

slide.com
jhu intprog

601.220 Intermediate Programming

Summer 2022, Meeting 17 (July 18th)

Today's agenda

- Review exercises 29 and 30
- Day 31 recap questions
- Exercise 31
- Day 32 recap questions
- Exercise 32

Reminders/Announcements

- HW7 is due **Thursday, July 21st**
 - We are covering template classes today, this material is needed for the TTree class
- Final project team formation
 - Submit the Google form in Piazza post 218 (pinned) by **11 am on Tuesday (July 19th)**
 - If you aren't registered as being on a team by Wednesday (July 20th) you will be assigned to a team

Exercise 29 review

Complex c;

;

cout << c;

Overloading the output stream insertion operator for the Complex class:

```
// in complex.h (in the Complex class definition)
```

```
friend std::ostream& operator<<(std::ostream &out,  
                                const Complex &c);
```

```
// in complex.cpp
```

```
std::ostream& operator<<(std::ostream &out,  
                        const Complex &c) {  
    out << c.rel << " + " << c.img << "i";  
    return out;  
}
```

Exercise 29 review

Copy constructor and assignment operator

// in complex.cpp

```
Complex::Complex(const Complex &other)
    : rel(rhsother), img(other.img) {
}
```

```
Complex& Complex::operator=(const Complex &rhs) {
    → if (this != &rhs) {
        rel = rhs.rel;
        img = rhs.img;
    }
    return *this;
}
```

Exercise 29 review

Complex a, b;
:
a + b \longleftrightarrow a.operator+(b)

Overloaded operators for arithmetic, in complex.h (in the class definition for the Complex class):

```
Complex operator+(const Complex& rhs) const;  
Complex operator-(const Complex& rhs) const;  
Complex operator*(const Complex& rhs) const;  
Complex operator*(const float& rhs) const;  
Complex operator/(const Complex& rhs) const;
```


Since these are defined as member functions, they only need one parameter, which is the right-hand-side operand. (The left hand Complex object in the expression will be the receiver object.)

Exercise 29 review

Implementations of arithmetic operators in `complex.cpp`:

```
Complex Complex::operator+(const Complex& rhs) const {  
    Complex sum(rel+rhs.rel, img+rhs.img);  
    return sum;  
}
```

```
Complex Complex::operator-(const Complex& rhs) const {  
    return *this + (rhs * -1.0f);  
}
```



Exercise 29 review

Implementation of multiplication:

```
Complex Complex::operator*(const Complex& rhs) const {  
    float a = rel;  
    float b = img;  
    float c = rhs.rel;  
    float d = rhs.img;  
    // (a+bi) * (c+di) = a*c + (a*d)i + (b*c)i + (b*d)(i^2)  
    //                  = (a*c - b*d) + (a*d + b*c)i  
    return Complex(a*c - b*d, a*d + b*c);  
}
```

Handwritten notes:

- A blue bracket on the left groups the four float assignments.
- A blue bracket underlines the return statement.
- A blue circle highlights the i^2 term in the comment, with a handwritten -1 above it.

Exercise 29 review

Non-member `*` operator for float times Complex:

```
// in complex.h
```

```
friend Complex operator*(float lhs, const Complex &rhs);
```

```
// in complex.cpp
```

```
Complex operator*(float lhs, const Complex &rhs) {  
    return rhs * lhs;  
}
```

This operator can't be a member function because the value on the left-hand-side is not an object. Also, this function technically doesn't need to be a friend because it invokes the public `Complex` times `float` operator.

Exercise 30 review

// copy constructor

```
int_set::int_set(const int_set& orig)
    : head(nullptr), size(0) {
    *this = orig;
}
```

// destructor

```
int_set::~int_set() {
    clear();
}
```

frick



Exercise 30 review

$a = (b = c);$

```
// += operator  
int_set& int_set::operator+=(int new_value) {  
    add(new_value);  
    return *this;  
}
```

Exercise 30 review

```
// assignment operator
int_set& int_set::operator=(const int_set &rhs) {
    if (this != &rhs) {
        clear(); // delete old linked list

        int_node *n = other.head;
        while (n != nullptr) {
            add(n->get_data());
            n = n->next;
            n -> get_next();
        }
        return *this;
    }
}
```

Note: inefficient because add is $O(N)$. Overall running time is $O(N^2)$.

Exercise 30 review

$(cout \ll a) \ll b;$

// output stream insertion operator

```
std::ostream& operator<<(std::ostream& os, const int_set& s){  
    int_node *n = s.head;  
    os << "{";  
    while (n != nullptr) {  
        os << n->get_data();  
        if (n->get_next() != nullptr) { os << ", "; }  
        n = n->get_next();  
    }  
    os << "}";  
    return os;  
}
```

Day 31 recap questions

- ① How do we declare a template function?
- ② Under what conditions would you consider making a function templated?
- ③ What is template instantiation?
- ④ Can we separate declaration and definition when using templates?
- ⑤ Why shouldn't template definitions be in .cpp files?

1. How do we declare a template function?

// Example

→ `template<typename T>`

type parameter

```
T get_max(const T &left, const T &right) {  
    if (left > right) { return left; }  
    else { return right; }  
}
```

Type parameter inference

T is a “type parameter”. In a call to `get_max`, T is inferred from the argument type. E.g.

```
int a = get_max(3, 4);           // T is int
double b = get_max(5.0, 6.0);    // T is double
```

```
std::string s1 = "hi", s2 = "hello";
std::string c = get_max(s1, s2); // T is std::string
```


2. Under what conditions would you consider making a function templated?

Template functions are usefful when you want to allow the function to work with a variety of different data types.

The data types that will be substituted for the type parameter(s) must have common operations that will be used by the template function.

For example, the `get_max` function (shown previously) requires the data type `T` to have a `>` operator, and also a copy constructor.

All of the built-in types (`int`, `double`, etc.) have these a `>` operator, assignment operator, and copy constructor. (A.k.a. “value semantics”.)

3. What is template instantiation?


Template instantiation is the substitution of actual data types for type parameters. For example, in

```
int a = get_max(3, 4);
```

the type `int` is substituted for `T`. So:

```
template<typename T>
T get_max(const T &left, const T &right) {
    if (left > right) { return left; }
    else                { return right; }
}

// instantiated with T=int
int get_max(const int &left, const int &right) {
    if (left > right) { return left; }
    else                { return right; }
}
```



4. Can we separate declaration and definition when using templates?

It is possible to separate the declaration and definition of template functions and classes.

However, this is more complicated and less flexible than just putting the definition of the template function or class in a header file.

The compiler doesn't normally instantiate a template function or class until the function or class is actually used. To instantiate the function or class, the compiler needs the definition.

So, we generally put the definitions for template functions and classes directly in a header file.

5. Why shouldn't template definitions be in .cpp files?

Template functions (including member functions of template classes) can't be compiled into machine code and put into an object (.o) file.

The basic problem is that there are an infinite number of possible ways a template function or class could be instantiated. For example, `vector<int>`, `vector<string>`, `vector<YourClass>`, etc. The compiler doesn't know which instantiations your program will need until it actually sees the code that uses `vector`, and knows which types will be substituted for `vector`'s type parameter.

“.inc” files

One way to allow template function definitions to be separated from a template class declaration, but still be available from a header file, is to use an “.inc” file. The .inc file contains the definitions of member functions of the template class. The header file would look like this:

```
#ifndef MY_SET_H
#define MY_SET_H
```

```
{ template<typename T> class my_set {
    // declarations of member functions of my_set<T>
};
```

```
➡ #include "my_set.inc"
```

```
#endif // MY_SET_H
```

Exercise 31

11:20

- Convert your ^{int} ~~E~~set class from Exercise 30 to a template class
- Note that ex30-sol (Exercise 30 reference solution) has been added to the public repo, `git pull` to get it
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have questions!

Day 32 recap questions

- ❶ Do derived classes inherit constructors?
- ❷ What does protected imply for a class field?
- ❸ What is polymorphism?
- ❹ What is the purpose of the `virtual` keyword?
- ❺ Can a child class have multiple parents?

1. Do derived classes inherit constructors?

No. Each derived class must define its own constructors. These will call one of the base class's constructors in its initializer list.

```
// example base class
```

```
class Point2D {
```

```
private:
```

```
    double x, y;
```

```
public:
```

```
    Point2D() : x(0.0), y(0.0) { }
```

```
    Point2D(double x, double y)
```

```
        : x(x), y(y) { }
```

```
    double get_x() const { return x; }
```

```
    double get_y() const { return y; }
```

```
};
```


Derived class, constructors

```
// derived class
class Point3D : public Point2D {
private:
    double z;

public:
    Point3D() : Point2D(), z(0.0) { }
    Point3D(double x, double y, double z)
        : Point2D(x, y), z(z) { }

    double get_z() const { return z; }
};
```

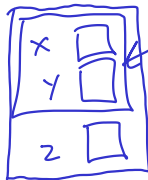
Call base class constructors

Picture of Point2D and Point3D objects

Point2D



Point3D



base class
object "embedded"

2. What does protected imply for a class field?

A protected member may be directly accessed by code in derived classes, but may not be accessed by code in “unrelated” classes or functions.

Opinion: It is never really necessary to make a member function protected. Derived classes can (and should) use public getter and setter functions to access private data values in the base class.

3. What is polymorphism?

Polymorphism is the phenomenon that anywhere in a program that there is either a pointer to a base class type or a reference to a base class type, that pointer or reference could really refer to an object that is an instance of a class derived from the base class type.

E.g.:

```
public class Dog : public Animal { ... };  
public class Cat : public Animal { ... };  
public class Owl : public Animal { ... };
```

```
void do_stuff(Animal &a) {
```

```
    // the reference could refer to a Dog, Cat, or Owl object,  
    // or an instance of any class deriving from Animal
```

```
}
```

LSP

an instance of
a derived class
may be used
anywhere a
reference to a
base class object
is allowed

or pointer

4. What is the purpose of the `virtual` keyword?

The `virtual` keyword marks a member function that can be overridden by a derived class. This allows the derived class to provide its own behavior for that member function.

* A base class will *usually* have at least one virtual member function. The idea is that virtual member functions in the base class define common operations which can be implemented by derived classes with varying behavior.

Example base class with a virtual member function

*abstract
concept*

```
// base class with a virtual member function representing  
// a common operation  
class Animal {  
public:  
    virtual void vocalize() { cout << "?\n"; }  
    // ...  
};
```

?

Example derived classes overriding a virtual member function

concrete concepts

```
class Dog : public Animal {  
public:  
    virtual void vocalize() { cout << "woof\n"; }  
    // ...  
};
```

```
class Cat : public Animal {  
    virtual void vocalize() { cout << meow"cat\n"; }  
    // ...  
};
```

Polymorphism!

```
void stuff(Animal &a) {  
    a.vocalize();  
}
```

```
int main() {  
    Dog leo;  
    Cat ingo;  
  
    stuff(leo); // prints "woof"  
    stuff(ingo); // prints "meow"  
}
```


5. Can a child class have multiple parents?

Yes. However, this is a feature that is not used very widely.

One example: the iostream type inherits from both `istream` and `ostream`.

Since stringstream inherits from `iostream`, this explains why you can both read data from a `stringstream` and also write data to a `stringstream`.

Exercise 32

- Practice with examples of classes using inheritance and `virtual` member functions
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have questions!

Notes

Notes

Notes

Notes

Notes