OOP and Generic Programming in C++

Picked from Lippman 5th edition

Object Oriented Programming

- Classes connected by inheritance tend to form a hierarchy.
- The base class defines as `virtual` those functions which it expects its derived classes to define for themselves.
- The derived class should specify which classes it wants to inherit from, as `class Derived : public Base`.
- The base class should define a virtual destructor.
- A nonstatic member function is declared virtual by adding `virtual` infront of the prototype, only in the declaration inside the class definition.
- Virtual functions are also implicitly virtual in the derived class.
- The protected access specifier lets only inherited classes access specific members.
- We can use any object of the derived type as if it were of the base type, through pointers or references.
- The derived class constructor needs to delegate base member initialization to a base type constructor in the initializer list.
- The base class is always initialized first.
- If a base class has a static member, there is only once instance of this type for the entire hierarchy.
- The declaration of a derived class should not contain the derivation list.
- A class must be defined, not just declared, before it can be used as a base class.
- We can prevent inheritance from a class by declaring it as `class Derived final : Base`.
- The static type is the one known to us at compile time.
- The dynamic type of an expression that is neither a reference nor a pointer is the exact same as its static type.
- There is no automatic conversion from base to derived.
- Usually we have references as the parameters in copy constructors and in assignment operators. This lets us exploit them in derived to base conversions implicitly.
- It is import to note that the copied object has been sliced down to the Base type.
- We must always define a virtual function, irrespective of wether it is actually used.
- If a virtual function is called thorugh an object (instead of a reference or pointer), then it is determined at compile time.
- A derived class function overriding a base function must have the exact same signature.

- The parameter types need to be the exact same, however the return types may be related through inheritance.
- To avoid mistakes like these where the derived class ends up defining a different function instead of overriding the one from the base class, make you intent clear in the derived class using `override` keyword at the end of the function declaration (after any const or reference qualifiers).
- We can also designate a function as `final` in the same syntax, and overriding a final function is an error.
- Virtual functions can have default arguments. When summoned from a pointer/reference to base class, the defaults specified in the base class declaration(s) are used.
- To call a specific version of a function, circumventing virtualness, we can use the scope operator, `Base::func(...)`.
- This might be useful if the derived class only needs to do a little more work than the base version of a function.
- Some base classes should not have any instantiations, these are called abstract base classes.
- This can be done by defining the offending function to be purely virtual. This done by setting its prototype to zero on the declaration inside the class body, `... = 0;`.
- Note that this does not mean you should not define constructors or stuff, the derived class would still need that.
- A derived class constructor should only delegate to its direct base class constructor, not anything up in the hierarchy.
- Protected members can be accessed thorugh derived classes. Also, friends and other such functions/classes can only access the stuff through the derived class.
- The type of inheritance of the derived class restricts access to the users of the derived class. Essentially it upperbounds the access in the order public > protected > private.
- Public inheritance also allows derived to base conversions in the user code.
- Protected inheritance allows children of the derived class to access this conversion, but not general users.
- Friendship is in no way inherited.
- Under protected or private inheritance, to exempt individual names to retain their original access modifiers in the base class (or change stuff altogether), we can add a using declaration:

```
access-modifier:
    using Base::type;
```

- The access modifer determines the new access level.
- Class has default private inheritance, struct has default public inheritance.
- Derived classes define a nested scope inside the base class scope.
- A derived class reusing a base class name essentially hides that name in the derived class scope.
- As usual, name lookup happens before type checking.

- If you need to override only some functions in an overloaded set, put a using declaration in the derived class definition. . You cannot specify any parameter list, just the function name.
- Defining the destructor as virtual lets us delete objects of derived type from pointers of base type. This will end up turning off synthesized move.
- The default constructor of a derived class first calls the default constructor of the direct base class, which continues up the hierarchy.
- Treating base class subobject as a member object of the derived object gives correct answers for the synthesized copy constructor and assignment operator in general.
- Base class makes derived destructor => move not synthesized in base => move not synthesized in derived. So define move in base.
- To define a derived copy constructor, delegate to the appropriate base class constructor. We might have to use `std::move` for that. For eg:

```
D(const D &d): Base(d) {...};
D(D &&d):Base(std::move(d)) {...};
```

- Similar stuff for assignment:

```
D &D::operator=(const D &rhs) {
    Base::operator=(rhs);
    ...
    return *this;
}
```

- No need to explicitly call base destructor, happens on its own.
- Calls to functions in constructor or destructor are to the type of the constructor/destructor itself.
- To inherit a constructor, write a line like `using Base::Base;` in the derived class definition.
- For each base constructor, the compiler generates a derived constructor of the form `derived(params):base(args){}`.
- The constructor using declaration does not change the access level of that constructor.
- A using declaration also cannot specifiy explicit or constexpr.
- If you have default arguments in the base, the derived gets multiple constructors, each successive one omitting one more parameter.
- The copy, move and default are not inherited.
- We can redefine some of these inherited constructors.

Templates

- A template definition starts with `template <...>`, within the brackets is the template parameters list.
- To instantiate a template function, the compiler tries to use the arguments of the call to deduce the template arguments for us. For eg:

```cpp
// template defn
template <typename T> int compare(const T &v1, const T &v2) {
    ...
}
compare(1, 0); // T is int
compare(ivec1, ivec2); // T is vector<int>
```

- The type parameter(s), usually referenced as `T`, can be used as a normal type within the function, and can also define the paramter or return type.
- One may use `class` instead of `typename` also.
- Other kinds of parameters to templates can be nontype, but these still need to be calculable at compile time (hence `constexpr`).
- The template nontype parameter acts as a constant value inside the template definition.
- The template part of the function must come even before the inline or constexpr qualifiers.
- Compiler generates code from a template only when it is instantiated.
- To generate an instantiation, the compmiler needs the complete code. Usually, this means template functions need to be completely added in the headers.
- A class template is ablueprint for generating classes.
- Compiler cannot generally deduce types of the class template type parameters.
- We need to explicitly provide the template arguments, eg: `vector<int>`.
- A member function of a template class defined outside needs to again be templatized. For eg:

```cpp
template <typename T> ret_type Class<T>::func_name(params)
```

- Constructors will look like `template <typename T> Class<T>::Class(params): ... {}`.
- A member function is instantiated only if the code uses it. This lets you instantiate some classes which may not really be correct.
- Inside the template class definition, you can use the name of the class without arguments.
- Class templates can control which classes to make friends. Nontemplated friend declarations give the friend access to all instantiations.
- The following gives one-to-one friendship:

```cpp
template <typename T> Class {
    friend class OtherClass<T>;
    friend ret_type func<T>(params<T>);
};
```

- For the above to work, the template declaration for `OtherClass` must already be present above.
- To allow cross friendships and access, use:

```cpp
template <typename T> Class {
```

```
    template <typename X> friend class OtherClass<X>; // the <X> is not needed
};
```

- We can also make `T` itself a friend, `friend T`.
- We can also define template type alias:

```
template<typename T> using twin<T> = pair<T, T>;
```

- For static member, each instantiation has its own copy of the static member.
- We need to define this similar to a member function template:

```
temaplte <typename T> static_type Class<T>::member_name = initializer;
```

- A static member is instantiated only if it is used (just like the functions).
- We cannot reuse the typename within the declaration, as in `template <typename U, typename U>` is illegal.
- Inside the templated class, the compiler does not know if `T::mem` defines a type or a static data member. By default it is assumed not to be a type.
- If it is intended to be a type, explicitly say this as in `template <typename T> typename T::mem ....`
- We can also provide default arguments to the template arguments, like `template <typename T, typename F = less<T>>`. To use all defaults in a class template, use `className<>`.
- When defining template functions inside template classes outside the class, use two different `template <typename>` sets, first the class and then the function.
- To avoid the overhead of multiple instantiations, we use `extern template ...` and `template ...` where `...` looks like `ret_type func_name(params)` or `class className<type>`.
- When extern is used, we ask the compiler to trust us and a nonextern usage is there somwhere in the final code.
- The instantiation definition ends up defining all member functions and static members and stuff.
- MORE STUFF COMING UP