COMP2004
Programming Practice
2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Operator Overloading

- Makes classes act like built-in types
- It has both good and evil uses
- The key is not to overuse it
- Using for output is almost always good
- Allows any class to be output:
  - List list;
  - std::cout << list << std::endl;

Output Operator Overloading

- Not a class method, ie. an external function:

```
class List {
    // ...
};

ostream& operator<<(ostream &os,
                    const List &list) {
    // ...
}
```

Output Operator Overloading

```
ostream& operator<<(ostream &os,
                    const List &list) {
    Node *c = list.head;
    if (c) {
        os << c.number;
        c = c->next;
    }
    for (; c; c = c->next)
        os << ' ' << c.number;
    return os;
}
```

Friend functions

- This requires access to List internals
- Classes can declare functions as being friends
- The function can then access class internals

```
class List {
    // ...
    friend ostream& operator<<
        (ostream &os, const List &list);
};
```

Better Design

- If a function doesn't need to be a friend it shouldn't be
- If the class internals change, then friend functions might also have to
- OO is supposed to stop this by hiding internals
- operator<< shouldn't need to be a friend

Operator<<

```
ostream& operator<<(ostream &os,
                    const List &list) {
    List::ConstIterator b = list.begin(),
                        e = list.end();
    if (b != e) {
        os << *b;
        ++b;
    }
    for (; b != e; ++b)
        os << ' ' << *b;
    return os;
}
```

Converting types

- Converting other types to List
- Define a constructor with a single parameter, eg:
  `List::List(const std::string &s);`
- Also acts as user-defined conversion
- No need for corresponding assignment operator

Conversion operators

- Converting List to other types
  ```
  List::operator bool() const {
      return head;
  }
  ```
- No return type (since it's implied)
- Works for other types and classes too
- Easy to run into problems though
  - Particularly memory leaks
  - Usually due to implicit conversions

Conversion operator example

```
List::operator bool() const {
    return head;
}
int main() {
    List l;
    if (l)
        cout << "List has stuff" << endl;
    else
        cout << "List is empty" << endl;
}
```

Inheritance

- Inheritance in C++ is a bit complicated
- The defaults are usually not what you want

```
class Alarm {
public:
    void turn_on() {
        std::cout << "Alarm on\n";
    }
};

class BuzzerAlarm : public Alarm {
public:
    void turn_on() {
        std::cout << "Buzzer on\n";
    }
};
```

What Is Output?

```
void activate(Alarm a) {
        a.turn_on();
}

int main() {
        BuzzerAlarm b;
        activate(b);
}
```

A Problem

- It is called slicing
- The Alarm part of b is passed
- The BuzzerAlarm part is not
- This is almost always an error
- We can stop slicing by passing by reference or pointer

First Attempt At Fix

```
void activate(Alarm &a) {
        a.turn_on();
}

void activate(Alarm *a) {
        a->turn_on();
}
```

More Problems

- It still doesn't work
- Slicing no longer occurs
- Now we have a binding problem
- We want dynamic (or late) binding
- C++ defaults to compile time (or early) binding
- We can fix this too

Fixed Alarm class

```
class Alarm {
public:
        virtual void turn_on() {
                std::cout << "Alarm on\n";
        }
};
```

Virtual

- A virtual function uses runtime lookup
- Also known as dynamic dispatch
- It is slower and uses more memory
- On mono we have:
  - 1.6 times as long as a normal call
  - class is 4 bytes larger
- But it lets you use OO techniques, ie:
  - BuzzerAlarm b;
  - Alarm &a = b;
  - a.turn_on();

Accessibility and Inheritance

- private members are not accessible by derived classes
  - This is a good thing
- protected members are accessible by derived classes
  - Provides an interface for derived classes

Accessibility Example

```
class Base {
      public:      int i;
      protected:   int j;
      private:     int k;
};
class Child : public Base {
      void test() {
            i = 1;   // legal
            j = 1;   // legal
            k = 1;   // ERROR
      }
};
```

Constructors and Inheritance

- Base class constructors must be called (manually)
- This is done with initialisers
- If you don't the default constructor will be called

Constructors Example

```
class Person {
public:
   Person(const std::string &name);
};

class Student : public Person {
public:
   Student(const std::string &name, const
         std::string &sid) : Person(name) {
      // ...
   }
};
```

Destructors and Inheritance

- Destructors are automatically called
- Non-virtual destructors can cause problems
- All classes which may have child classes should have virtual destructors

Abstract Classes

- Separating interface from implementation is useful
- In C++ this can be done with abstract classes
- An abstract class is a class with at least one pure virtual method
  - Can still have normal methods and variables
- Can't create objects of abstract classes

Abstract Example

```
class Alarm {
public:
   virtual void turn_on() = 0;
   virtual void turn_off() = 0;
   virtual bool is_on() = 0;
   virtual ~Alarm() { };
};
```

Multiple Inheritance

- C++ supports multiple inheritance
- Can be useful - often leads to problems
- Common base classes can cause problems
- `virtual` inheritance solves most problems
- You don't need MI for the assignments
- It won't be in the exam either

Non-public Inheritance

`class B : private A`

- public and protected members of A are private in B
- Doesn't affect the B class itself
  - Just other classes
  - Including derived classes

`class B : protected A`

- public members of A are protected in B
- Again doesn't affect B itself

Non-public Inheritance Uses

- Allows inheritance of implementation but not interface
- Following code not allowed

```
class A { };
class B : private A { };
B b;
A *a = &b;
```

- The object `b` is not of type `A`
  - For the "is-a" rule

Namespaces

- Along with classes C++ also provides namespaces
- Namespaces provide a way to make logical groupings
- Standard library items are placed in namespace `std`
- Namespaces allow name reuse

Namespace Example

```
namespace a {
   std::string func() { return "a func\n"; }
}
namespace b {
   std::string func() { return "b func\n"; }
}
std::string func() { return "func\n"; }
int main() {
   std::cout << a::func() << std::endl;
   std::cout << b::func() << std::endl;
   std::cout << func() << std::endl;
}
```

Using Namespaces

- It's possible to "pull in" names from a namespace

```
namespace A { int fred = 10; }
int fred = 20;
int main() {
    std::cout << fred << std::endl;
    using A::fred;
    std::cout << fred << std::endl;
    std::cout << A::fred << std::endl;
}
```

Using Namespaces II

- Entire namespaces can be pulled in

```
namespace A {
    int fred = 10;
    double bill = 10.0;
}
int main() {
  using namespace A;
  std::cout << fred << ' ' << bill;
  std::cout << std::endl;
}
```

Unnamed Namespaces

- Namespaces can be used to hide data
- Not generally very useful

```
namespace {
    int number;
    void function() { ... }
}
```

- Equivalent to

```
namespace ??? { ... }
using namespace ???;
```

Namespace Aliases

- Namespaces reduce name clashes
- For namespace names:
  - Very short names may clash
  - Long names are a pain to type
- Aliases give the best of both worlds
  - Long names - less clashes
  - Short aliases - no need for lots of typing

Alias Example

```
namespace A_Name_Which_Is_Long {
    int foo = 42;
}

namespace short =
             A_Name_Which_Is_Long;

int main() {
    std::cout << short::foo << std::endl;
}
```