



Challenge Mode:

```
const int* const myClassMethod(const int* const & param) const;
```

This function takes
in a const pointer...

This function returns
a const pointer...

...to a const int.

And this is a const
member function,
i.e. this function
can't modify any
variables of the
this instance

...to a const int.



When to use each?

<https://stackoverflow.com/questions/15999123/const-before-parameter-vs-const-after-function-name-c>



Namespaces

大白话：include导入库；using namespace告诉complier用库里面的那些函数；
类似python：from xx import xx; xx.function

*I'm uh speaking so fast I'm trying to
make up some lost time yes*



Namespaces (C++)

Most modern languages use namespaces to fix this

```
# Count how many times value appears in C++  
#include <algorithm>  
  
std::count(v.begin(), v.end(), 1);
```

ID: 558-124-450

Stop Share



Inheritance



Interface

In Java:

```
interface Drink {  
    public void make();  
}
```

```
class Tea implements Drink {  
    public void make() {  
        // implementation  
    }  
}
```

In C++:

```
class Drink {  
public:  
    virtual void make() = 0;  
};
```

Called a **pure virtual function**,
denoted by the **= 0**.

Means that the inheriting class
must define that function.



Interfaces

There is no interface keyword in C++!

要构interface: 全用pure virtual function

- To be an interface, a class must consist only of pure virtual functions
 - What if we do want to define some functions in our class?
- To implement an interface, a class must define all of those virtual functions



Abstract Classes

只要有一个pure virtual function, 就是一个abstract class, 就不能实例化

If a class has at least one pure virtual function, then it's called an abstract class. (Interfaces are a subset of abstract classes.)

Abstract classes cannot be instantiated.

```
class Base {  
public:  
    virtual void foo() = 0; // pure virtual function  
    virtual void foo2(); // non-pure virtual function  
    void bar() = { return 42; }; // regular function  
};
```

在derived class中overload virtual function, 需在public里面再次声明函数, 但无需virtual 关键词



Aside: Inherited Members

No “virtual” members - instead, if a member has the same name as an inherited member, it **hides** it

```
struct A {  
    int a;  
};  
  
struct B : public A {  
    double a; // Hides A::a  
};
```



Template Classes and Concepts

Bad Dad Joke of the Day:

- Got food poisoning?
- It could B. cereus.

Creds: PV



Terminology

Base (aka **superclass** or **parent**) class:

- the class that gets inherited from

Derived (aka **subclass** or **child**) class:

- the class that inherits from the base class



Constructors

Always calls the superclass constructor.

If not specified, calls the default constructor of Base.

```
class Derived : public Base {  
    Derived() : Base(args), /*others*/ {  
        // rest of constructor  
    }  
};
```

style guide: 一般在derived class里面的constructor要在initialization list里面先调用父类里面的constructor，再做一些个性化话的操作



Destructors

If you intend to make your class inheritable (i.e. if your class has any virtual functions), make your destructor virtual!

```
virtual ~Base() {}
```

Otherwise will almost definitely have memory leaks.

Aside: Whether or not the destructor is virtual is a good indication of whether a class is intended to be inherited from.

一个inheritable的类的destructor一定要是virtual的；（标志）



Terminology: Access Specifiers

Fancy name for three words you've seen a lot:

- private
 - Can only be accessed by **this** class
- protected
 - Can only be accessed by **this** class or **derived classes**
- public
 - Can be accessed by **anyone**



Terminology: Access Specifiers

```
class Drink {  
public:  
    void foo(); } Accessible by Rock  
protected:  
    void bar(); } Accessible by Tea  
private:  
    void baz(); } Accessible by Drink  
};
```

Derived classes can access `foo` and `bar`. Unrelated classes can only access `foo`.

The screenshot shows the Qt Creator IDE interface with the following details:

- Projects View:** Shows the project structure with "SimpleInheritance" selected. Inside "SimpleInheritance", there are "Sources" containing "main.cpp". Other projects like "StringVector" and "TemplateClasses" are also listed.
- Code Editor:** Displays the file "SimpleInheritance/main.cpp*". The code is as follows:

```
using std::cout;      using std::endl;

class Drink {          'Drink' has no out-of-line virtual method d...
public:
    Drink() = default;
    Drink(std::string flavor) : flavor(flavor) {}

    virtual void make() = 0;
    virtual ~Drink() = default;
private:
    std::string flavor;
};

class Tea : public Drink {      'Tea' has no out-of-line virtu...
public:
    Tea() = default;

    Tea(std::string flavor) : Drink(flavor) {}

    virtual ~Tea() = default;

    void make () {
        cout << "Made tea from the Tea class!" << endl;
        // go get a cup
        // pour tea in
    }
}
```

- Toolbars and Status Bar:** The top bar includes standard file operations (New, Open, Save, etc.). The bottom status bar shows tabs for "Issues" (7), "Search Results", "Application Outp...", "Compile Output", "QML Debugger ...", "General Messages", and "Test Results".
- Top Right Corner:** A tooltip displays the message "'Drink' has no out-of-line virtual method d..." over the class definition.



Templates vs. Derived Classes

Really, static vs. dynamic polymorphism



When to use each?

template: static polymorphism (make copies at compile time)

inheritance: dynamic polymorphism

Prefer templates when:

- Runtime efficiency is most important
- No common base can be defined

Prefer derived classes when:

- Compile-time efficiency is most important
- Want to hide implementations
- Don't want code bloat



Side Note: Casting

All of these are legal:

```
int a = (int)b;  
int a = int(b);  
int a = static_cast<int>(b);
```

Using `static_cast` is considered best practice (although many still use old style casts)



Template Classes



Templates Review

A **function template** describes how to build a family of similar-looking functions.

A **class template** describes how to build a family of similar-looking classes.

Definition

Template Class: A class that is parametrized over some number of types. A class that is comprised of member variables of a general type/types.

Writing a Template Class: Syntax

//mypair.h

```
template<typename First, typename Second> class MyPair {  
public:  
    First getFirst();  
    Second getSecond();  
  
    void setFirst(First f);  
    void setSecond(Second f);  
private:  
    First first;  
    Second second;  
};
```

Generic! First and Second can
be any type (including
custom types) not just ints 😊

Use generic typenames as placeholders!

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

template<class First, typename Second>
First MyPair<First, Second>::getFirst() {
    return first;
}
// Fixed!
```

Implementing a Template Class: Syntax

Template Function

class and typename interchangeable

List of template variables

```
template<class First, typename Second>
First MyPair<First, Second>::getFirst() {
    return first;
}
// Fixed!
```

Member Types

- Sometimes, we need a name for a type that is dependent on our template types
- Recall: iterators

```
std::vector<int> a = {1, 2};  
std::vector<int>::iterator it = a.begin();
```

- iterator is a **member type** of vector

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    public:
        using iterator = T* // something internal like T*
        iterator begin();
}
```

```
//vector.cpp
template <typename T>
typename vector<T>::iterator vector<T>::begin() { ... }
```

Member Types: Summary

- Used to make sure your clients have a standardized way to access important types.
- Lives in your namespace: **vector<T>::iterator**.
- After class specifier, you can use the alias directly (e.g. inside function arguments, inside function body).
- Before class specifier, use **typename**.



Concepts (C++20 only)



Constraints: requirements on the template arguments

```
template <typename It, typename Type>
    requires Input_Iterator<It> && Iterator_of<It> &&
        Equality_comparable<Value_type<It>, Type>
int countOccurrences(It begin, It end, Type val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```



A concept is a predicate, evaluated at compile-time, that is a part of the interface.

```
template <typename It, typename Type>
    requires Input_Iterator<It> && Iterator_of<It> &&
        Equality_comparable<Value_type<It>, Type>
int countOccurrences(It begin, It end, Type val);
```

The client can easily see the concepts `It` and `Type` must satisfy.



C++20 Concepts and Constraints

Can be used with **class templates, function templates, and non-template functions** (typically members of class templates).

The Standard library has concepts you can use, or we can define our own!



(C++20) STL Concepts

```
template<class T, class U>           concept Same;
template<class Derived, class Base>  concept DerivedFrom;
template<class From, class To>       concept ConvertibleTo;
template<class T, class U>           concept CommonReference;
template<class T, class U>           concept Common;
template<class T>                   concept Integral;
template<class T>                   concept SignedIntegral;
template<class T>                   concept UnsignedIntegral;
template<class LHS, class RHS>      concept Assignable;
template<class T>                   concept Swappable;
template<class T, class U>           concept SwappableWith;
template<class T>                   concept Destructible;
template<class T, class... Args>     concept Constructible;
template<class T>                   concept DefaultConstructible;
template<class T>                   concept MoveConstructible;
template<class T>                   concept CopyConstructible;
```

... and many more!



Defining Your Own Concept

```
template<class Derived, class Base>
concept DerivedFrom =
    is_base_of_v<Base, Derived> &&
    is_convertible_v<const volatile Derived*, const volatile Base*>;
```

```
template<class D, class B>
void f(T) requires DerivedFrom<D, B> {
}
```