 } <-- *ptr freed here when ~StringPtr() called
```

- The object p's destructor will delete the heap string object

The Basic Idea Behind All Smart Pointers

```
1  #include <iostream>
2
3  class StringPtr {
4  public:
5    StringPtr(std::string *p) : ptr{p} { }
6    ~StringPtr() { delete ptr; }
7    std::string* operator->() { return ptr; }
8    std::string& operator*() { return *ptr; }
9  private:
10   std::string *ptr;
11 };
12
13 int main() {
14   std::string *ps = new std::string{"smart pointer"};
15   StringPtr p{ps};
16   std::cout << *p << std::endl;
17   std::cout << p->size() << std::endl;
18 }
```

Will look at smarter pointers next week

Type Conversion Operators

Two-way user-defined conversions when defining a new class:

- Single-arg constructors: old types to the new type
- the new type to some old types:

  ```
  operator type() const;
  ```

Conversion Operators

Conversion operators must be defined as member functions. They do not take any parameters, nor do they specify a return type. Typically conversions don't modify the object and are declared `const`.

User-Defined Conversion Operators

- A class for allows Tiny objects to be mixed with ints:

```
1  class Tiny {
2  public:
3    class Bad_range { };
4    Tiny(int i) { assign(i); }
5    operator int() const { return v; }
6  private:
7    void assign(int i) {
8      if (i&~077) throw Bad_range();
9      v = i;
10   }
11   int v;
12 };
```

- User code:

```
1  Tiny t1{1};
2  std::cout << t1 + 1; // equivalent to: int(t1) + 1;
```

Explicit Conversion Operators

- A class for allows Tiny objects to be mixed with ints:

```
1  class Tiny {
2  public:
3    class Bad_range { };
4    Tiny(int i) { assign(i); }
5    explicit operator int() const { return v; }
6  private:
7    void assign(int i) {
8      if (i&~077) throw Bad_range();
9      v = i;
10   }
11   int v;
12 };
```

- User code:

```
1  Tiny t1{1};
2  std::cout << t1 + 1; // error
3  std::cout << static_cast<int>(t1) + 1; // ok
```

User-Defined Conversion Operators (Cont'd)

- User-defined conversions are useful because a constructor can specify type conversions but it cannot specify:
  - a conversion from a user-defined to a basic type, or
  - a conversion from a new class to an existing class (if you cannot modify the existing class)
- Use operator overloading or conversion operator but not both:

```
1 int operator+(Tiny, Tiny);
2
3 void f(Tiny t, int i) {
4   t + i; // ambiguous:
5          // operator+(t, Tiny(i)) or int(t) + i
6 }
```

Name Lookup /Resolution (§14.9.1 – 14.9.3)

Avoiding Ambiguous Conversions

You might be tempted to overload as many operators and provide as many conversions as you can think of, but less is more! Design wisely and only provide what is necessary!

The implicit and explicit type conversions we define for our class types compound with the C++ type conversions to create a veritable maze of actual and possible types. The compiler has to use these types for argument matching and overload resolution. This has to be done in a deterministic way, thus if there are multiple possibilities with equal fit then the compiler will report an ambiguity error and list the possible candidates.