Function Templates

In a similar way, we can use function templates. Here is a function template for copying arrays of different types:

```cpp
template <class TYPE>
void copy(TYPE a[], TYPE b[], int n)
{
    for (int i = 0; i < n; i++)
        a[i] = b[i];
}
```

Which of the following calls fail to compile?

double f1[50], f2[50];
char c1[25], c2[50];
int i1[75], i2[75];
char *ptr1, *ptr2;
copy(f1, f2, 50);
copy(c1, c2, 10);
copy(i1, i2, 40);
copy(ptr1, ptr2, 100);
copy(i1, f2, 50);
copy(ptr1, f2, 50);
copy(i1, f2, 50);

The last two invocations of copy() fail to compile because their types cannot be matched to the template type.

Static Members of Template Classes

Static members of template classes are not universal but are specific to each instantiation:

```cpp
template <class T>
class Thing
{
public:
    static int count;
    ...
};

Thing<int> a;
Thing<double> b;
```

The static variables Thing<int>::count and Thing<double>::count are distinct.

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:

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:

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:
  ```cpp
  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:
```cpp
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.
```

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;
};
```

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.

**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

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**.

**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.
• A GraduateStudent is a Student, but a Student does not have to be a GraduateStudent (**is-a relationship**, or **interface inheritance**).
Inheritance

```cpp
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`.

Inheritance

```cpp
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`.

Typing Conversions and Visibility

Example for pointer conversions:

```cpp
typedef 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()
    ps->Print();  // Student::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 function 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.

Virtual Functions

- Note the difference in selection of the appropriate **overridden virtual function** from an **overloaded** member function:
  - An overloaded member function is selected at **compile time**, based on its argument types, and it can have distinct return types.
  - A virtual function is selected at **runtime**, based on the **object's type**, which is passed to it as its **this** pointer argument.
  - Once a function is declared **virtual**, this property is **automatically carried along** to all redefinitions in derived classes.
Virtual Functions

Example: Virtual function selection

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;
    pb = &d;
    pb->Print();
    pb->Print();
}

Output:
1 inside B
2 inside D

Virtual Functions

• In the Student/GradStudent example, the selection of Print() is based on the pointer type, known at compile time.
• In the current (B/D) example, selection is based on what is being pointed at.
• Here, the pointer’s base type is not determining the function selection.
• Instead, different class objects are processed by different functions, determined at runtime.

Virtual Functions

Example:

class Shape
{
    public:
        virtual double Area() const
        { return 0; }
    protected:
        double x, y;
    }

class Rectangle : public Shape
{
    public:
        double Area() const
        { return (height*width); }
    private:
        double height, width;
}

class Circle : public Shape
{
    public:
        double Area() const
        { return (PI*radius*radius); }
    private:
        double radius;
}

int main()
{
...
    Shape *s[N];
...
    for (int i = 0; i < N; i++)
        totArea += s[i]->Area();
}

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.