Challenge Mode:

This function takes in a const pointer...

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

This function returns a const pointer...

...to a const int.

...to a const int.

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

When to use each?

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

ID: 558-124-450 Stop Share

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

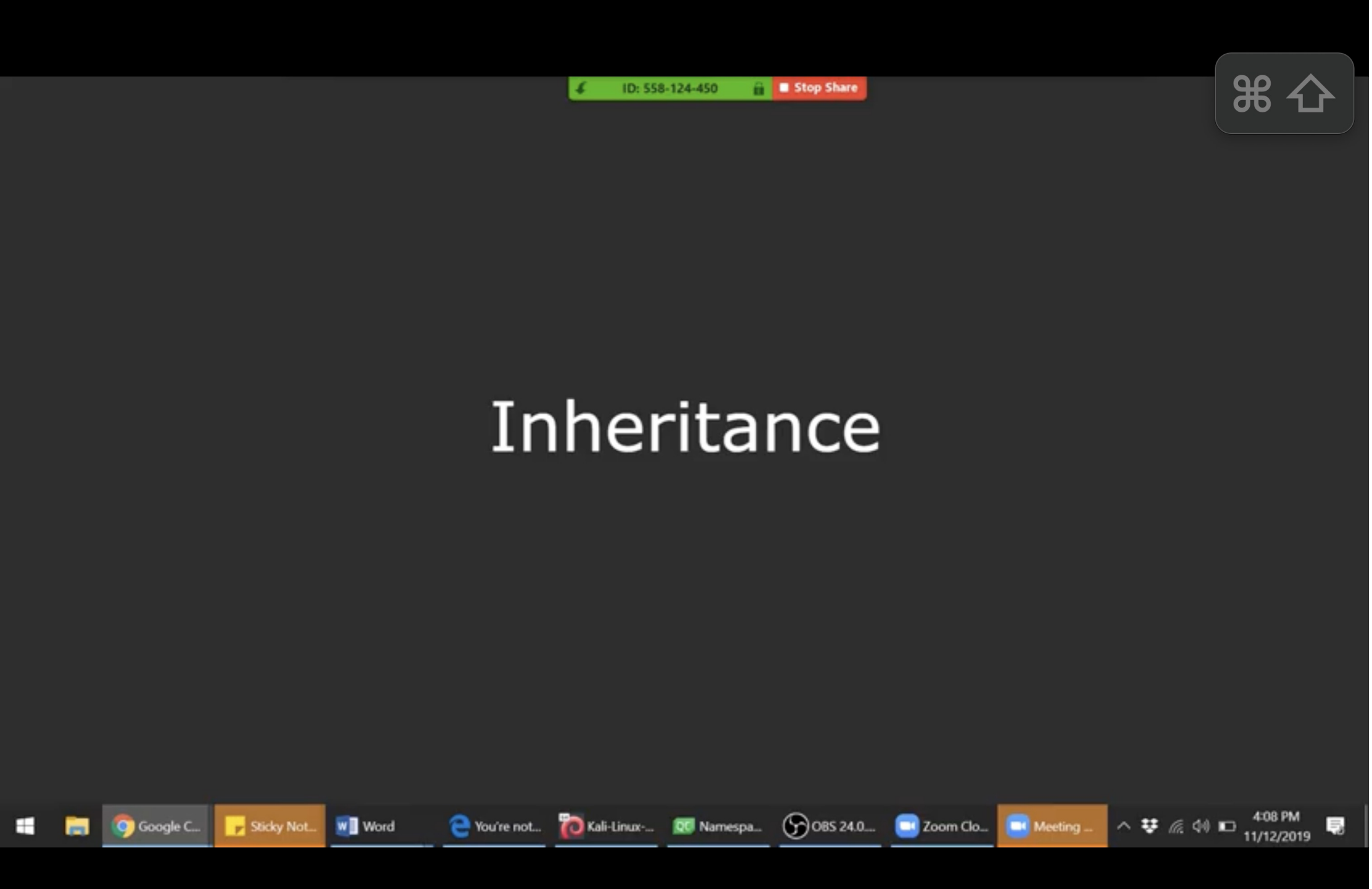
I'm uh speaking so fast I'm trying to make up some lost time yes

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
Google C...
Sticky Not...
Word
You're not...
Kali-Linux-...
Namespa...
OBS 24.0....
Zoom Clo...
Meeting ...
4:08 PM
11/12/2019

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

CS 106L Fall 2019 Avery Wang and Anna Zeng

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要来initialization list里面先调用父类里面的constructor，再做一些个性话的操作

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

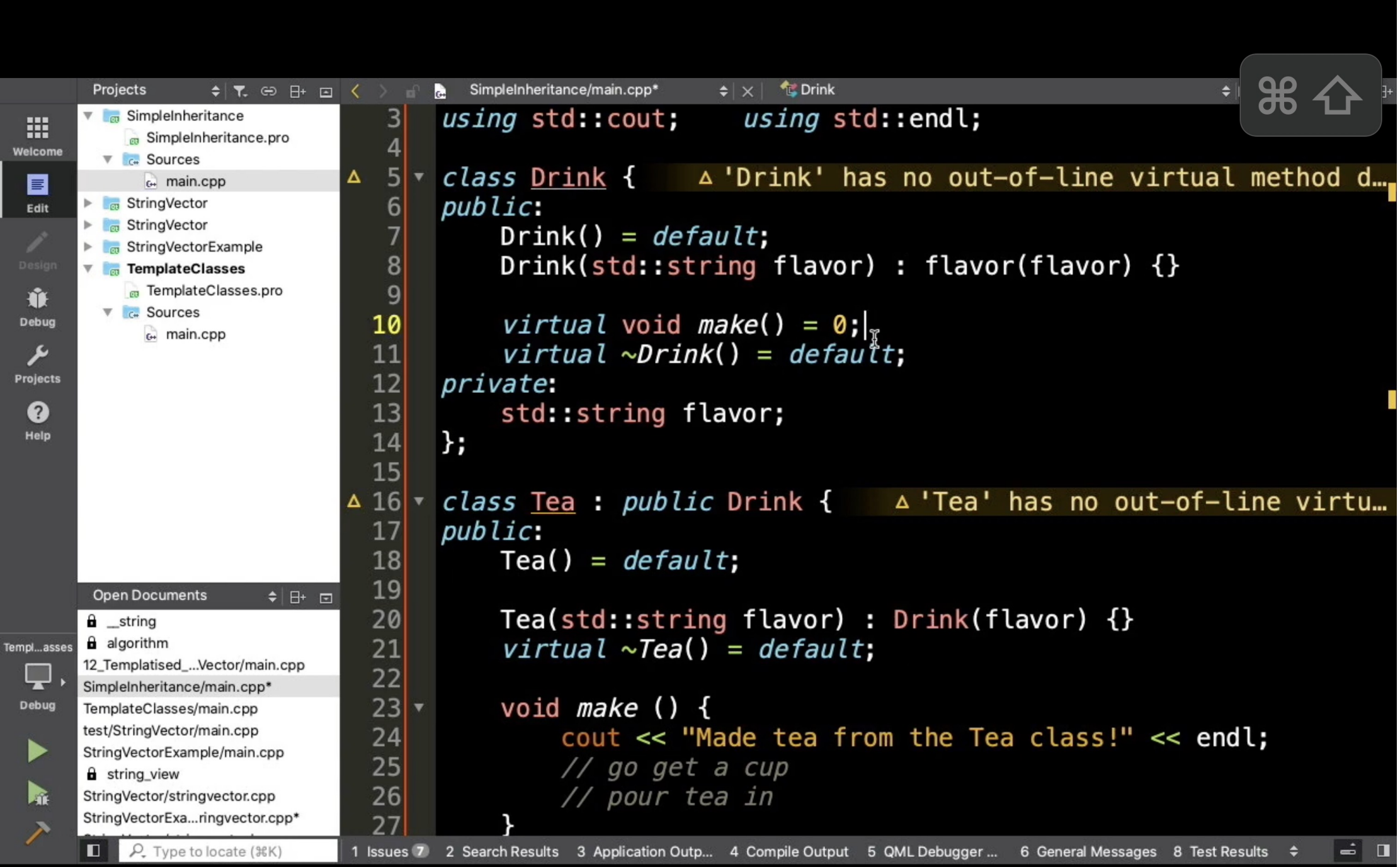
```
class Drink {
public:
   void foo();
protected:
   void bar();
private:
   void baz();
};
```

Accessible by Drink

Accessible by Tea

Accessible by Rock

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


Projects
SimpleInheritance
SimpleInheritance.pro
Sources
main.cpp
StringVector
StringVector
StringVectorExample
TemplateClasses
TemplateClasses.pro
Sources
main.cpp
Open Documents
__string
algorithm
12_Templatised_...Vector/main.cpp
SimpleInheritance/main.cpp*
TemplateClasses/main.cpp
test/StringVector/main.cpp
StringVectorExample/main.cpp
string_view
StringVector/stringvector.cpp
StringVectorExa...ringvector.cpp*
Welcome
Edit
Design
Debug
Projects
Help
Templ...asses
Debug
SimpleInheritance/main.cpp*
Drink
3 using std::cout; using std::endl;
4
5 class Drink { 'Drink' has no out-of-line virtual method d…
6 public:
7 Drink() = default;
8 Drink(std::string flavor) : flavor(flavor) {}
9
10 virtual void make() = 0;
11 virtual ~Drink() = default;
12 private:
13 std::string flavor;
14 };
15
16 class Tea : public Drink { 'Tea' has no out-of-line virtu…
17 public:
18 Tea() = default;
19
20 Tea(std::string flavor) : Drink(flavor) {}
21 virtual ~Tea() = default;
22
23 void make () {
24 cout << "Made tea from the Tea class!" << endl;
25 // go get a cup
26 // pour tea in
27 }
Type to locate (⌘K)
1 Issues 7
2 Search Results
3 Application Outp...
4 Compile Output
5 QML Debugger ...
6 General Messages
8 Test Results

polymorphism: treat diff types with same code

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:

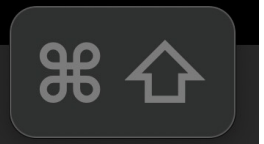
- Compile-time efficiency is most important
- Want to hide implementations
- Don’t want code bloat

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.

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 countOccurences(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 countOccurences(It begin, It end, Type val);
```

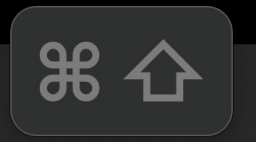
The client can easily see the concepts It and Type must satisfy.

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!

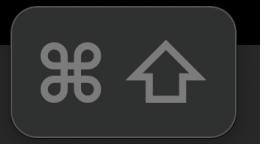
(C++20) STL Concepts



```
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> {

}
```