Friends of Template Classes

- Template classes can contain friends.
- A friend function that does not use a template specification is universally a friend of all instantiations of the template class.
- A friend function that incorporates template arguments is specifically a friend of its instantiated class.

Example:

```cpp
template <class T>
class Matrix
{
    public:
        friend void Foo(); // universal
        friend vect<T> Product(vect<T> v); // instantiated
        ...
};
```

Template Arguments

You can also use constant expressions, function names, and character strings as template arguments.

Example:

```cpp
template <class T, int n>
class AssignArray
{
    public:
        T a[n];
};
```

AssignArray<double, 50> x, y;
x = y;

Default Template Arguments

- In the standard library, the class complex is now a template class.
- Usually, we would instantiate it to double: complex<double> x, y, z[10];
- A template provider could decide that this is such a common case that it should be provided as a default.
- To achieve this, we can use default template arguments.

Example:

```cpp
template<class T = double>
class complex
{
    ...
    private:
        T real, imaginary;
    ...
    complex<> c; // same as complex <double> c;
};
```

Inheritance

A class can be derived from an existing class by using the following form:

```
class class-name:(public|protected|private) base-name
{  
    member declarations
};
```

The keywords public, protected, and private are used to specify how the base-class members can be accessed by the derived class.

Public derivation is far more important than private or protected derivation; it should be considered the normal form of inheritance.
Inheritance

Example:

```cpp
class Student {
public:
    enum year {fresh, soph, junior, senior, grad};
    Student(string nm, int id, double g, year y);
    void Print() const;
protected:
    string name;
    int student_id;
    double gpa;
    year yr;
};
```

Inheritance

```cpp
class GradStudent : public Student {
public:
    enum support {ta, ra, fellowship, other};
    GradStudent(string nm, int id, double g, year y, 
                 support s, string d, string th);
    void Print() const;
protected:
    support supp;
    string dept;
    string thesis;
};
```

Inheritance

In the preceding example, GradStudent is the derived class, and Student is the base class. Using the keyword public following the colon in the derived-class header has several effects:

- The protected and public members of Student are inherited as protected and public members, resp., of GradStudent.
- Private members are inaccessible.
- GradStudent is a subtype of Student.
- A GraduateStudent is a Student, but a Student does not have to be a GraduateStudent (is-a relationship, or interface inheritance).

Inheritance

• A derived class is a modification of the base class, inheriting the public and protected members of the base class.
• Notice: Only constructors, destructors, and the member function operator() cannot be inherited.
• Frequently, a derived class adds new members to the existing class members.
• It is also possible to override existing class members.
• Notice: Overriding is different from overloading, in which the same function name can have different meanings for each unique signature.

Typing Conversions and Visibility

A variable of a publicly derived class can in many ways be treated as if it were the base-class type. For example, a pointer whose type is pointer to base class can point to objects that have the derived-class type.

To examine the properties of base classes and derived classes, let us first take a closer look at our examples Student and GradStudent.

Benefits of using a derived class:

• Code is reused: GradStudent uses existing, tested code from Student.
• The hierarchy reflects a relationship found in the problem domain. In the real world, graduate students make up a subgroup of all students.
• Various polymorphic mechanisms will allow client code to treat GradStudent as a subtype of Student, which simplifies the code but keeps distinctions among subtypes.
Typing Conversions and Visibility

Implementation of constructors:

Student::Student(string nm, int id, double g, year y):
name(nm), student_id(id), gpa(g), yr(y)
{}

GradStudent::GradStudent(string nm, int id, double g, year y, support s, string d, string th):
Student(nm, id, g, y), supp(s), dept(d), thesis(th)
{}

Notice: The constructor of Student is invoked as part of the initializer list in the constructor of GradStudent.

Typing Conversions and Visibility

Because GradStudent is a subtype of Student, a reference to the derived class GradStudent may be implicitly converted to a reference to the public base class Student.

Example:

GradStudent gst("John Miller", 31416, 3.99, grad, ra, "Computer Science", "Eye Movements in the Dark");
Student &st = gst;

Inheritance

class GradStudent : public Student { … }

Using the keyword public following the colon in the derived-class header has several effects:

• The protected and public members of Student are inherited as protected and public members, resp., of GradStudent.
• Private members are inaccessible.
• GradStudent is a subtype of Student.

Inheritance

class GradStudent : protected Student { … }

Using the keyword protected following the colon in the derived-class header has several effects:

• The public and protected members of Student are inherited as protected members of GradStudent.
• However, GradStudent can re-declare protected members of Student as public.
• Private members of Student are inaccessible by GradStudent.
• GradStudent is not a subtype of Student.

Inheritance

class GradStudent : protected Student { … }

Using the keyword protected following the colon in the derived-class header has several effects:

• The public and protected members of Student are inherited as protected members of GradStudent.
• However, GradStudent can re-declare public members of Student as public.
• Private members of Student are inaccessible by GradStudent.
• GradStudent is not a subtype of Student.

Inheritance

class GradStudent : private Student { … }

Using the keyword private following the colon in the derived-class header has several effects:

• The protected and public members of Student are inherited as private members of GradStudent.
• However, GradStudent can re-declare protected members of Student as protected.
• …and public members of Student as either public or protected.
• Private members of Student are inaccessible by GradStudent.
• GradStudent is not a subtype of Student.

Typing Conversions and Visibility

Example for pointer conversions:

int main()
{
    Student s("Joe Smith", 111, 2.57, student::fresh);
    Student *ps = &s;
    GradStudent gs("John Miller", 31416, 3.99, student::grad, ra, "Computer Science", "Eye Movements in the Dark");
    GradStudent *pgs;
    ps->Print(); // Student::Print()
    ps = pgs = &gs;
    pgs->Print(); // GradStudent::Print()
    pgs->Print(); // GradStudent::Print()
}
Virtual Functions

- Overloaded member functions are invoked by a type-matching algorithm.
- These types are known at compile time and allow the compiler to select the appropriate member directly.
- As you will see, it would be nice to dynamically select at runtime the appropriate member function from among base- and derived-class functions.
- Such a mechanism is provided by the keyword virtual; it may be used only to modify member function declarations.
- Virtual functions combined with public inheritance are a form of pure polymorphism.

Virtual Functions

- When a virtual function is invoked, its semantics are the same as those of other functions.
- In a derived class, a virtual function can be overridden by another function with a matching signature.
- The selection of which function definition to invoke for a virtual function is dynamic.
- A pointer to base class can point at either a base-class object or a derived-class object.
- The member function selected will depend on the class of the object being pointed at, not on the pointer type.

Example: Virtual function selection

```cpp
class B
{
public:
    int i;
    virtual void Print() const
    { cout << i << " inside B" << endl; }
};

class D : public B
{
    void Print() const
    { cout << i << " inside D " << endl; }
};

int main()
{
    B b;
    B *pb = &b;
    D d;
    d.i = 1 + (b.i = 1);
    pb->Print();
    pb = &d;
    pb->Print();
}
```

Output:

```
1 inside B
2 inside D
```
Virtual Functions

Notice:

- The declaration of an identifier in a scope hides all declarations of that identifier in outer scopes.
- A base class is an outer scope of any class derived from it.
- This rule is independent of whether the names are declared virtual or not.
- If the selected function is inaccessible, we get a compile-time error.

Virtual Functions

We can define abstract base classes that include pure virtual functions (also called deferred methods).

These are declared as follows:

```
virtual function prototype = 0;
```

An abstract base class specifies the basic common properties of its derived classes but cannot itself be used to declare objects.

It is used to declare pointers that can access subtype pointers derived from the abstract class.

Design Patterns

As software engineers, we commonly have to make decisions about how to create complex objects, how to encapsulate some actions, how to allow for the undoing of certain operations, etc.

Instead of developing individual solutions every time, it is useful to apply standard design patterns.

The patterns have been used and tested by many other software engineers.

Moreover, they are known by many software engineers and well documented in literature so that using them will make it easier for other programmers to understand your code.

Design Patterns

It typically takes programmers some time to get proficient with design patterns.

Once they get used to these techniques, their productivity increases.

There are entire catalogs of design patterns with formal specifications including pattern name, description, etc.

We will look at several important and common design patterns in a less formal way.

The Singleton Design Pattern

The singleton design pattern is a simple creational pattern, i.e., it applies to the creation of objects.

This pattern can be used when at most one instance of a particular object is allowed to exist at any given time.

For example, there should only be one mouse cursor object.

The singleton design pattern prevents direct access to the object constructors but provides an instance() method for calling member operations.

When called for the first time, the instance() method will create the only instance of the object.

The Singleton Design Pattern

```cpp
class Singleton {
public:
    static Singleton* instance();
    void op1();
protected:
    Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
private:
    static Singleton* onlyInstance_;
};

Singleton* Singleton::onlyInstance_ = 0;
Singleton* Singleton::instance() {
    if (onlyInstance_ == 0)
        onlyInstance_ = new Singleton;
    return onlyInstance_;
}
```

Returns pointer to the only Singleton object. Uses lazy evaluation (instance is not created unless explicitly requested).

Operations should be accessed through instantiating operation:

```
Singleton::instance()->op1();
```

Constructors are protected, so the following is impossible:

```
Singleton* s = new Singleton();
```

Operations should be accessed through instantiating operation:

```
Singleton::instance()->op1();
```

Could easily be modified to allow at most n instances.

Returns pointer to the only instance or 0 if no instance exists.

`OnlyInstance or 0 if no instance exists. Could easily be modified to allow at most n instances.`