CMPE 180-92

# Data Structures and Algorithms in C++

October 12 Class Meeting

---

Department of Computer Engineering
San Jose State University


Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

# Notes and Schedule

- 2-3: You have a complex object, and you are adding another one to it.
- 2.4: You have a complex object, and you are adding a real number to it.
- 2.5: You are adding a real number and a complex number, in that order.

- Midterm until 7:30
- Break until 7:45

- Then a short lecture about templates, inheritance, and polymorphism.

# Function `exchange`

- A useful function that exchanges the values of its two parameters:

```
void exchange(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}
```

- This version only works with integers.
- Can we define a version that works with multiple types?

# Templates

ExchangeTemplate.cpp

```
template <typename T>
void exchange(T& first, T& second)
{
    T temp = first;
    first = second;
    second = temp;
}

template <typename T>
void print(T first, T second)
{
    cout << first << " " << second << endl;
}
```

- This is not actual code – it's a template (mold) for the compiler to generate source code on an as-needed basis.

# Templates, *cont'd*

ExchangeTemplate.cpp

```
int main()
{
    int i = 5, j = 7;
    print(i, j);
    exchange(i, j);
    print(i, j);

    cout << endl;

    double pi = 3.14, e = 2.72;
    print(pi, e);
    exchange(pi, e);
    print(pi, e);

    ...
}
```

Generate `int` versions of the `exchange` and print functions

Generate `double` versions of the `exchange` and print functions

Demo

# Template Class Example

Pair.h

```
template <typename T1, typename T2>
class Pair
{
public:
    Pair(T1 a_value, T2 b_value);
    T1 first()  const;
    T2 second() const;

private:
    T1 a;
    T2 b;
};

template <typename T1, typename T2>
Pair<T1, T2>::Pair(T1 a_value, T2 b_value) : a(a_value), b(b_value) {}

template <typename T1, typename T2>
T1 Pair<T1, T2>::first()  const { return a; }

template <typename T1, typename T2>
T2 Pair<T1, T2>::second() const { return b; }
```

# Template Class Example, *cont'd*

PairTests.cpp

```
#include "Pair.h"

using namespace std;

template <typename T1, typename T2>
ostream& operator <<(ostream &outs, Pair<T1, T2>& p);

int main()
{
    Pair<int, double>    p1(2, 3.14);
    Pair<double, string> p2(3.14, "Hello");
    Pair<string, string> p3("Bob", "Ron");

    cout << p1 << endl;
    cout << p2 << endl;
    cout << p3 << endl;
}

template <typename T1, typename T2>
ostream& operator <<(ostream &outs, Pair<T1, T2>& p)
{
    outs << p.first() << " " << p.second();
    return outs;
}
```

```
2 3.14
3.14 Hello
Bob Ron
```

Demo

# A “Safe” Array Type: Version 6

SafeArray6.h

```
template <typename T>
class SafeArray
{
public:
    SafeArray();
    SafeArray(int len);
    SafeArray(const SafeArray<T>& other);  // copy constructor
    ~SafeArray();

    int get_length() const;

    SafeArray<T>& operator =(const SafeArray<T>& rhs);
    T& operator [](int i) const;

private:
    T *elements;
    int length;
};
```

# A "Safe" Array Type: Version 6, *cont'd*

SafeArray6.h

```
template <typename T>
SafeArray<T>::SafeArray() : elements(nullptr), length(0)
{
}

template <typename T>
SafeArray<T>::SafeArray(int len) : elements(nullptr), length(len)
{
    elements = new T[length];
}

template <typename T>
SafeArray<T>::SafeArray(const SafeArray<T>& other)
    : elements(nullptr), length(0)
{
    length = other.length;
    elements = new T[length];

    for (int i = 0; i < length; i++)
    {
        elements[i] = other.elements[i];
    }
}
```

# A "Safe" Array Type: Version 6, *cont'd*

SafeArray6.h

```
template <typename T>
SafeArray<T>::~SafeArray()
{
    if (elements != nullptr) delete[] elements;
}

template <typename T>
int SafeArray<T>::get_length() const { return length; }

template <typename T>
SafeArray<T>& SafeArray<T>::operator =(const SafeArray<T>& rhs)
{
    if (this == &rhs) return *this;
    if (elements != nullptr) delete[] elements;

    length = rhs.length;
    elements = new T[length];

    for (int i = 0; i < length; i++)
    {
        elements[i] = rhs.elements[i];
    }

    return *this;
}
```

San José State University

# A “Safe” Array Type: Version 6, *cont’d*

SafeArray6.h

```
template <typename T>
T& SafeArray<T>::operator [](int i) const
{
    assert((i >= 0) && (i < length));
    return elements[i];
}
```

# A “Safe” Array Type: Version 6, *cont’d*

SafeArrayTests6.cpp

```
template <typename T>
void print(SafeArray<T> a);

void test_int();
void test_string();

int main()
{
    test_int();
    cout << endl;
    test_string();
}

template <typename T>
void print(SafeArray<T> a)
{
    for (int i = 0; i < a.get_length(); i++)
    {
        cout << " " << a[i];
    }
    cout << endl;
}
```

# A "Safe" Array Type: Version 6, *cont'd*

SafeArrayTests6.cpp

```
void test_int()
{
    SafeArray<int> a1(10), a2, a3;

    for (int i = 0; i < 10; i++) a1[i] = 10*i;

    a3 = a2 = a1;
    a1[4] = -a1[4];

    cout << "a1 ="; print(a1);
    cout << "a2 ="; print(a2);
    cout << "a3 ="; print(a3);
}
```

```
a1 = 0 10 20 30 -40 50 60 70 80 90
a2 = 0 10 20 30 40 50 60 70 80 90
a3 = 0 10 20 30 40 50 60 70 80 90
```

# A “Safe” Array Type: Version 6, *cont’d*

SafeArrayTests6.cpp

```
void test_string()
{
    SafeArray<string> a1(4), a2, a3;

    a1[0] = "Fee";
    a1[1] = "Fie";
    a1[2] = "Foe";
    a1[3] = "Fum";

    a3 = a2 = a1;
    a1[2] = "XXX";

    cout << "a1 ="; print(a1);
    cout << "a2 ="; print(a2);
    cout << "a3 ="; print(a3);
}
```

```
a1 = Fee Fie XXX Fum
a2 = Fee Fie Foe Fum
a3 = Fee Fie Foe Fum
```

Demo

# Object-Oriented Programming

- Encapsulation
  - Classes
- Inheritance
  - Subclasses
- Polymorphism
  - Virtual functions

# Inheritance

- A very powerful and important feature of object-oriented programming.
- A new class (the derived class) is created from another class (the base class).
- A derived class is also known as a child class.
- The base class is the parent class.
- A child class is also known as a subclass.

# Inheritance, *cont'd*

- A child class inherits member variables and functions from its parent class.

```
class Person
{
public:
    string activity() { return "Eat and sleep."; }
};

class Student : public Person
{
public:
    string study() { return "Study and study." ; }
};
```

A **Student** "is a" **Person**.

# Inheritance, *cont'd*

```
class Person
{
public:
    string activity() { return "Eat and sleep."; }
};

class Student : public Person
{
public:
    string study() { return "Study and study." ; }
};
```

- Let **`s`** be type **`Student`**.
  - Valid: **`s.study()`**
  - Valid: **`s.activity()`**

# Inheritance, *cont'd*

```
class Person
{
public:
    string activity() { return "Eat and sleep."; }
};

class Student : public Person
{
public:
    string study() { return "Study and study." ; }
};
```

- Subclass **`Student`** inherits the member function **`activity`** from its parent class.
  - The **`Student`** class can also override the definition of function **`activity`** by defining its own version.

```
class Animal
{
public:
    string speak() { return "Shhh!"; }
};

class Mammal : public Animal
{
public:
    string speak() { return "Grrr!" ; }
};

class Cat : public Mammal
{
public:
    string speak() { return "Roar!"; }
};

class Kitty : public Cat
{
public:
    string speak() { return "Meow!"; }
};

string make_sound(Cat& c) { return c.speak(); }

int main()
{
    Kitty k;
    cout << make_sound(k) << endl;
}
```

# Subclasses

- Variable **k** is a **Kitty**.
  - **k** is also a **Cat**, a **Mammal**, and an **Animal**.
  - Each subclass overrides the definition of member function **speak**.
- What is the output?
  - The type of parameter **c** is a **Cat**.

**Roar!**

# Polymorphism

- Polymorphism is the ability of a variable to have different behaviors at run time.
- How the variable behaves depends not on the type of the variable, but on the type of its value at run time.
- Polymorphism is implemented in C++ with virtual functions.

# Polymorphism, *cont'd*

```
class Person
{
public:
    virtual string activity() { return "Eat and sleep."; }
};

class Student : public Person
{
public:
    string activity() { return "Study and study." ; }
};

class EngineeringMajor : public Student
{
public:
    string activity() { return "Design and build."; }
};

class SoftwareMajor : public EngineeringMajor
{
public:
    string activity() { return "Code and test."; }
};

string do_it(Student& s) { return s.activity(); }

int main()
{
    SoftwareMajor sw;
    cout << do_it(sw) << endl;
}
```

## What is the output?

- Member function **`activity`** is virtual in **`Person`** and all subclasses.
- The type of parameter **`s`** is **`Student`**.
- The type of the value of **`s`** is **`SoftwareMajor`**.

```
Code and test.
```

# Polymorphism, *cont'd*

```
string activity(Student& s) { return s.activity(); }

int main()
{
    SoftwareMajor sw;
    cout << do_it(sw) << endl;

    EngineeringMajor em;
    cout << do_it(em) << endl;

    Student st;
    cout << do_it(st) << endl;
}
```

```
Code and test.
Design and build.
Study and study.
```

Demo

# Virtual Destructors

- From now on, make destructors virtual.
  - Example:

```
virtual ~Foo();
```

- A virtual destructor ensures that the correct destructor is called for an object when the object is being destroyed.