

C++ Notes

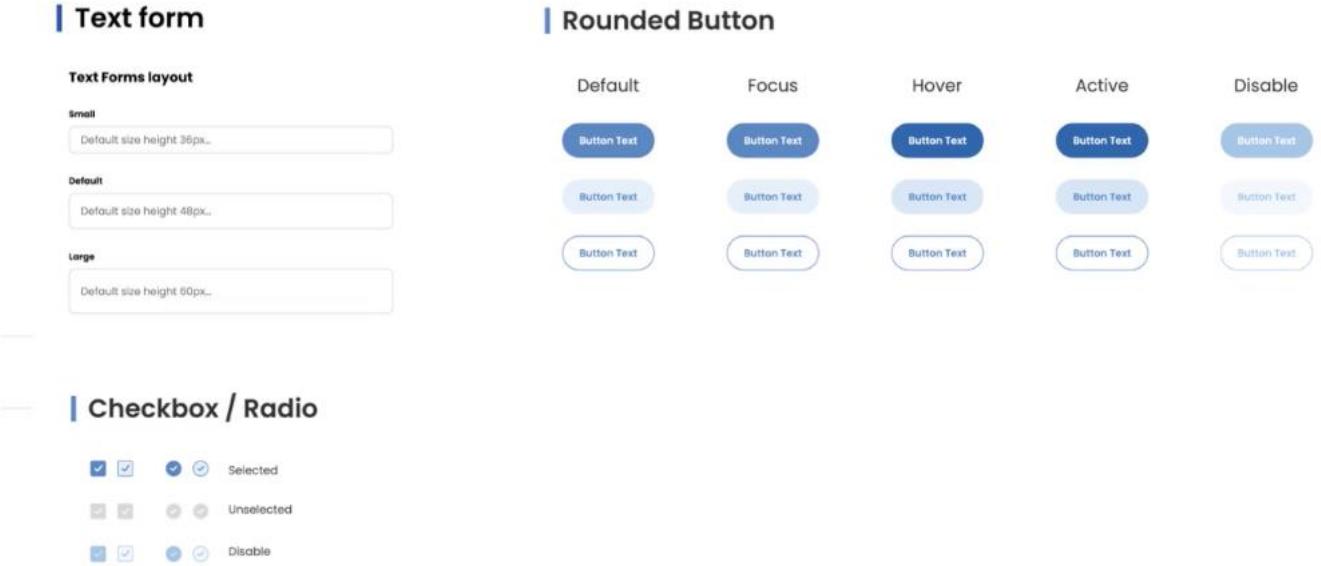
Part 3-2

**Slides were created from
CodewithMosh.com**

Inheritance and Polymorphism

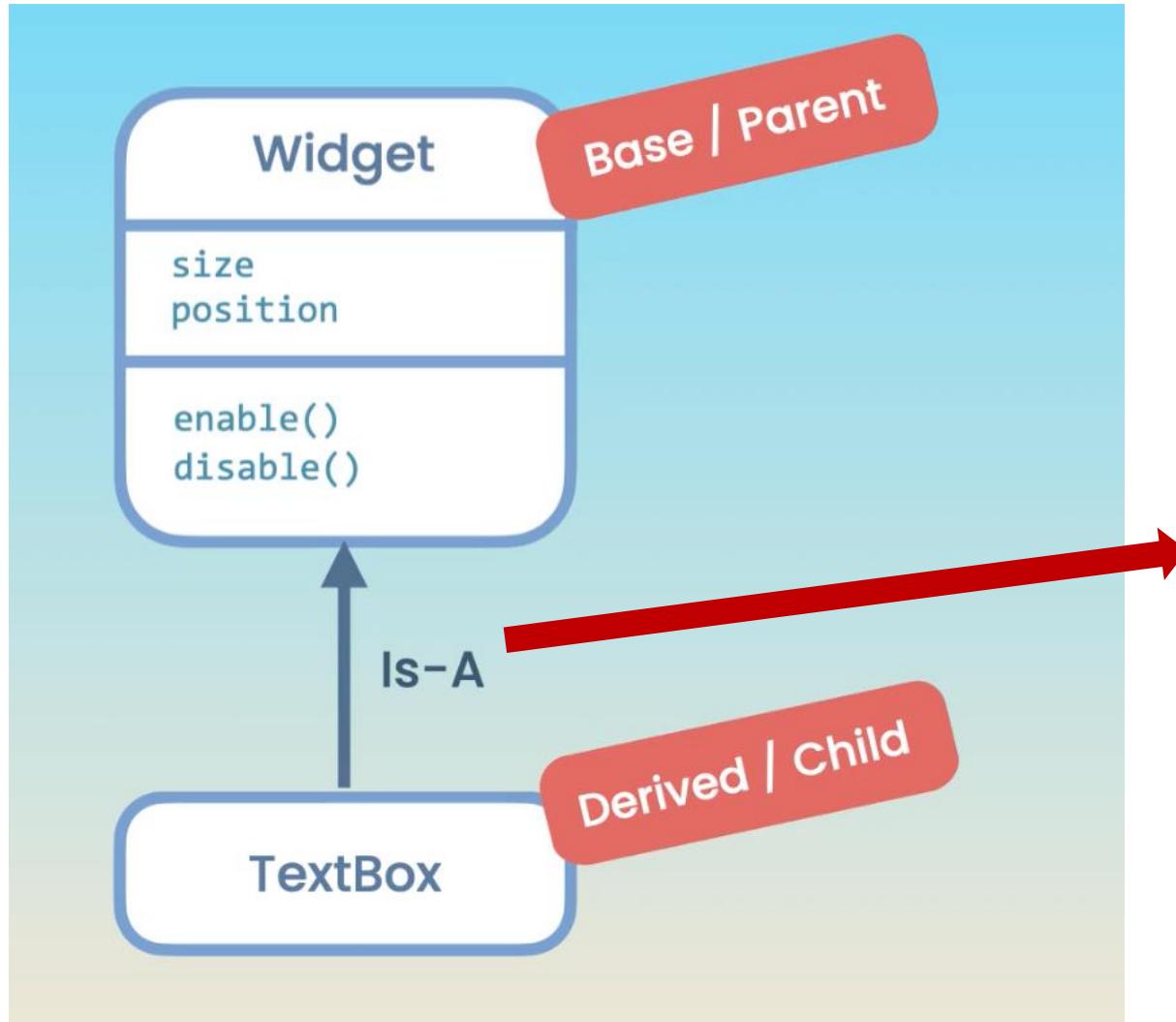
- Protected members
- Virtual methods
- Method overriding
- Abstract classes
- Final classes

Inheritance



- ❑ Let's suppose we want to build a framework to create GUI with different components.
- ❑ All these components have some common features.
- ❑ We do not want to repeat these features for each component.
- ❑ This is where we use inheritance.

Inheritance



- ❑ Inheritance is a mechanism for reusing code.
- ❑ Represents the inheritance relationship.

Inheritance

Widget.cpp X Widget.h X

```
4  
5     #include "Widget.h"  
6  
7     void Widget::enable() {  
8         enabled = true;  
9     }  
10    void Widget::disable() {  
11        enabled = false;  
12    }  
13  
14    bool Widget::isEnabled() const {  
15        return enabled;  
16    }  
17  
18
```

Widget.cpp X Widget.h X

```
7  
8  
9     class Widget {  
10        public:  
11            void enable();  
12            void disable();  
13            bool isEnabled() const;  
14        private:  
15            bool enabled;  
16    };  
17
```



Widget class header and implementation.

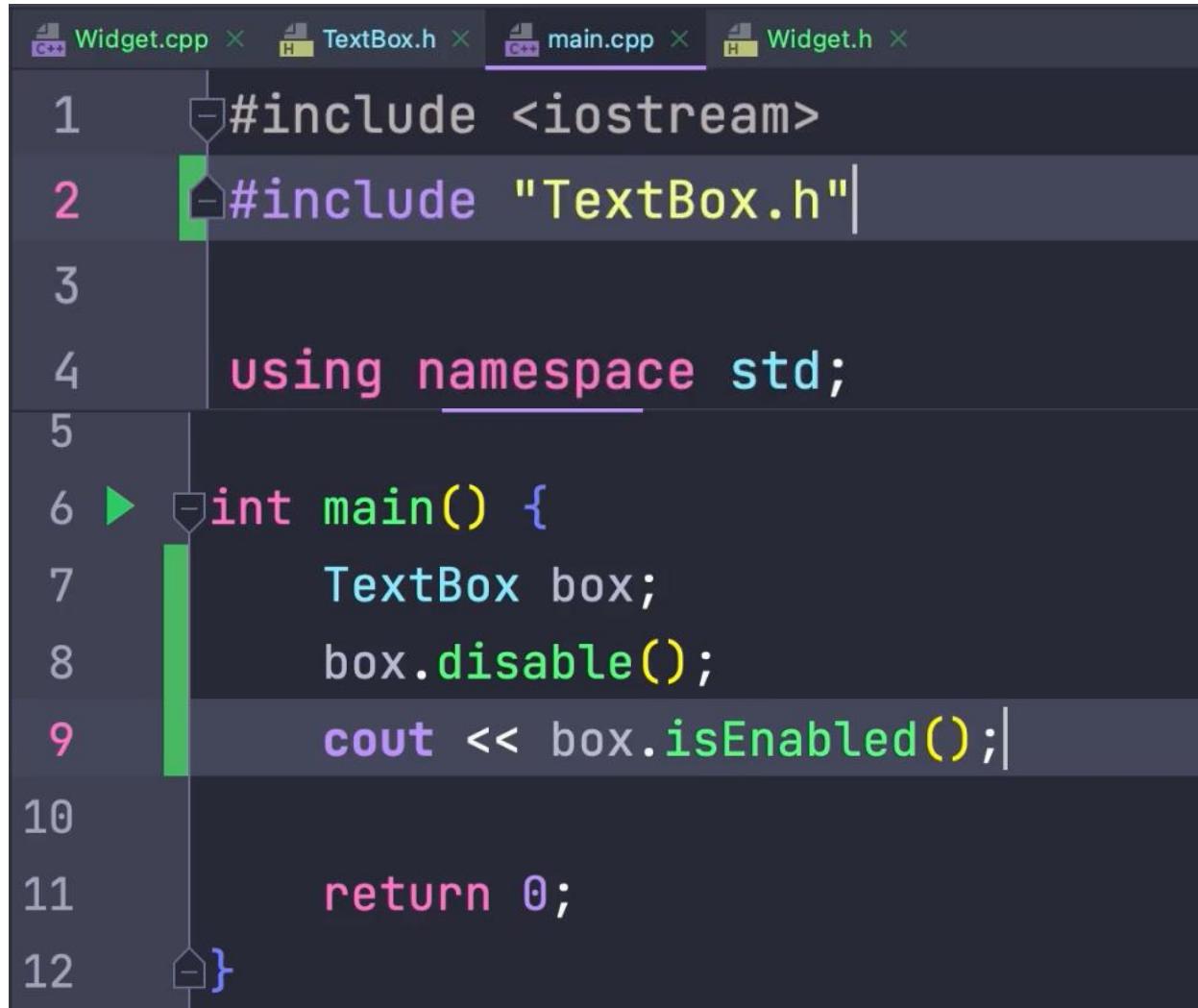
Inheritance

- All the public members of the inherited class are also public.
- If we use the access modifier “**private**”, then the public members of the base class will be private in the inheriting class.
- They will not be accessible outside of the class.

```
Widget.cpp × TextBox.h × Widget.h ×
7
8 #include <string>
9 #include "Widget.h"
10
11 using namespace std;
12
13 class TextBox : public Widget {
14     public:
15         TextBox() = default;
16         explicit TextBox(const string& value);
17         string getValue();
18         void setValue(const string& value);
19     private:
20         string value;
21 };
22
```

```
Widget.cpp × Widget.h ×
7
8
9 class Widget {
10     public:
11         void enable();
12         void disable();
13         bool isEnabled() const;
14     private:
15         bool enabled;
16 }
```

Inheritance



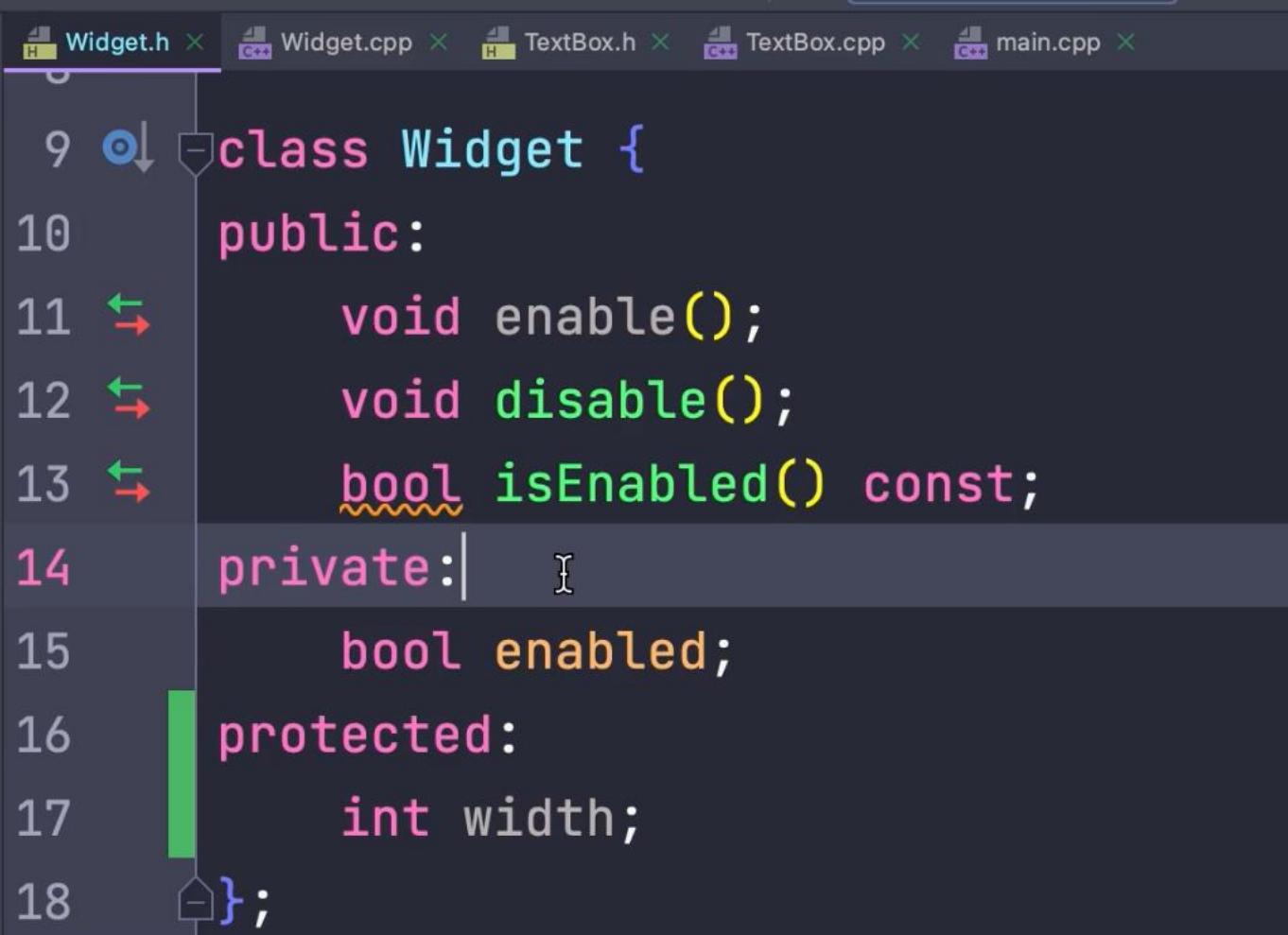
A screenshot of a code editor showing the `main.cpp` file. The file contains the following C++ code:

```
1 #include <iostream>
2 #include "TextBox.h"
3
4 using namespace std;
5
6 int main() {
7     TextBox box;
8     box.disable();
9     cout << box.isEnabled();
10
11    return 0;
12 }
```

The code includes headers for `<iostream>` and `TextBox.h`, uses the `std` namespace, and defines a `main` function that creates a `TextBox` object, disables it, and then checks its enabled state using `isEnabled()`.

- Using inheritance, we can create new classes by reusing and expanding on existing classes.

Protected Members

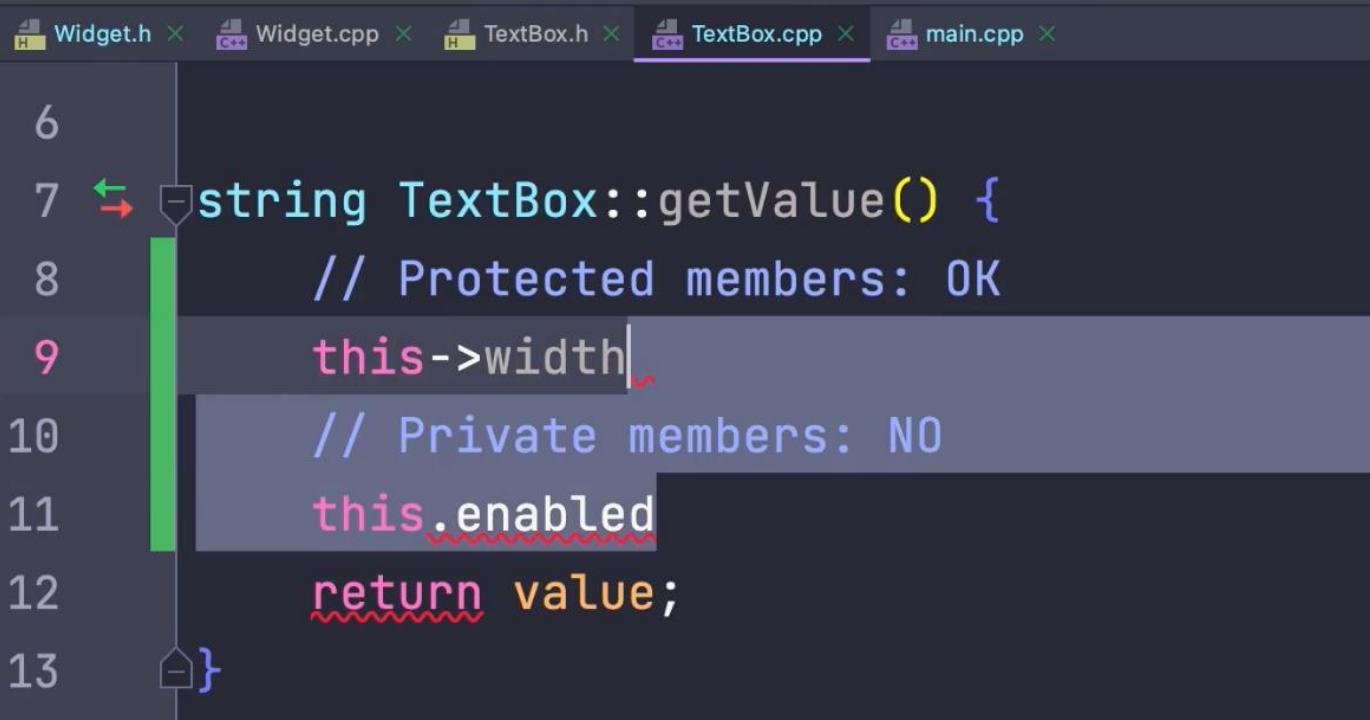


A screenshot of a code editor showing the `Widget.h` header file. The code defines a class `Widget` with three sections: `public`, `private`, and `protected`. The `public` section contains methods `enable()`, `disable()`, and `isEnabled() const`. The `private` section contains a member variable `enabled`. The `protected` section contains a member variable `width`. The code editor interface shows tabs for `Widget.h`, `Widget.cpp`, `TextBox.h`, `TextBox.cpp`, and `main.cpp`.

```
Widget.h
9  class Widget {
10 public:
11     void enable();
12     void disable();
13     bool isEnabled() const;
14 private:
15     bool enabled;
16 protected:
17     int width;
18 }
```

- Protected members are similar to private members.
- They cannot be accessed outside of the class.
- **Difference:** Protected members are accessible within the derived classes, but private members are not.

Protected Members

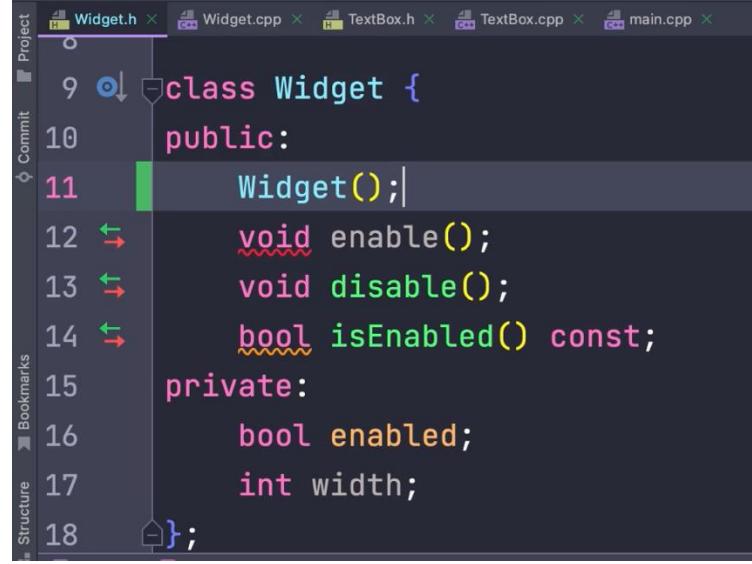


```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp
6
7     string TextBox::getValue() {
8         // Protected members: OK
9         this->width
10        // Private members: NO
11        this.enabled
12        return value;
13    }
```

- Protected members are similar to private members.
- They cannot be accessed outside of the class.
- **Difference:** Protected members are accessible within the derived classes, but private members are not.

Constructors and Inheritance

- In inheritance, base class constructor is called before the derived class constructor.



```
Widget.h
9 class Widget {
10 public:
11     Widget();
12     void enable();
13     void disable();
14     bool isEnabled() const;
15 private:
16     bool enabled;
17     int width;
18 };
```



```
Widget.cpp
4
5 #include <iostream>
6 #include "Widget.h"
7
8 using namespace std;
9
```



```
Widget.cpp
10     bool isEnabled() const {
11         return enabled;
12     }
13
14     Widget::Widget() {
15         cout << "Widget constructed" << endl;
16     }
17 }
```



```
main.cpp
6
7 int main() {
8     TextBox box;
9
10    return 0;
11 }
12
```

Constructors and Inheritance

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
13  class TextBox : public Widget {
14
15  public:
16      TextBox();
17      explicit TextBox(const string& value);
18      string getValue();
19      void setValue(const string& value);
20
21  private:
22      string value;
23};
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
4
5 #include <iostream>
6
7 #include "TextBox.h"
8
9 using namespace std;
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
10
11     . . .
12
13     const string& value, . . .
14
15 }
16
17
18
19 TextBox::TextBox() {
20     cout << "TextBox constructed" << endl;
21
22 }
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
6
7 int main() {
8     TextBox box;
9
10    return 0;
11}
12
```

```
/Users/moshfeghamedani/CLionProject
Widget constructed
TextBox constructed
Process finished with exit code 0
```

Constructors and Inheritance

- This will give a compilation error, because now the Widget class does not have a default constructor.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
9  o class Widget {
10 public:
11     • Widget(bool enabled);
12     void enable();
13     void disable();
14     bool isEnabled() const;
15 private:
16     bool enabled;
17     int width;
18 };
```

Updated constructor

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
21
22     • Widget::Widget(bool enabled) : enabled{enabled} {
23         cout << "Widget constructed" << endl;
24     }
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
18     • TextBox::TextBox(const string &value) : value{value} {
```

- TextBox constructor does not know what value to pass to the constructor of the base class (Widget).

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
6
7 ► - int main() {
8     • TextBox box;
9
10    return 0;
11 }
12 }
```

Constructors and Inheritance

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x

9 o class Widget {
10 public:
11     Widget() = default; ←
12     Widget(bool enabled);
13     void enable();
14     void disable();
15     bool isEnabled() const;
16 private:
17     bool enabled;
18     int width;
```

Solution 1:

- The simplest solution is to add a default constructor to the Widget class.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x

6
7 ► - int main() {
8     ● TextBox box;
9
10    return 0;
11 }
12 }
```

Constructors and Inheritance

Solution 2:

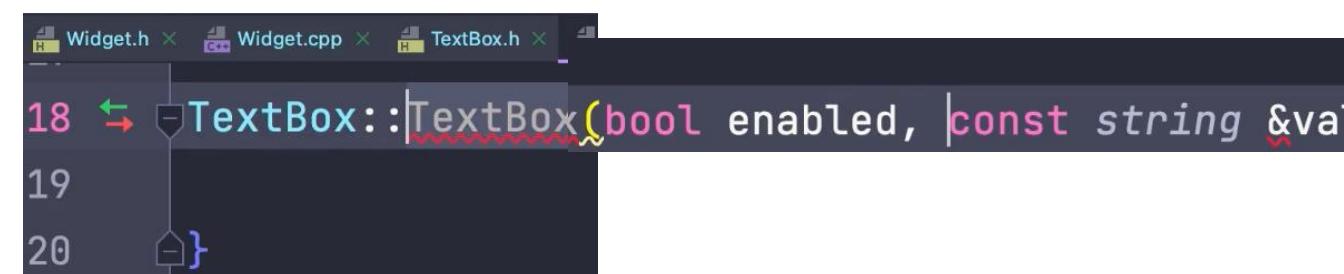
- Pass the parameter to the base class constructor from the derived class constructor.



```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp

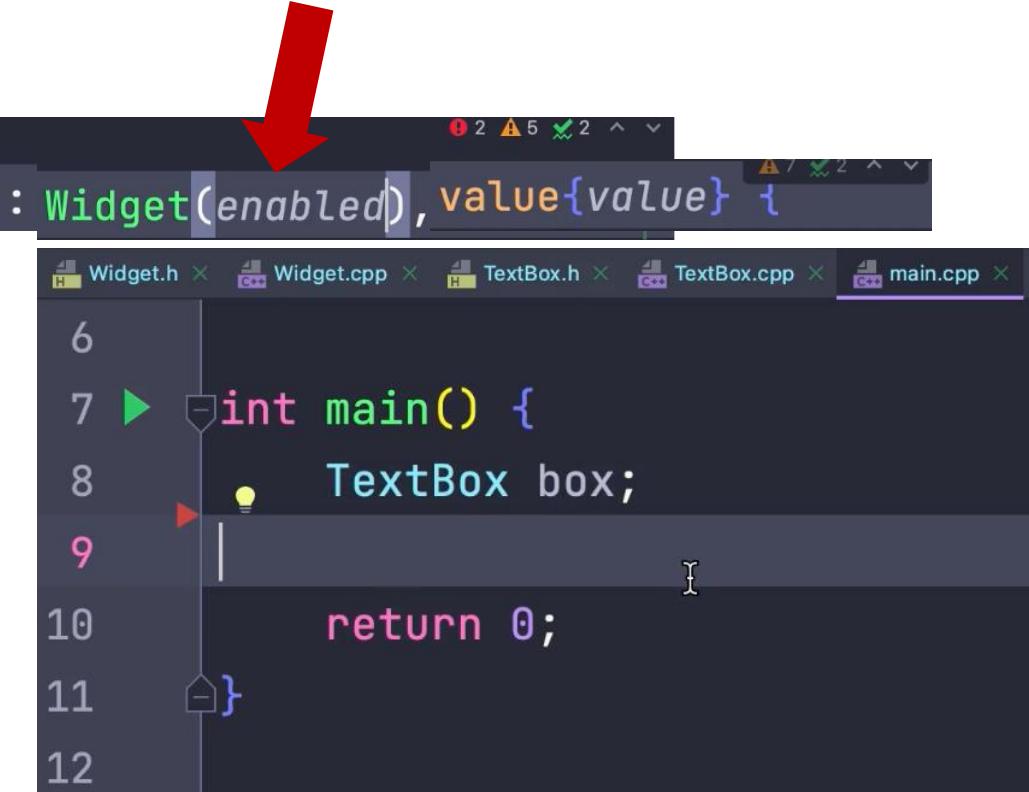
13 class TextBox : public Widget {
14     public:
15         TextBox();
16         explicit TextBox(bool enabled, const string& value);
17         string getValue();
18         void setValue(const string& value);
19     private:
20         string value;
21 };
```

A red arrow points from the underlined 'value' in the constructor declaration to the corresponding line in the constructor definition below.



```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp

18 TextBox::TextBox(bool enabled, const string &value) : Widget(enabled), value{value} {}
```

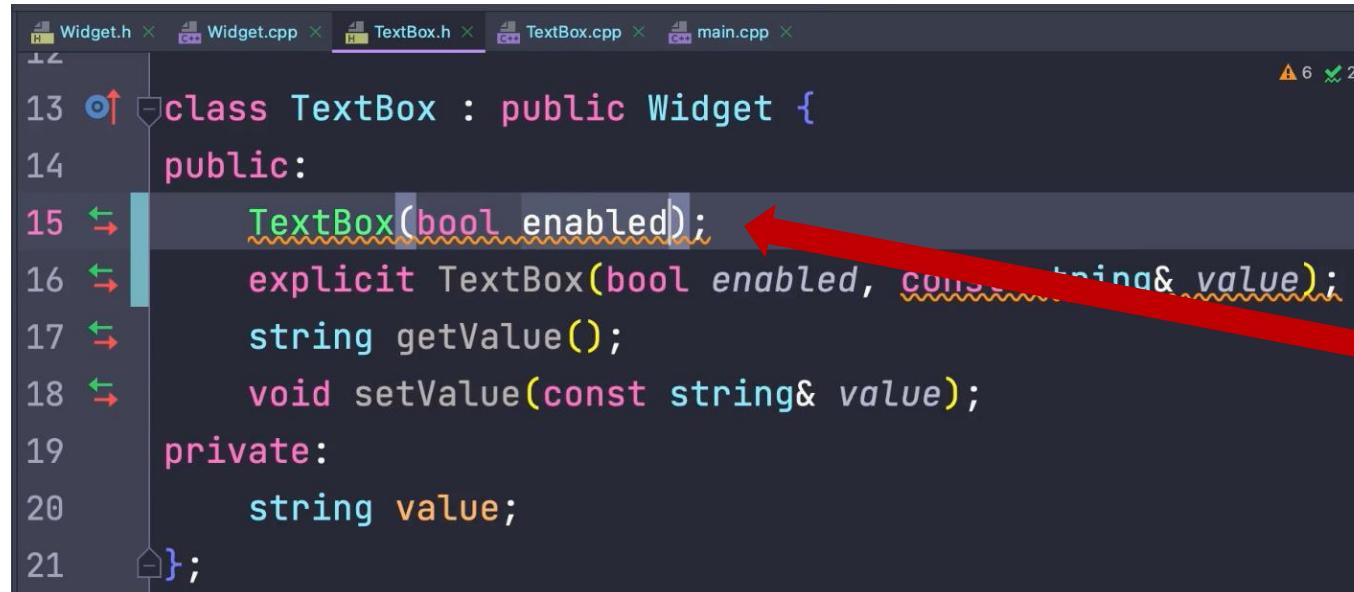


```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp

6
7 int main() {
8     TextBox box;
9
10    return 0;
11}
```

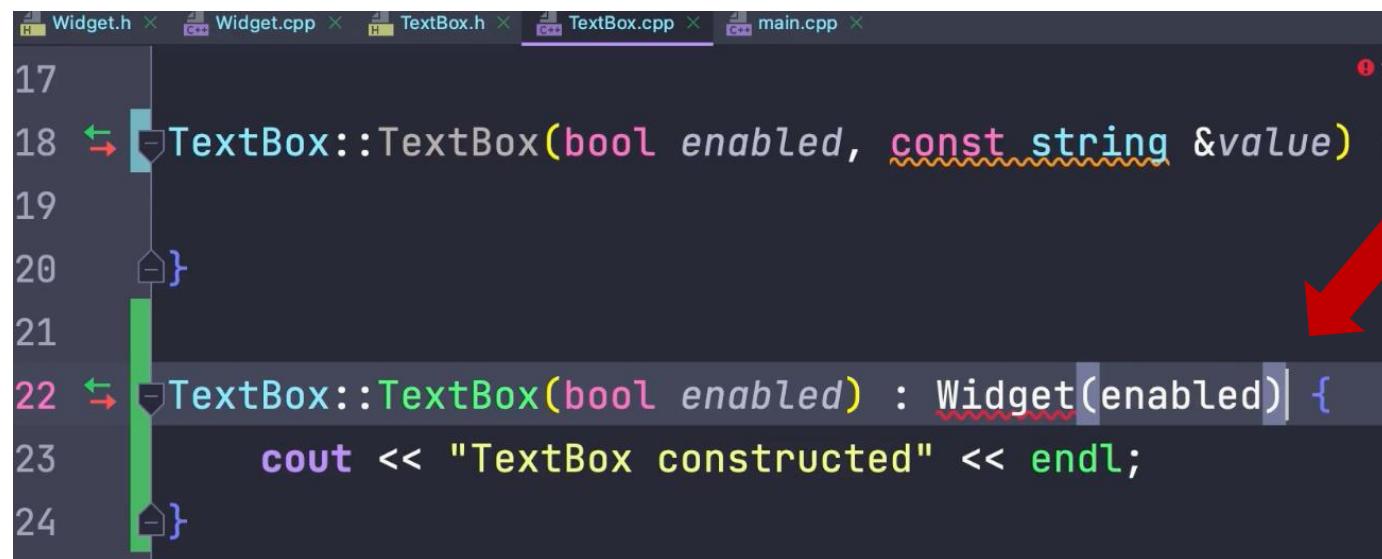
Constructors and Inheritance

- Update the base class to pass the parameter.



```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp

13 class TextBox : public Widget {
14     public:
15         TextBox(bool enabled); // Red arrow points here
16         explicit TextBox(bool enabled, const string& value);
17         string getValue();
18         void setValue(const string& value);
19     private:
20         string value;
21 };
```



```
Widget.h Widget.cpp TextBox.h TextBox.cpp main.cpp

17
18     TextBox::TextBox(bool enabled, const string &value) :
19
20     }
21
22     TextBox::TextBox(bool enabled) : Widget(enabled) {
23         cout << "TextBox constructed" << endl;
24 }
```

Constructors and Inheritance

- ❑ If a constructor is used just to pass a parameter to the base class, then we can replace it by inheriting the constructor of the base class.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
13 class TextBox : public Widget {
14     public:
15         TextBox(bool enabled);
16         explicit TextBox(bool enabled, const string& value);
17         string getValue();
18         void setValue(const string& value);
19     private:
20         string value;
21     };

```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
17
18     TextBox::TextBox(bool enabled, const string &value) : Widget(enabled, value) {}
19
20 }
21
22     TextBox::TextBox(bool enabled) : Widget(enabled) {}
23 }
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
13 class TextBox : public Widget {
14     public:
15         using Widget::Widget;
16         explicit TextBox(bool enabled, const string& value);
17         string getValue();
18         void setValue(const string& value);
19     private:
20         string value;
21     };

```

- ❑ Delete it. It is replaced with base class constructor using inheritance.

Destructors and Inheritance

- ❑ In inheritance, destructors are called in the reverse order.

The screenshot shows a code editor with multiple tabs at the top: Widget.h, Widget.cpp, TextBox.h, TextBox.cpp, and main.cpp. The Widget.h tab is active. The code displays the class definition for Widget:

```
9 class Widget {  
10     public:  
11         Widget(bool enabled);  
12         ~Widget();  
13         void enable();  
14         void disable();  
15         bool isEnabled() const;  
16     private:  
17         bool enabled;  
18         int width;
```

A blue circular icon with a white dot is positioned next to the opening brace of the class definition on line 9. A green vertical bar highlights the entire class body from line 12 to the final closing brace on line 18.



```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
22 → Widget... Widget(Widget) . ~Widget(Widget)
23     cout << "Widget constructed" << endl;
24 }
25
26 ← Widget::~Widget() {
27     cout << "Widget destructed" << endl;
28 }
```

Destructors and Inheritance

- In inheritance, destructors are called in reverse order.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
13 class TextBox : public Widget {
14     public:
15         using Widget::Widget;
16         explicit TextBox(bool enabled, const string& value);
17         ~TextBox();
18         string getValue();
19         void setValue(const string& value);
20     private:
21         string value;
22 };
```

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
19
20 }
21
22 TextBox::~TextBox() {
23     cout << "TextBox destructed" << endl;
24 }
```

Destructors and Inheritance

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
6
7 ► int main() {
8     TextBox box;
9
10    return 0;
11}
```

- There is a compilation error. We need to pass a boolean value to the object we are creating.
- This is not a good design. We should assume that all the widgets are enabled by default.
- We will remove all the constructors with the boolean parameter.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
6
7 ► int main() {
8     TextBox box{ enabled: true };
9
10    return 0;
11}
```

Destructors and Inheritance

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
9 class Widget {
10 public:
11     Widget(bool enabled);
12     ~Widget();
13     void enable();
14     void disable();
15     bool isEnabled() const;
16 private:
17     bool enabled;
18     int width;
```

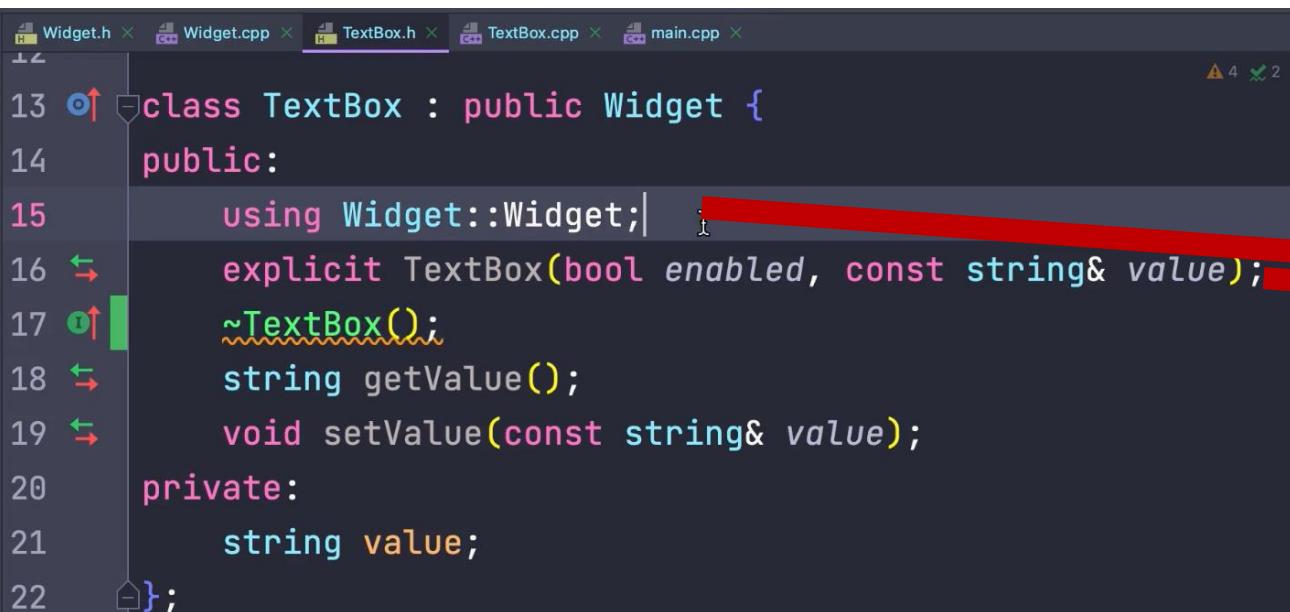
Remove the declaration from the Widget class header file.

```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
21
22 Widget::Widget(bool enabled) : enabled{enabled} {
23     cout << "Widget constructed" << endl;
24 }
25
26 Widget::~Widget() {
27     cout << "Widget destructed" << endl;
28 }
```

Remove the definition from the implementation file.

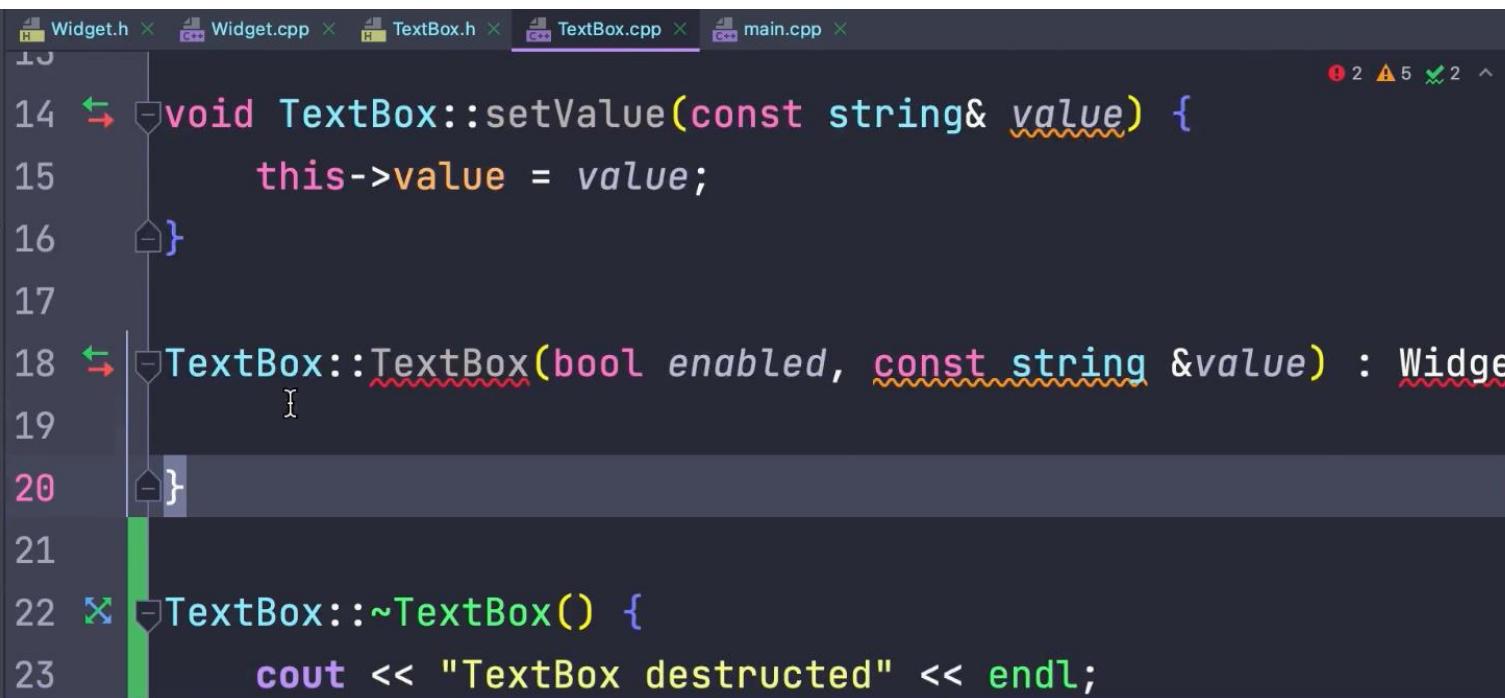
Destructors and Inheritance

Remove the inheritance and the other constructor.



```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
13 ⚡ class TextBox : public Widget {
14     public:
15         using Widget::Widget;
16         explicit TextBox(bool enabled, const string& value);
17     ~TextBox();
18         string getValue();
19         void setValue(const string& value);
20     private:
21         string value;
22 };
```

Remove the definition from the implementation file.



```
Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x main.cpp x
14 ⚡ void TextBox::setValue(const string& value) {
15     this->value = value;
16 }
18 ⚡ TextBox::TextBox(bool enabled, const string &value) : Widget(enabled)
19 {
20 }
22 ⚡ ~TextBox() {
23     cout << "TextBox destructed" << endl;
```

Destructors and Inheritance



```
Widget.h X Widget.cpp X TextBox.h X TextBox.cpp X main.cpp X
6
7 ► int main() {
8     TextBox box;
9
10    return 0;
11 }
```

- Remove the inheritance and the other constructor.
- In this design, since we are not using any system resources that need to be released, we do not need to explicitly create destructors. We can delete them.



```
Run: Advanced X
/Users/moshfeghhamedani/CLionProjects/Advanced/cmake-build-deb
TextBox destructed
Widget destructed
Process finished with exit code 0
```

Output of running the main function.

- TextBox object is destructed first, and Widget is destructed next.

Conversion Between Base and Derived Classes

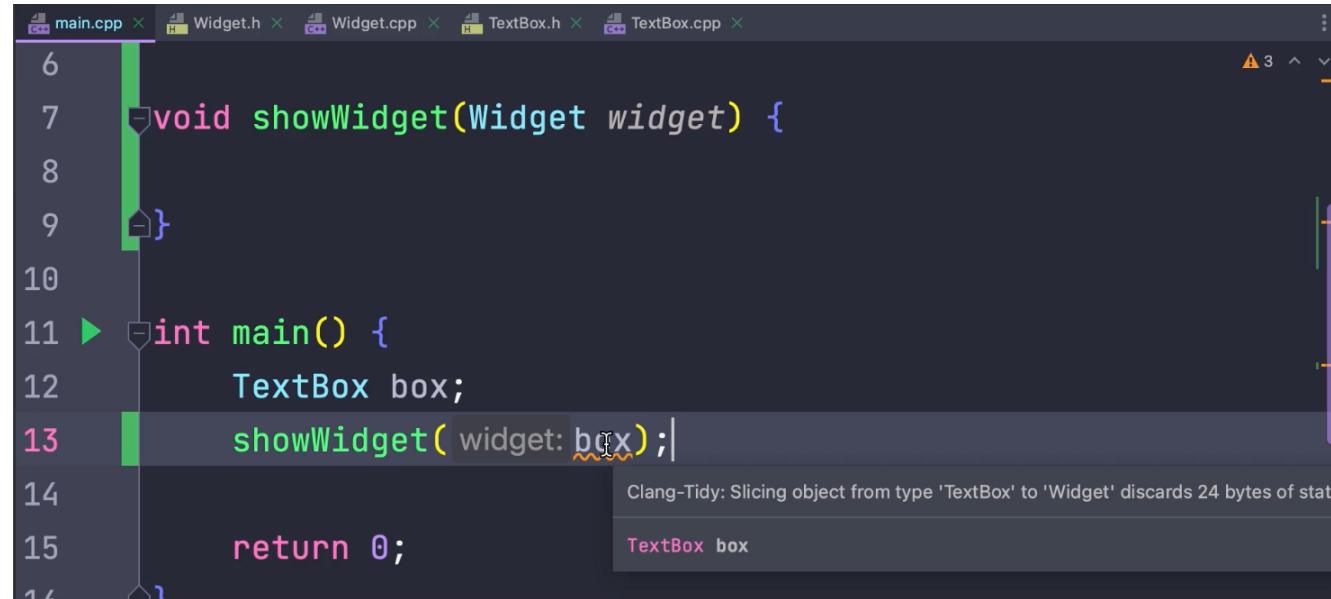
```
main.cpp X Widget.h X Widget.cpp X TextBox.h X TextBox.cpp X
6
7 > int main() {
8     TextBox box;
9     // Upcasting
10    Widget widget = box; 
11
12    return 0;
13 }
```

```
main.cpp X Widget.h X Widget.cpp X TextBox.h X TextBox.cpp X
6
7 > int main() {
8     Widget widget;
9     TextBox box = widget; 
10
11    return 0;
12 }
```

- TextBox is implicitly converted to a widget.
- It is called upcasting.

- Opposite of upcasting (downcasting) is illegal.
- TextBox class has additional members, and the compiler does not know how to add those additional members to the TextBox.

Conversion Between Base and Derived Classes



```
main.cpp Widget.h Widget.cpp TextBox.h TextBox.cpp
6
7     void showWidget(Widget widget) {
8
9 }
10
11 int main() {
12     TextBox box;
13     showWidget(widget: box);
14
15     return 0;
16 }
```

Clang-Tidy: Slicing object from type 'TextBox' to 'Widget' discards 24 bytes of state

- Box object is converted to a widget and passed to the showWidget function.
- This is pass by value.
- The private member “value” of TextBox is not copied to widget and discarded.
- It is called object slicing.



```
main.cpp Widget.h Widget.cpp TextBox.h TextBox.cpp
6
7     void showWidget(Widget& widget) {
8         widget.
9     }
10
11 int main() {
12     TextBox box;
13     showWidget(&box);
14
15     return 0;
16 }
```

- We can use pass-by-reference.
- Even though we are passing the address of the TextBox object to the function, the function cannot access to any members of TextBox.

Overriding Methods

```
main.cpp X Widget.h X Widget.cpp X TextBox.h X TextBox.cpp X
8
9 class Widget {
10 public:
11     void draw() const; // Method declaration
12     void enable();
13     void disable();
14     bool isEnabled() const;
15 private:
16     bool enabled;
17     int width;
18 }
```

```
main.cpp X Widget.h X Widget.cpp X TextBox.h X TextBox.cpp X
19     return enabled;
20 }
21
22 void Widget::draw() const { // Method definition
23     cout << "Drawing a Widget" << endl;
24 }
25
```

- Adding draw() function to Widget class.

Overriding Methods

- ☐ TextBox should have its own draw() function.
- ☐ Adding draw() function to TextBox class.

```
13 ⚡ class TextBox : public Widget {  
14     public:  
15         void draw() const;  
16         string getValue();  
17         void setValue(const string& value);  
18     private:  
19         string value;  
20     };
```

```
14 → void TextBox::setValue(const string& value)  
15             this->value = value;  
16     }  
17  
18 ✘ void TextBox::draw() const {  
19     cout << "Drawing a TextBox" << endl;  
20 }
```

```
main.cpp x Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x
6
7     void showWidget(Widget& widget) {
8         widget.draw();
9     }
10
11 int main() {
12     TextBox box;
13     showWidget(&box);
14
15     return 0;
16 }
```

```
Drawing a Widget
Process finished with exit code 0
```

Overriding Methods

- ❑ The compiler uses “static binding” (also called “early binding”) technique and calls draw() function of the Widget class.

- ❑ Screen output of main function.

Overriding Methods

```
8  
9  class Widget {  
10 public:  
11     virtual void draw() const; ←  
12     void enable();  
13     void disable();  
14     bool isEnabled() const;  
15 private:  
16     bool enabled;  
17     int width;  
18 };
```

```
6  
7  void showWidget(Widget& widget) {  
8      widget.draw(); ←  
9  }  
10  
11 int main() {  
12     TextBox box;  
13     showWidget(& box);  
14  
15     return 0;  
16 }
```

- If we want to use the draw() function of TextBox, then we use “virtual” keyword.
- The compiler uses “dynamic (late) binding” technique.
- The compiler decides which draw() method to use based on the object passed to the function.

- Draw() method of the TextBox is called.

Drawing a TextBox

⋮

Process finished with exit code 0

Overriding Methods

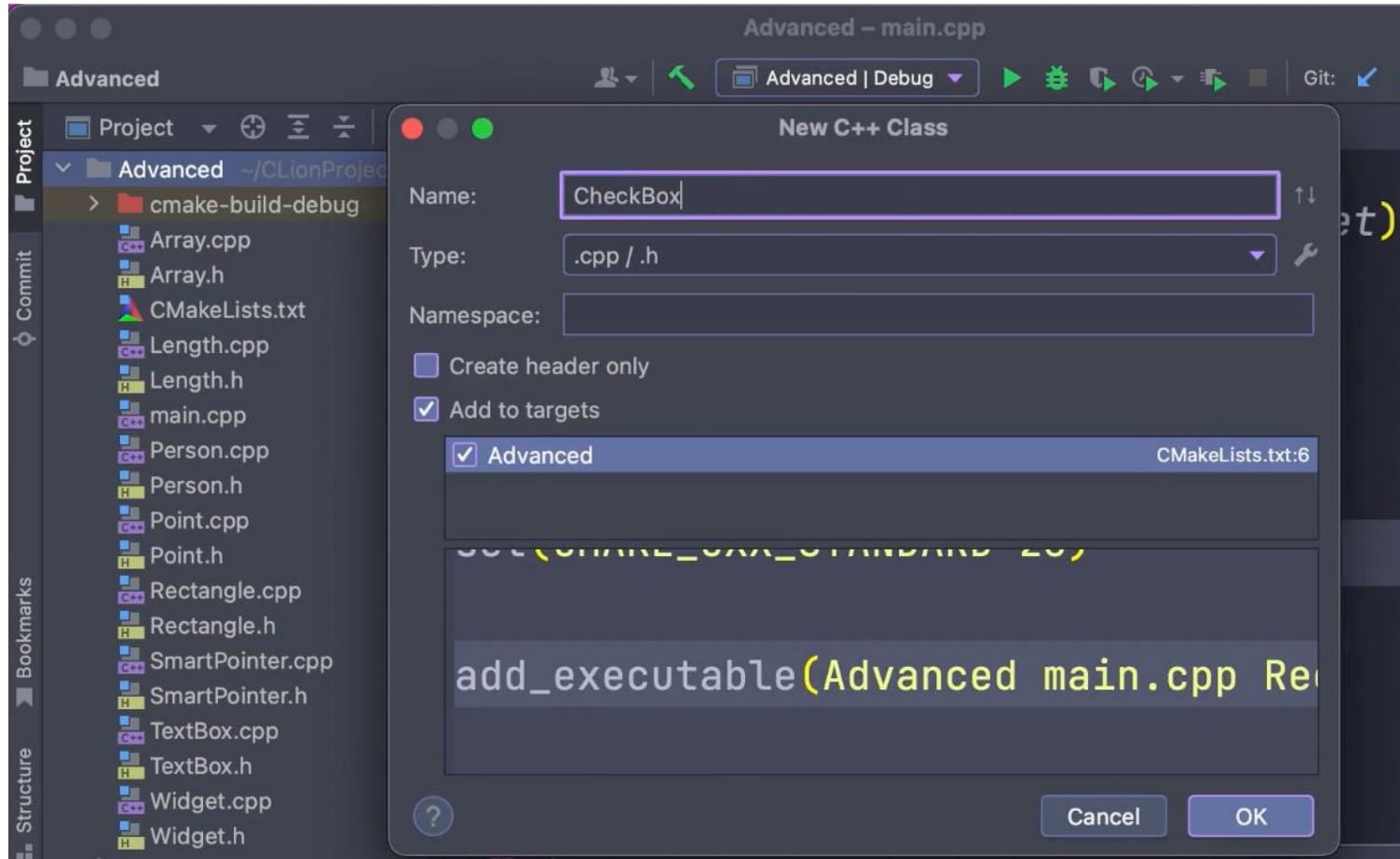
```
main.cpp x Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x
8
9  class Widget {
10 public:
11     virtual void draw() const;
12     void enable();
13     void disable();
14     bool isEnabled() const;
15 private:
16     bool enabled;
17     int width;
18 };
```

```
main.cpp x Widget.h x Widget.cpp x TextBox.h x TextBox.cpp x
12
13 class TextBox : public Widget {
14 public:
15     // Overriding
16     // Overloading
17     void draw() const override;
18     string getValue();
19     void setValue(const string& value);
20 private:
21     string value;
22 };
```

- ❑ If we want to use the draw() function of TextBox, then we use “virtual” keyword.
- ❑ The compiler uses “dynamic (late) binding” technique.
- ❑ Any method that is overridden by the inheriting class should be declared as virtual in the base class.
- ❑ Overriding is different than overloading.
- ❑ **Overloading:** We create multiple versions of the same function with different signatures (parameters)
- ❑ **Overriding:** Redefining a function with the same signature.

Polymorphism

- ❑ One of the core principles of OOP.
- ❑ Allows us to build applications that can easily be extended.



- ❑ Create a class called CheckBox

Polymorphism

Many forms

Polymorphism

An object taking many forms

```
7  
8     #include "Widget.h"  
9  
10 class CheckBox : public Widget {  
11  
12 };  
13  
14  
15 #endif //ADVANCED_CHECKBOX_H  
16
```

```
10 class CheckBox : public Widget {  
11  
12 };  
13  
14  
15 #endif //ADVANCED_CHECKBOX_H  
16
```

A context menu is open over the class definition, showing options like 'Generate', 'Constructor', 'Destructor', 'Getter', 'Setter', 'Getter and Setter', 'Equality Operators', 'Relational Operators', 'Stream Output Operator', 'Override Functions...', 'Implement Functions...', 'Copyright', and 'Generate Definitions...'. The 'Override Functions...' option is highlighted.

- Press CTRL+ENTER
- Select Override Functions

Polymorphism

- One of the core principles of OOP.
- Allows us to build applications that can easily be extended.



```
7  
8     #include "Widget.h"  
9  
10 class CheckBox : public Widget {  
11  
12 };  
13  
14  
15 #endif //ADVANCED_CHECKBOX_H  
16
```

```
10 class CheckBox : public Widget {  
11  
12 };  
13  
14  
15  
16
```

Generate
Constructor
Destructor
Getter
Setter
Getter and Setter
Equality Operators
Relational Operators
Stream Output Operator
Override Functions...
Implement Functions...
Copyright
Generate Definitions... ⌘⌘D

Polymorphism

- One of the core principles of OOP.
- Allows us to build applications that can easily be extended.

Polymorphism

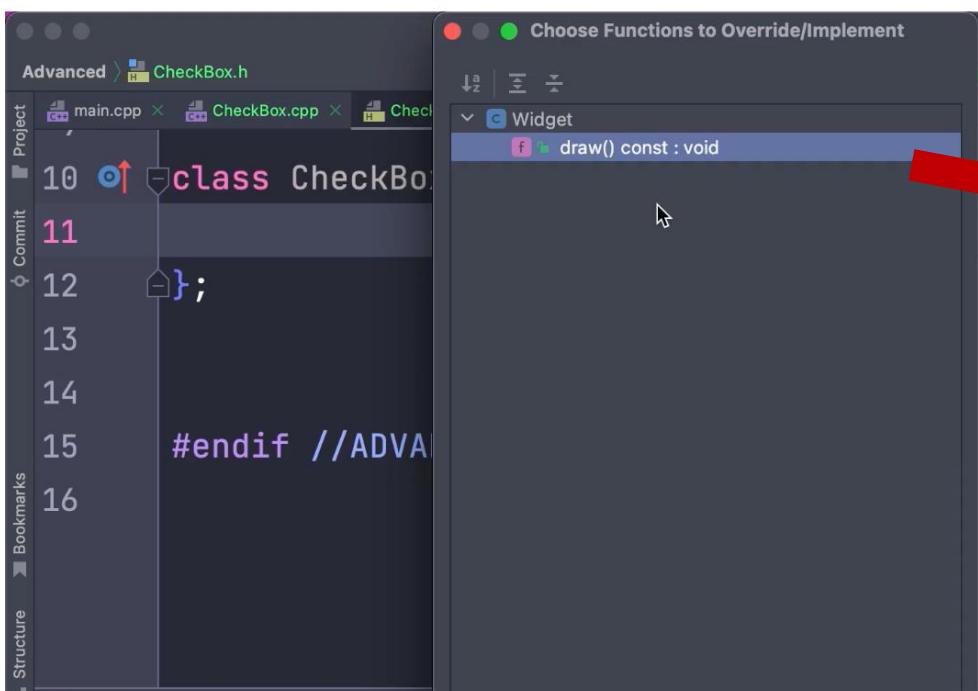
Many form

Polymorphism

An object taking many forms

- Press CTRL+ENTER
- Select Override Functions

Polymorphism



Advanced > CheckBox.h

Choose Functions to Override/Implement

Widget

draw() const : void

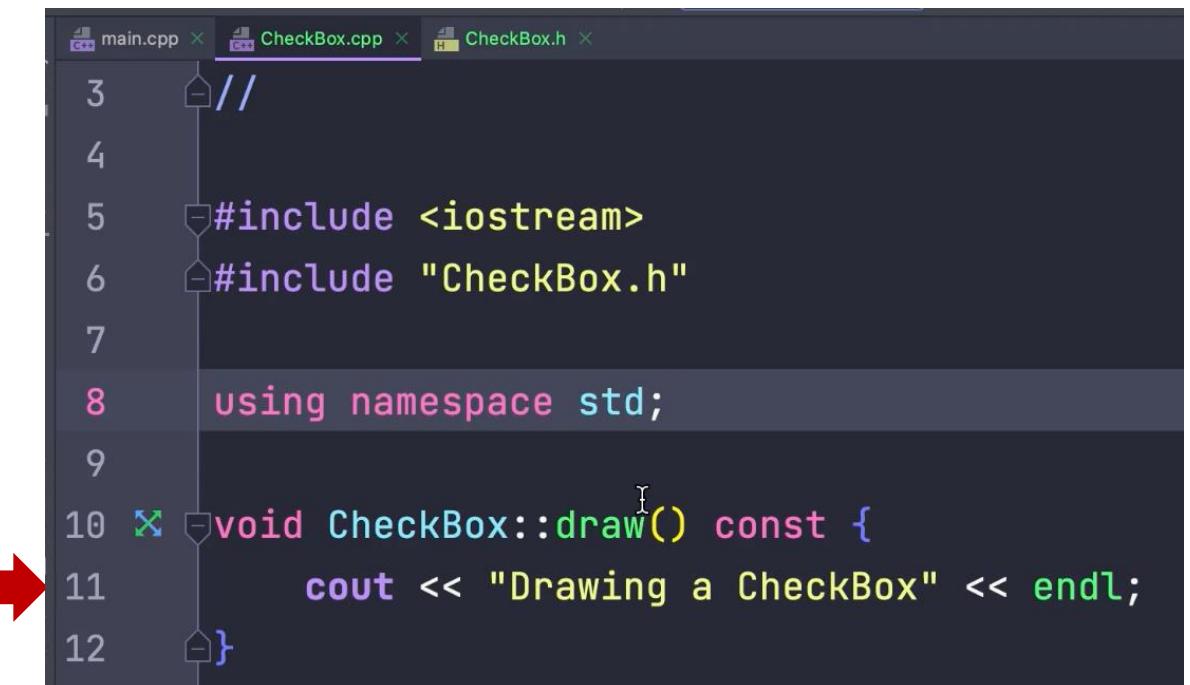
```
10 class CheckBox {
11
12     ...
13
14
15 #endif //ADVANCED
16 }
```

Select draw method



main.cpp CheckBox.cpp CheckBox.h

```
3 // ...
4
5 #include "CheckBox.h"
6
7 void CheckBox::draw() const {
8     Widget::draw();
9 }
10 }
```



main.cpp CheckBox.cpp CheckBox.h

```
3 // ...
4
5 #include <iostream>
6 #include "CheckBox.h"
7
8 using namespace std;
9
10 void CheckBox::draw() const {
11     cout << "Drawing a CheckBox" << endl;
12 }
```

Update the draw method.

Polymorphism

```
main.cpp  CheckBox.cpp  CheckBox.h
1 #include <iostream>
2 #include "CheckBox.h"
3 #include "TextBox.h"
4 #include "Widget.h"
5
6 using namespace std;
7
8 void showWidget(Widget& widget) {
9     widget.draw();
10 }
11
12 int main() {
13     TextBox box;
14     showWidget(& box);
15
16     CheckBox checkBox;
17     showWidget(checkBox);
18
19     return 0;
20 }
```

Widget takes the form of a TextBox

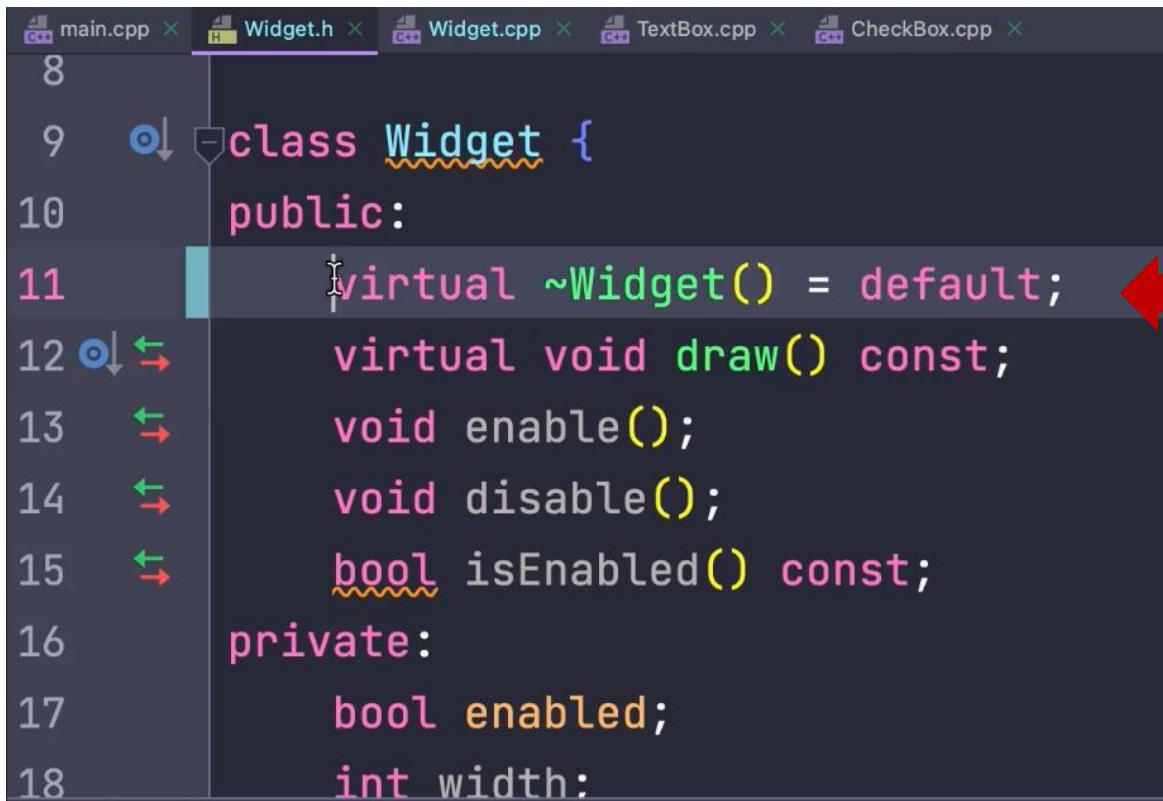
Widget takes the form of a CheckBox

```
/Users/moshfeghamedani/CLionProje
Drawing a TextBox
Drawing a CheckBox
{
Process finished with exit code 0
```

Output of the main function.

Virtual Destructors

- When we have virtual methods in a class, then we need to create a virtual destructor as well.
- If we have an empty destructor (we still need to declare it in the header file), then we do not need to define it; the compiler will generate for us.



```
main.cpp Widget.h Widget.cpp TextBox.cpp CheckBox.cpp
8
9 class Widget {
10 public:
11     virtual ~Widget() = default;
12     virtual void draw() const;
13     void enable();
14     void disable();
15     bool isEnabled() const;
16 private:
17     bool enabled;
18     int width;
```



```
main.cpp Widget.h Widget.cpp TextBox.cpp CheckBox.cpp
23     cout << "Drawing a Widget" << endl;
24 }
25
26 Widget::~Widget() {
27 }
```

Abstract Classes

```
main.cpp X Widget.h X Widget.cpp X TextBox.cpp X TextBox.h X CheckBox.cpp X
8
9  class Widget {
10
11     public:
12         virtual ~Widget() = default;
13         virtual void draw() const; // Red arrow points here
14         void enable();
15         void disable();
16         bool isEnabled() const;
17
18     private:
19         bool enabled;
20         int width;
21
22     public: // Red arrow points here
23         bool isEnabled() const {
24             return enabled;
25         }
26
27         void Widget::draw() const {
28             cout << "Drawing a Widget" << endl;
29         }
30 }
```

- The draw() method in the Widget class has no actual implementation.
- Each widget type has a different drawing algorithm and should implement its own draw method.

- No real implementation; we are simply printing a message.

Abstract Classes

- ❑ The draw() method in the Widget class has no actual implementation.
- ❑ We declare our virtual method in the Widget class as **pure virtual method**.
- ❑ If a class has at least one virtual method, the the class is called **“Abstract Class”**.
- ❑ Abstract classes cannot be instantiated; they exists purely to be inherited.
- ❑ This will be removed form the implementation file.

```
C++ main.cpp ✘ Widget.h ✘ Widget.cpp ✘ TextBox.cpp ✘ TextBox.h ✘ CheckBox.cpp ✘
8
9 // Abstract
10 class Widget {
11 public:
12     virtual ~Widget() = default;
13     // Pure virtual method
14     virtual void draw() const = 0;
15     void enable();
16     void disable();
17     bool isEnabled() const;
18 }
```



```
C++ main.cpp ✘ Widget.h ✘ Widget.cpp ✘ TextBox.cpp ✘ TextBox.h ✘ CheckBox.cpp ✘
10     bool isEnabled() const {
11         return enabled;
12     }
13
14     void Widget::draw() const {
15         cout << "Drawing a Widget" << endl;
16     }
17 }
```



Abstract Classes

- The draw() method in the Widget class has no actual implementation.
- We declare our virtual method in the Widget class as **pure virtual method**.
- If a class has at least one virtual method, the the class is called “**Abstract Class**”.
- Abstract classes cannot be instantiated; they exists purely to be inherited.
- We will get compiler error.

```
8
9 // Abstract
10 class Widget {
11 public:
12     virtual ~Widget() = default;
13     // Pure virtual method
14     virtual void draw() const = 0;
15     void enable();
16     void disable();
17     bool isEnabled() const;
```

```
13
14 int main() {
15     Widget widget;
16     vector<unique_ptr<Widget>> widgets;
17     widgets.push_back(make_unique<TextBox>());
18     widgets.push_back(make_unique<CheckBox>());
19     for (const auto& widget : const unique_ptr<...> & : widgets)
20         widget->draw();
```

Templates

- ❑ Templates help us write generic functions and classes.

- Understanding the problem
- Function templates
- Class templates

Defining a Function Template

```
C++ main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int larger(int first, int second) {
6     return (first > second) ? first : second;
7 }
8
9 int main() {
10    larger(first: 1.1, second: 2.2);
11
12    return 0;
13 }
```

- Double type arguments will be converted to int type and we will lose precision.

Defining a Function Template

```
C++ main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int larger(int first, int second) {
6     return (first > second) ? first : second;
7 }
8
9 double larger(double first, double second) {
10    return (first > second) ? first : second;
11 }
12
13 int main() {
14     larger(first: 1.1, second: 2.2);
15
16     return 0;
17 }
```



- **Solution:** Use overloading and implement same function with type double.
- **Problem:** We need to create overloading functions for each type.
- Code will get large and hard to maintain.

```
C++ main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 template<typename T>
6 T larger(T first, T second) {
7     return (first > second) ? first : second;
8 }
9
10 int main() {
11     larger(first: 1.1, second: 2.2);
12     larger(first: 1, second: 2);
13     larger(first: "a", second: "b");
14
15     return 0;
16 }
17 }
```

Defining a Function Template

- **Solution:** Use function template and make it generic.

- Instead of **template<typename T>**, **template<class T>** can also be used.

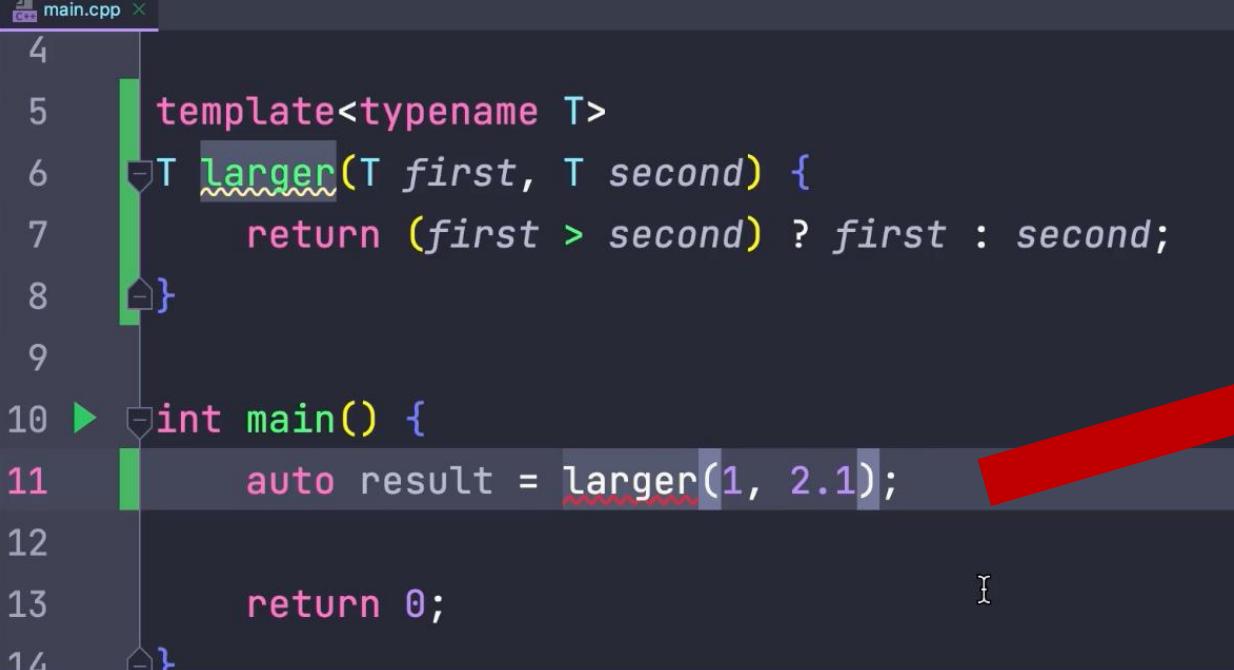
```
C++ main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 template<typename T>
6 T larger(T first, T second) {
7     return (first > second) ? first : second;
8 }
9
10 int main() {
11     larger(first: 1.1, second: 2.2);
12     larger(first: 1, second: 2);
13     larger(first: "a", second: "b");
14
15     return 0;
16 }
17 }
```

Defining a Function Template

- **Solution:** Use function template and make it generic.
- Instead of **template<typename T>**, **template<class T>** can also be used.
- Compiler generates only the instances of the functions where they are used in the code.
- Size of the executable file will be smaller compared to using function overloading.

Explicit Type Arguments

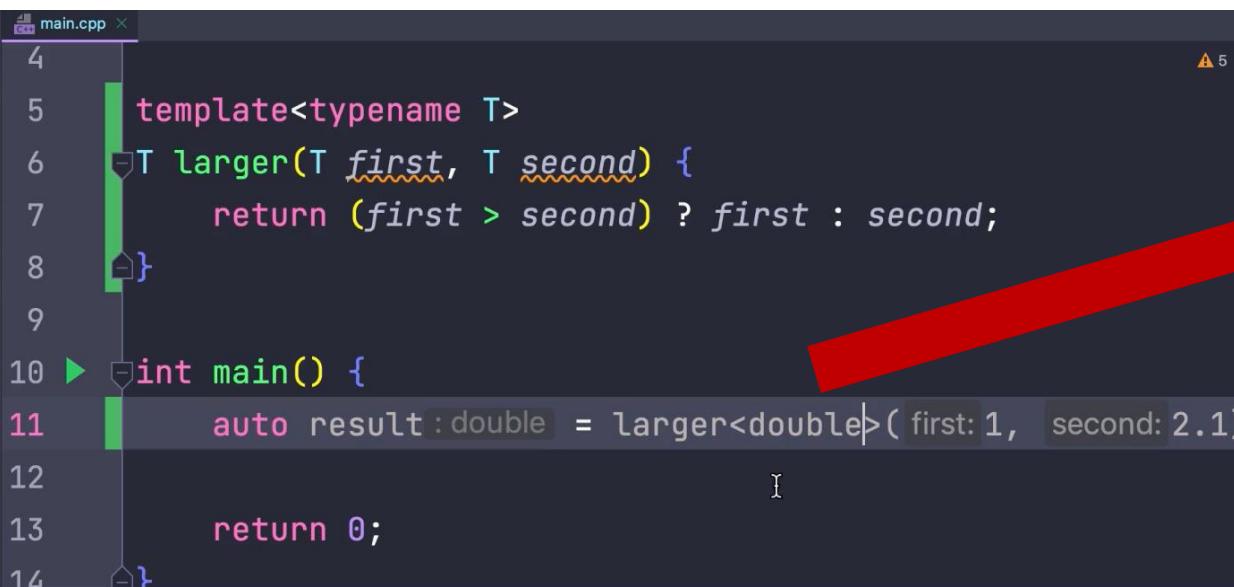
□ **Problem:** The C++ compiler cannot act properly to estimate the types of the arguments.



```
4
5     template<typename T>
6     T larger(T first, T second) {
7         return (first > second) ? first : second;
8     }
9
10    int main() {
11        auto result = larger(1, 2.1);
12
13        return 0;
14    }
```

A screenshot of a code editor showing a template function named 'larger' and its usage in 'main()'. The function takes two arguments of type T and returns the larger one. In the 'main()' function, there is a call to 'larger(1, 2.1)'. A red arrow points from this call to the 'Solution' section below.

□ **Solution:** We explicitly specify the type of the arguments.



```
4
5     template<typename T>
6     T larger(T first, T second) {
7         return (first > second) ? first : second;
8     }
9
10    int main() {
11        auto result : double = larger<double>(1, 2.1);
12
13        return 0;
14    }
```

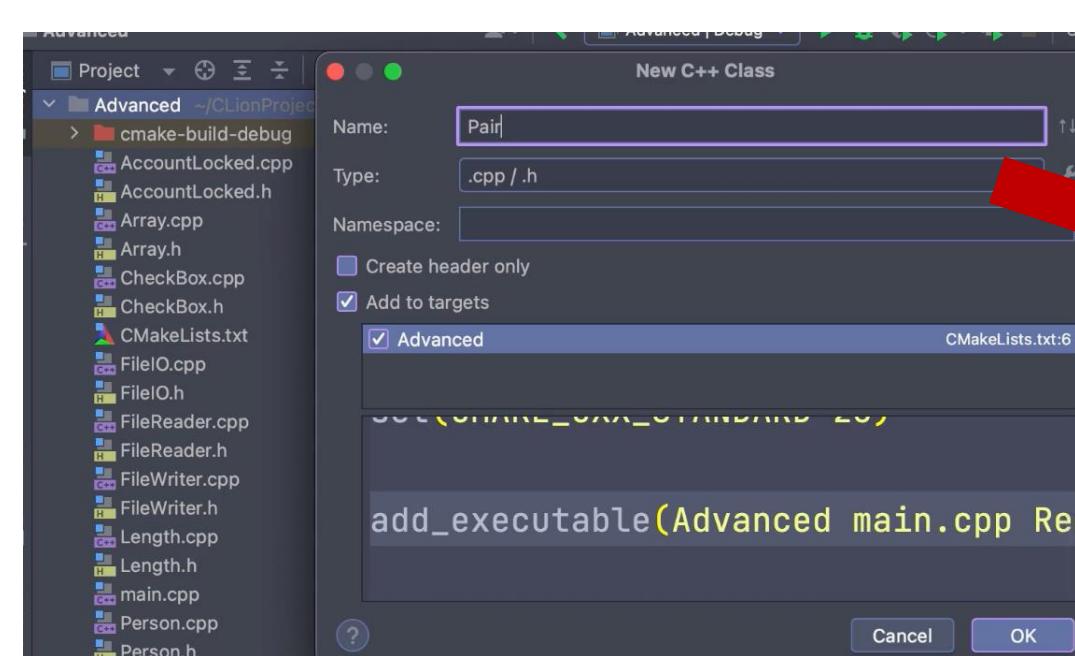
A screenshot of a code editor showing the same code as above, but with explicit type arguments. The call to 'larger' is now 'larger<double>(1, 2.1)', where the type 'double' is explicitly specified. A red arrow points from this call to the 'Problem' section above.

Templates with Multiple Parameters

```
main.cpp C++  
4  
5     template<typename T>  
6     T larger(T first, T second) {  
7         return (first > second) ? first : second;  
8     }  
9  
10    template<typename K, typename V>  
11    void display(K key, V value) {  
12        cout << key << "=" << value << endl;  
13    }  
14  
15 ▶ int main() {  
16     display( key: "a", value: 1);  
17     display( key: 1, value: 1);  
18  
19     return 0;  
20 }
```

Defining a Class Template

☐ Create a new class called Pair.

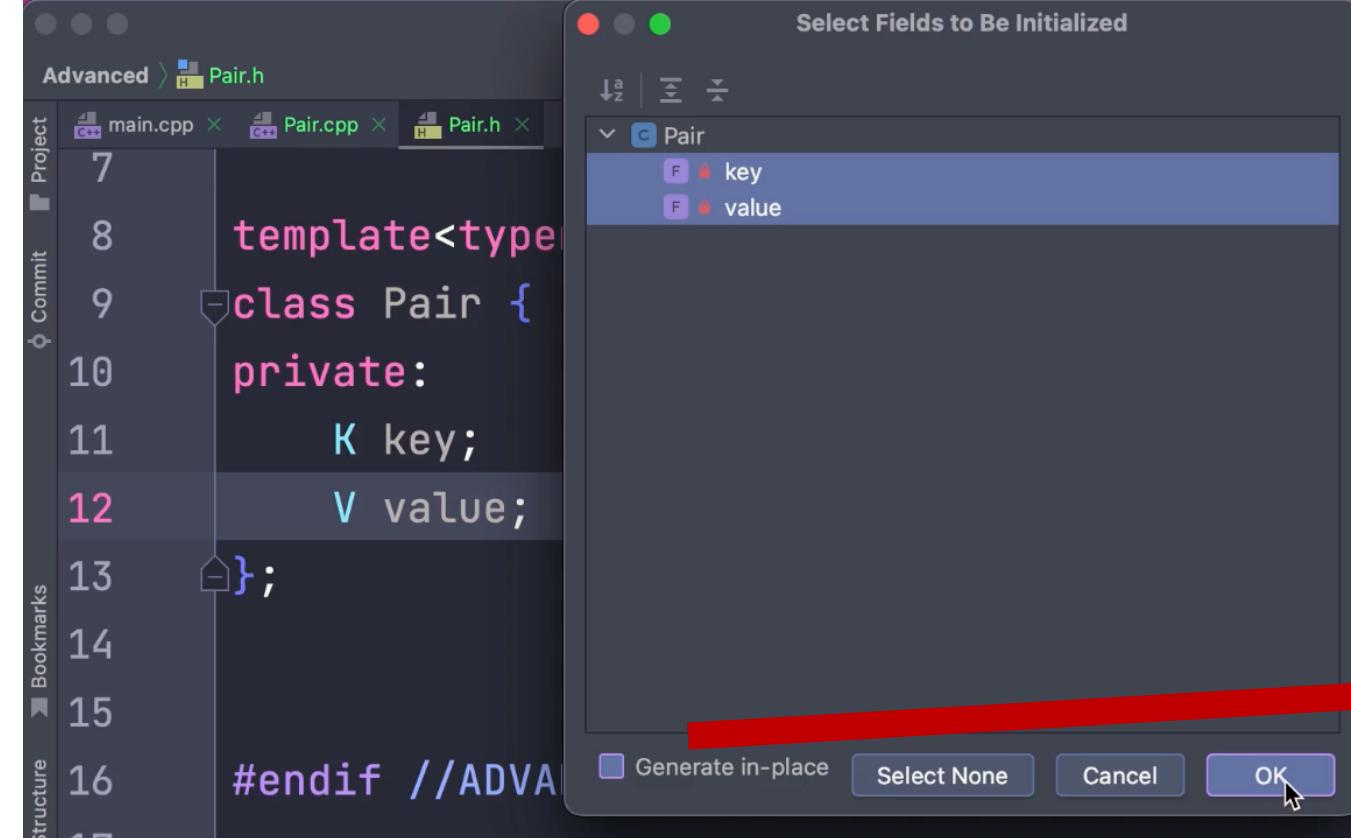


```
5 #ifndef ADVANCED_PAIR_H
6 #define ADVANCED_PAIR_H
7
8 template<typename K, typename V>
9 class Pair {
10 private:
11     K key;
12     V value;
13 };
```

Defining a Class Template

- Press CTRL-N (COMMAND-N) to bring the generate palette.

- Uncheck this box.



Defining a Class Template

```
main.cpp x  Pair.cpp x  Pair.h x
5  #ifndef ADVANCED_PAIR_H
6  #define ADVANCED_PAIR_H
7
8  template<typename K, typename V>
9  class Pair {
10  private:
11      K key;
12      V value;
13  public:
14      ↗ Pair(K key, V value);
15  };
16
17  template<typename K, typename V>
18  ↗ Pair<K, V>::Pair(K key, V value):key(key), value(value) {}
19
20
21  #endif //ADVANCED_PAIR_H
```

Defining a Class Template

A screenshot of a code editor showing a context menu for a class template. The menu is open over the word 'Pair' in the line 'Pair<typename K, typename V>'. The menu options include 'Generate', 'Constructor', 'Destructor', 'Getter', 'Setter', 'Getter and Setter', 'Equality Operators', 'Relational Operators', 'Stream Output Operator', and 'Override Functions...'. The 'Getter' option is highlighted.

```
7
8     template<typename K, typename V>
9     class Pair {
10     private:
11         K key;
12         V value;
13     public:
14     Pair(K k);
15 };
16 
```

A screenshot of a code editor showing a 'Select Fields to Generate Getters' dialog. The dialog lists fields for the 'Pair' class: 'key' and 'value'. Both fields have an unchecked checkbox next to them. A large red arrow points from the bottom-left towards the checkboxes.

```
Advanced > Pair.h
Project Commit Bookmarks Structure
7
8     template<typename K, typename V>
9     class Pair {
10     private:
11         K key;
12         V value;
13     public:
14     Pair(K k);
15 };
16 
```

Select Fields to Generate Getters

- Pair
 - key
 - value

Generate in-place Cancel OK

- Create Getter for key and value.
- CTRL-N, then select Getter.

- Make sure the box is unchecked.

Defining a Class Template

```
22 template<typename K, typename V>
23     Pair<K, V>::Pair(K key, V value):key(key), value(value) {}
24
25     template<typename K, typename V>
26         K Pair<K, V>::getKey() const {
27             return key;
28         }
29
30     template<typename K, typename V>
31         V Pair<K, V>::getValue() const {
32             return value;
33         }
```

```
1 #include <iostream>
2 #include "Pair.h"
3
4 using namespace std;
5
6 int main() {
7     Pair pair{ key: "a", value: 1 };
8
9     return 0;
10 }
```

- Compiler will create a Pair class with key is a string and value is an integer.