COMP2004
Programming Practice
2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Exam information

- Location: MacLaurin Hall, Main Quadrangle
- Duration: 2 hrs (10 mins reading)
- Date: Friday 15 February 2002
- Time: 9:20am to 11:30am
- Format: Closed book

- Clash => see or email me ASAP

Course Overview

- Basic C++
- Object-oriented C++
- Unix development tools
- Advanced C++
- C++ Standard Template Library (STL)

Course Overview

- Basic C++
  - Simple syntax, input/output, strings, vectors, etc
  - Pointers, arrays, memory allocation
- Object-oriented C++
  - Classes, operator overloading
  - Inheritance, namespaces
- Unix development tools
  - Shell, make, gdb, revision control

Course Overview

- Advanced C++
  - Generic code
  - Exceptions and exception safety
  - String streams
- C++ Standard Template Library (STL)
  - Iterators, containers
  - Function objects, algorithms

Basic C++

- IO (input / output)
  - Buffered / unbuffered, `cout` / `cerr`
- Manipulators
  - `cout << setw(10) << 42;`
- Strings
  - All obvious operators work
    - `s[0]`, `+`, `+=`, `==`, `<`, etc
  - Convert from C-style
    - `char *c = "hello"; string s(c);`
  - Convert to C-style
    - `s.c_str()`

Passing parameters

- Pass by value
  - void func(int value)
  - Copies values
- Pass by reference
  - void func(int& value)
  - Direct access to original object
- Pass by const reference
  - void func(const int& value)
  - Direct access, but no changes allowed

Pointers

```
int main() {
    int i = 42;
    int *p = &i;
    *p = 3;
    p = NULL;
    if (p)
        cout << *p << endl;
}
```

Arrays

```
int main() {
    int a[10];
    int b[ ] = { 1, 2, 3 };
    char c[ ] = "a c-style string";
    char *d = c;

    a[3] = b[1];
    cout << *c << *d << endl;
    d += 3;
    cout << d[1] << *(c + 4) << endl;
}
```

Arguments to main()

```
int main(int argc, char **argv) {
    if (argc >= 3) {
        cout << argv[2] << endl;
    }
}
```

```
bash$ prog These are the args
the
```

Memory Allocation

- Automatic - Local variables
  - Exist only during local scope
- Static
  - Global variables
  - Static local variables
  - Exist from program start to finish
- Dynamic - new and delete
  - Exist between explicit creation and destruction

Memory Leaks

- Every new needs a corresponding delete

```
int main() {
    int *data = new int(4);
    data = new int(12);
    data = new int(42);
    delete data;
}
```

Object-oriented C++

- Structs and classes
  - Structs default to public
  - Classes default to private

```
Student s;
Student *sp = &s;
s.sid = "9222194";
sp->courses.push_back("comp2004");
```

Example minimal class

```
class A {
public:
    A();
    A(const A& o);
    ~A();
    A& operator=(const A& o) {
        if (this == &o) return *this;
        ...
    }
};
```

Const correctness

```
class List {
    int length() const;
    void clear();
}
```

- length() method cannot modify object
- const reference cannot call non-const method
  - `const List &l;`
    `l.clear();    // not allowed`

Operator overloading

- Makes classes act like built-in types
- Eg: output operator:
- `ostream& operator<<(ostream &os,`

```
                      const List& list) {
    ...
    return os;
}
```

Type conversions

- Other to List:
  - `List::List(const string &s)`
  - Used to convert string to List
- List to other:
  - `List::operator bool() const`
  - Used to convert List to bool
  - Return type obviously bool

Inheritance

- `class BuzzerAlarm : public Alarm {`

```
    ...
};
```

- Don't pass base class objects by value:
  - `void activate(Alarm a)`
- The copying causes slicing
- Pass (const) reference instead:
  - `void activate(Alarm& a)`
  - `void activate(const Alarm& a)`

Accessibility

- public members visible to all
- private members hidden from children
- protected members visible to children, but still hidden from others

- Use public to provide interface to others
- Use protected to provide interface to children

Virtual methods

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

- If turn_on() is not virtual:
  - Calls turn_on() in Alarm
- If turn_on() is virtual:
  - Calls turn_on() in BuzzerAlarm
- Virtual methods require virtual destructor

Pure virtual methods

```
class Alarm {
    virtual void turn_on() = 0;
};
```

- Alarm is abstract
  - Cannot create Alarm objects
- Child classes must redefine turn_on()

Namespaces

```
namespace lib {
    string func() { ... }
}

cout << lib::func() << endl;
using namespace lib;
cout << func() << endl;
```