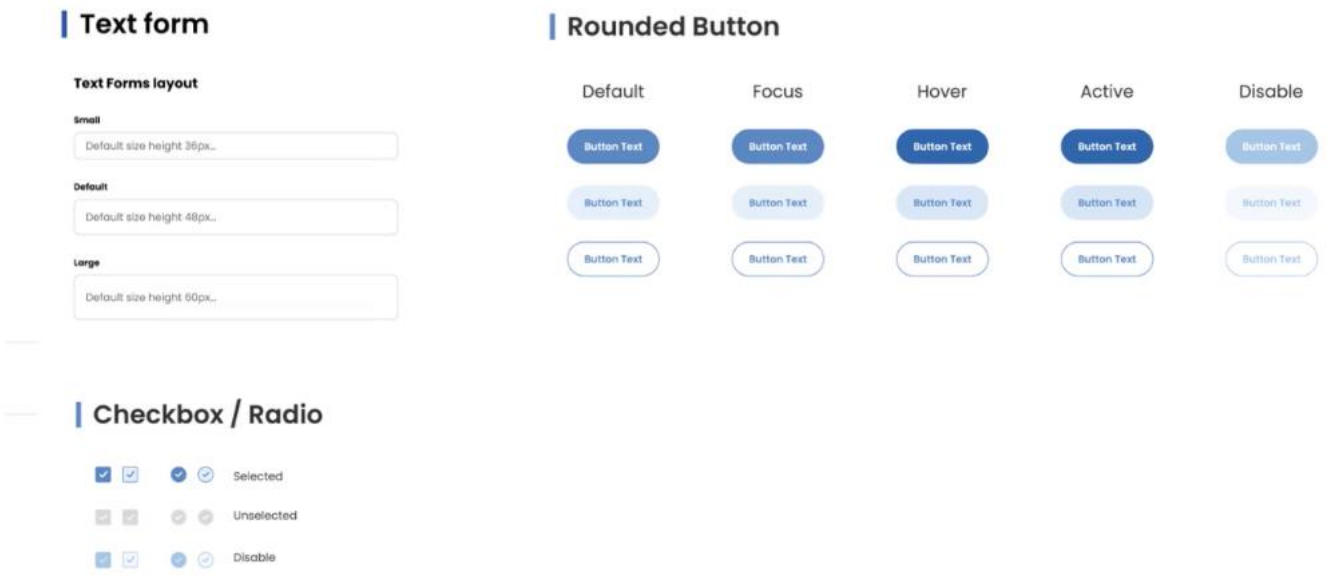
C++ Notes

Part 3-2

Slides were created from CodewithMosh.com

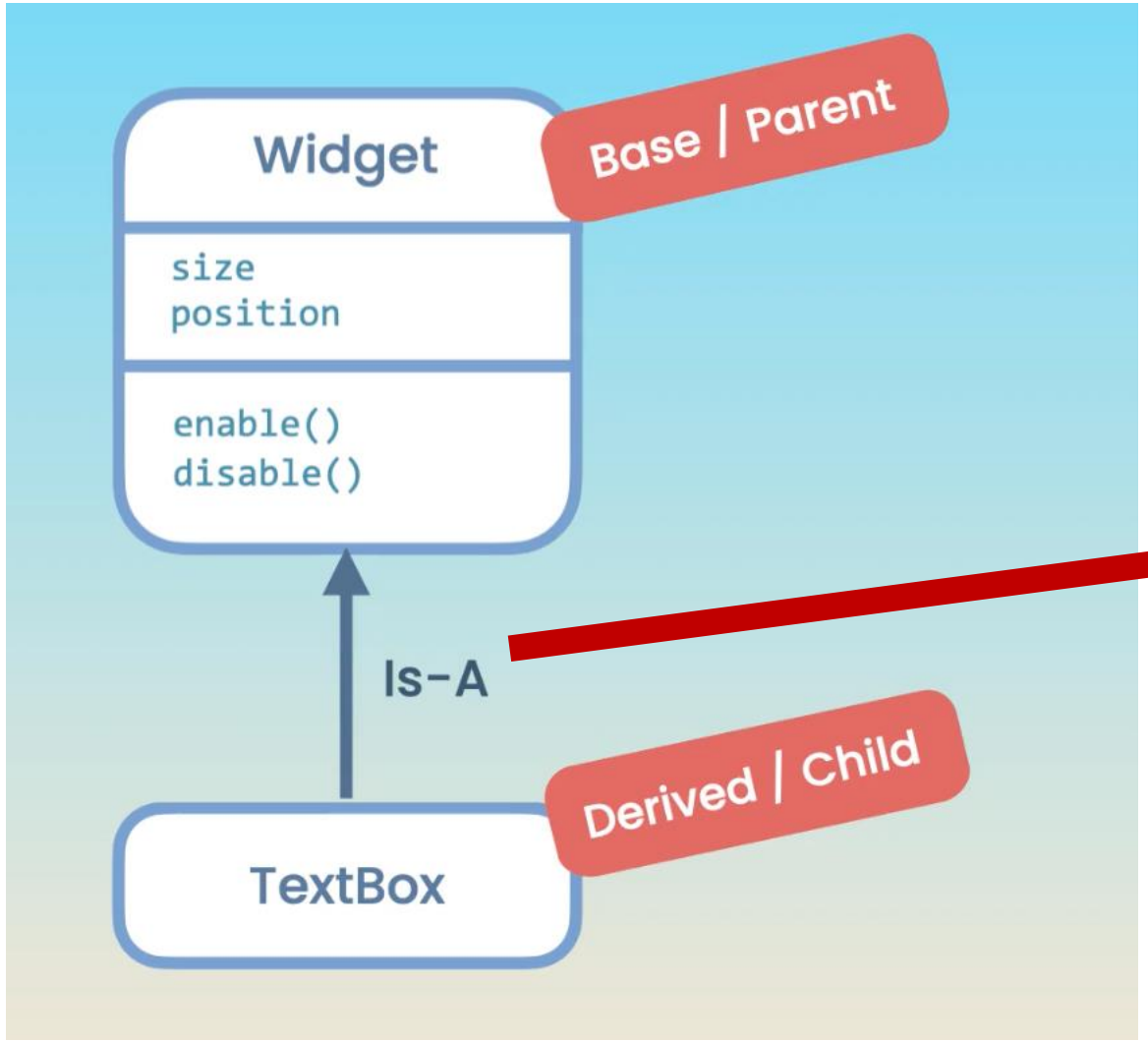
Inheritance and Polymorphism

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 | Widget.h

```cpp
//

#include "Widget.h"

void Widget::enable() {
    enabled = true;
}

void Widget::disable() {
    enabled = false;
}

bool Widget::isEnabled() const {
    return enabled;
}
```

Widget.cpp | Widget.h

```cpp

class Widget {
public:
    void enable();
    void disable();
    bool isEnabled() const;
private:
    bool enabled;
};
```

- 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.

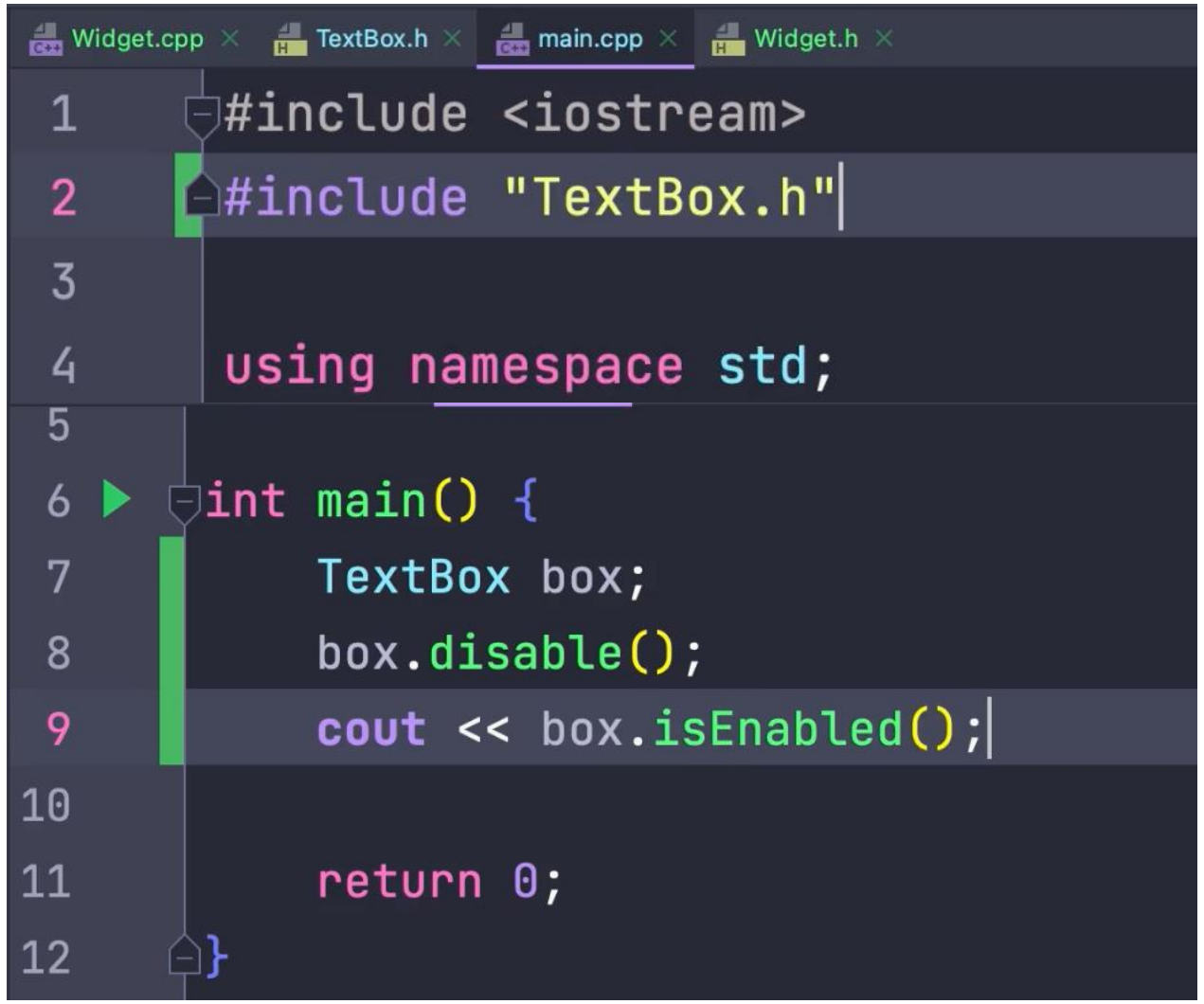
Widget.cpp | TextBox.h | Widget.h

```
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  };
```

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

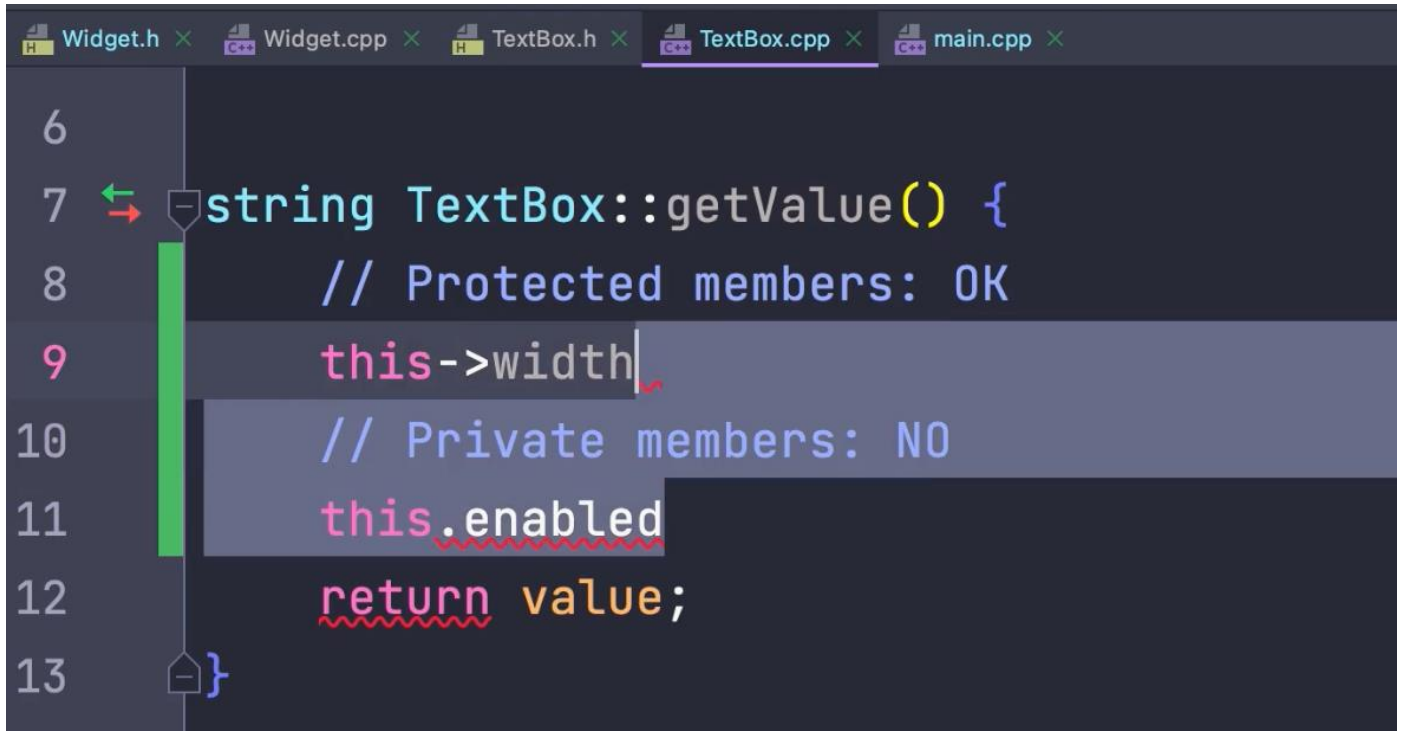


- Using inheritance, we can create new classes by reusing and expanding on existing classes.

Protected Members

- 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.

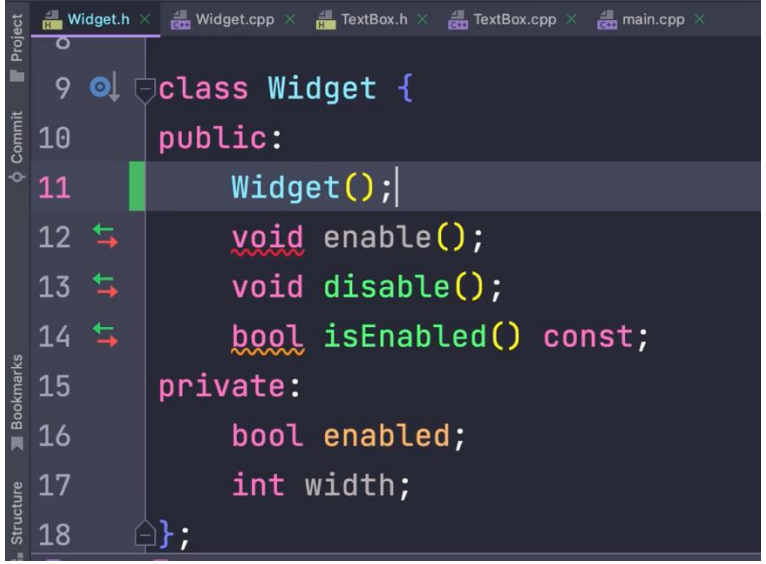
Protected Members

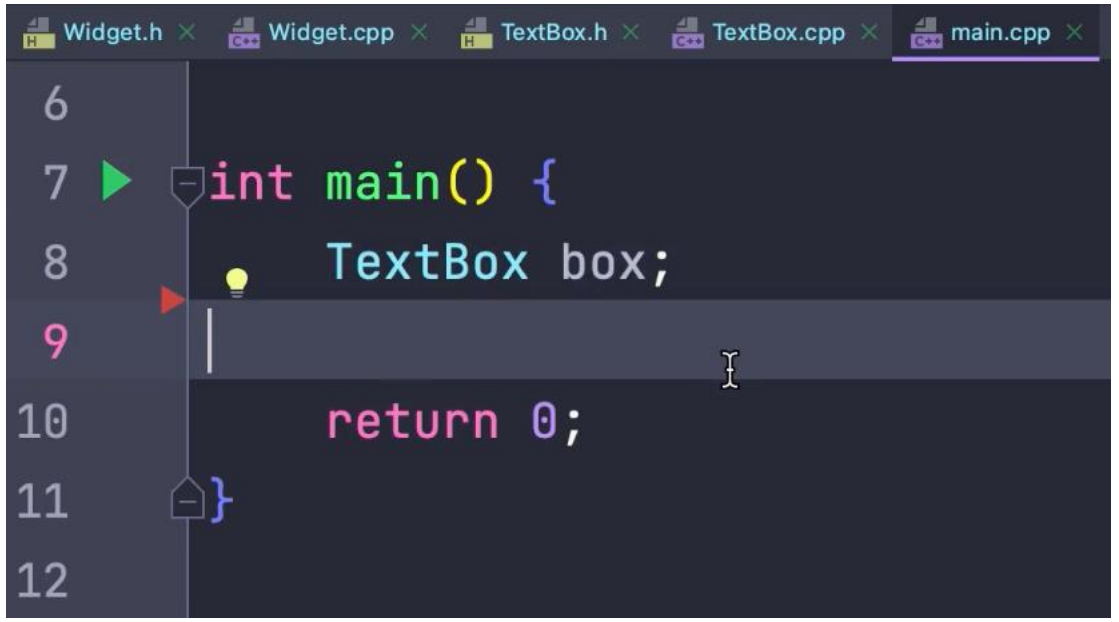


- Protected members are similar to private members.
- They cannot be accessed outside of the class.
- **Difference:** Protected members are accessible within the derived classes, but private members are not.

Constructors and Inheritance

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





Constructors and Inheritance

Widget.h | Widget.cpp | TextBox.h | TextBox.cpp | main.cpp

```
13  class TextBox : public Widget {
14  public:
15      TextBox();
16      explicit TextBox(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

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

Widget.h | Widget.cpp | TextBox.h | TextBox.cpp | main.cpp

```
16
17  }
18
19  TextBox::TextBox() {
20      cout << "TextBox constructed" << endl;
21  }
22
```

Widget.h | Widget.cpp | TextBox.h | TextBox.cpp | main.cpp

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

```
/Users/moshfeghhamedani/CLionProject
Widget constructed
TextBox constructed

Process finished with exit code 0
```

Constructors and Inheritance

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

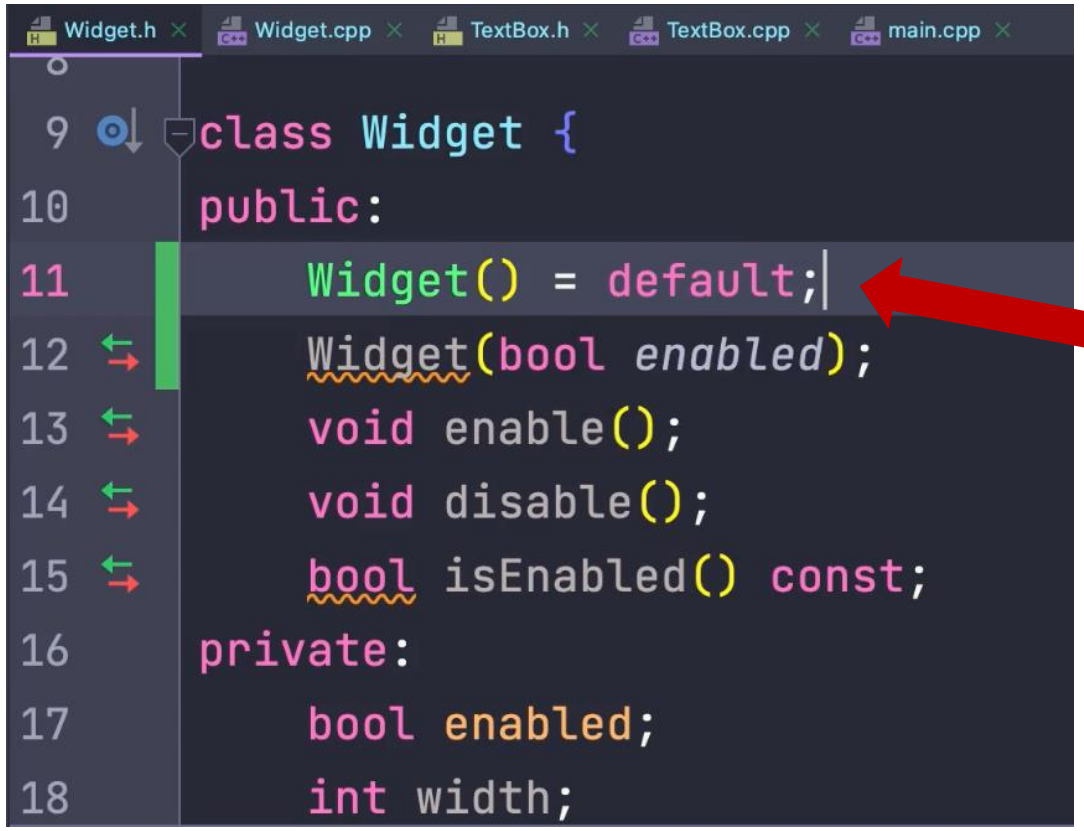
```
Widget.h  Widget.cpp  TextBox.h  TextBox.cpp  main.cpp
18  TextBox::TextBox(const string &value) : value{value} {
19
20  }
```

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

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

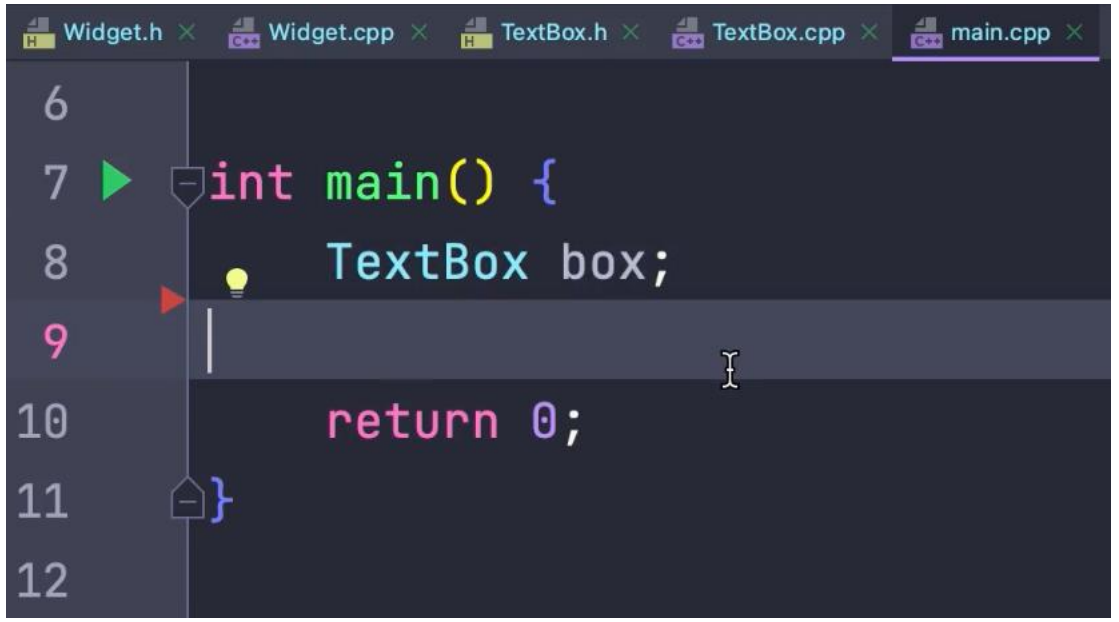
```
Widget.h  Widget.cpp  TextBox.h  TextBox.cpp  main.cpp
 6
 7  int main() {
 8      TextBox box;
 9
10      return 0;
11  }
12
```

Constructors and Inheritance



Solution 1:

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



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 };
```

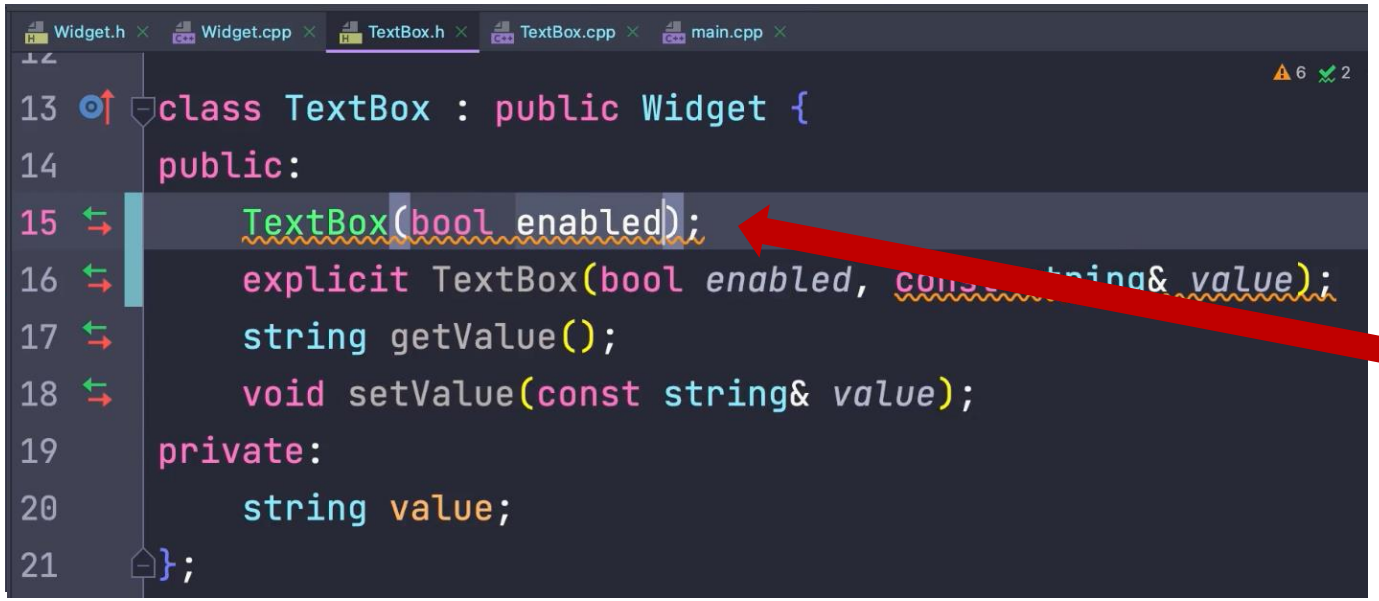
Widget.h | Widget.cpp | TextBox.h

```
18 TextBox::TextBox(bool enabled, const string &value) : Widget(enabled), value{value} {
19
20 }
```

Widget.h | Widget.cpp | TextBox.h | TextBox.cpp | main.cpp

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

Constructors and Inheritance



- Update the base class to pass the parameter.

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

```
Widget.h | Widget.cpp | TextBo | TextBox.cpp | main.cpp
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 };
```

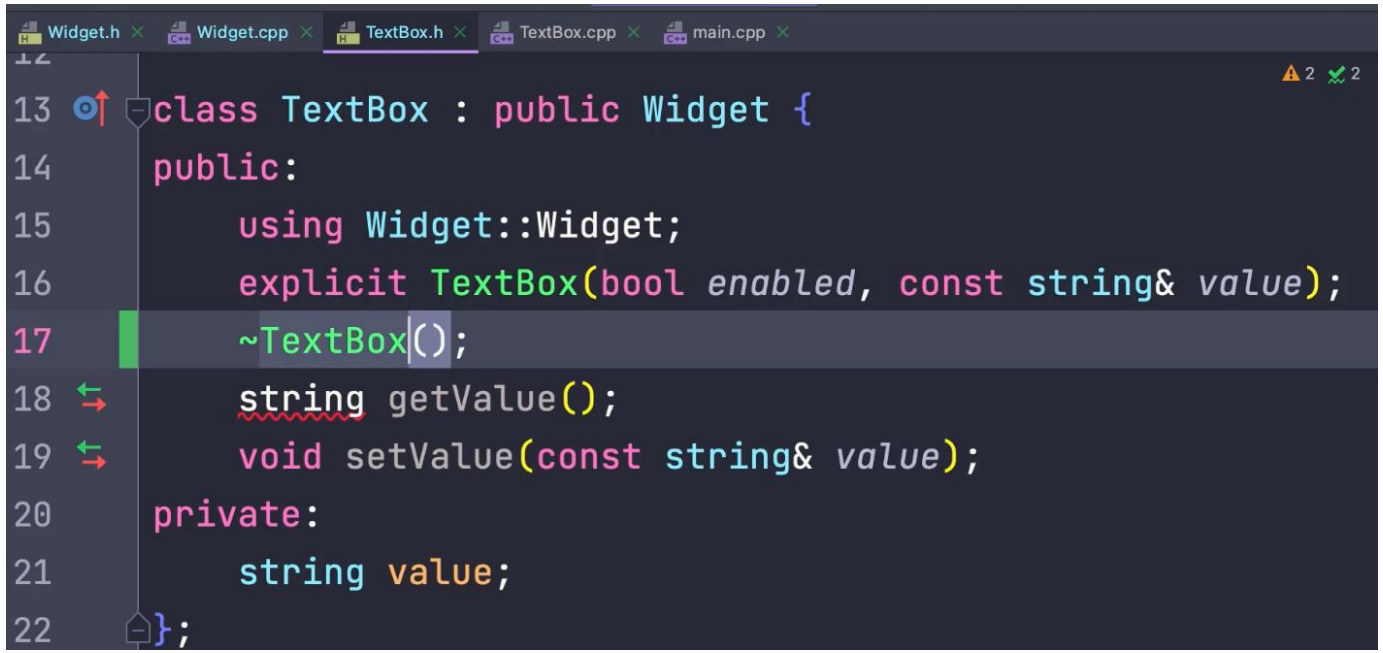
- Delete it. It is replaced with base class constructor using inheritance.

Destructors and Inheritance

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

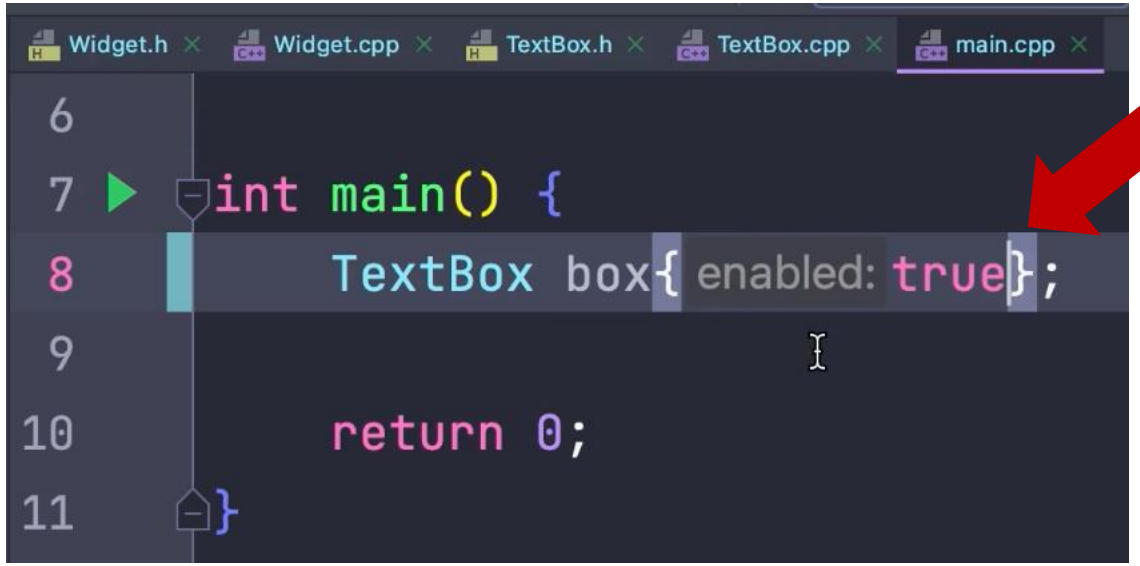
Destructors and Inheritance

- In inheritance, destructors are called in reverse order.



Destructors and Inheritance

- 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.

Destructors and Inheritance

Widget.h | Widget.cpp | TextBox.h | TextBox.cpp | main.cpp

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

```
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

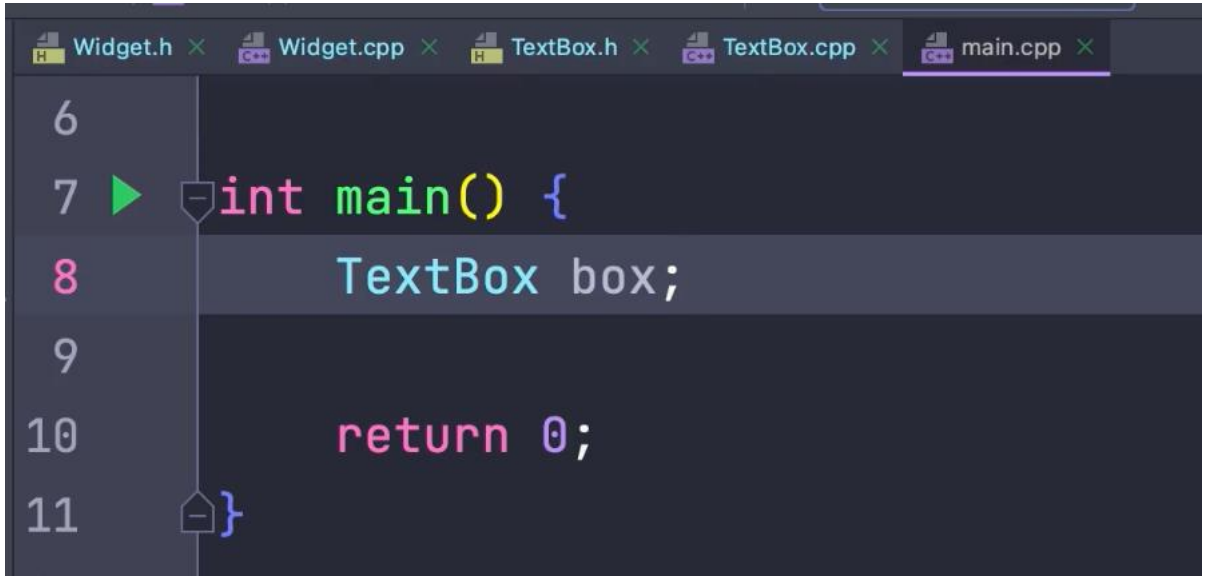
```
Widget.h  Widget.cpp  TextBox.h  TextBox.cpp  main.cpp
13  class TextBox : public Widget {
14  public:
15      using Widget::Widget;
16      explicit TextBox(bool enabled, const string& value);
17      ~TextBox();
18      string getValue();
19      void setValue(const string& value);
20  private: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

Remove the inheritance and the other constructor.

```
Widget.h  Widget.cpp  TextBox.h  TextBox.cpp  main.cpp
14  void TextBox::setValue(const string& value) {
15      this->value = value;
16  }
17
18  TextBox::TextBox(bool enabled, const string &value) : Widge
19
20  }
21
22  TextBox::~TextBox() {
23      cout << "TextBox destructed" << endl;
```

Remove the definition from the implementation file.

Destructors and Inheritance



- 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.

Output of running the main function.

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

Conversion Between Base and Derived Classes

- 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.

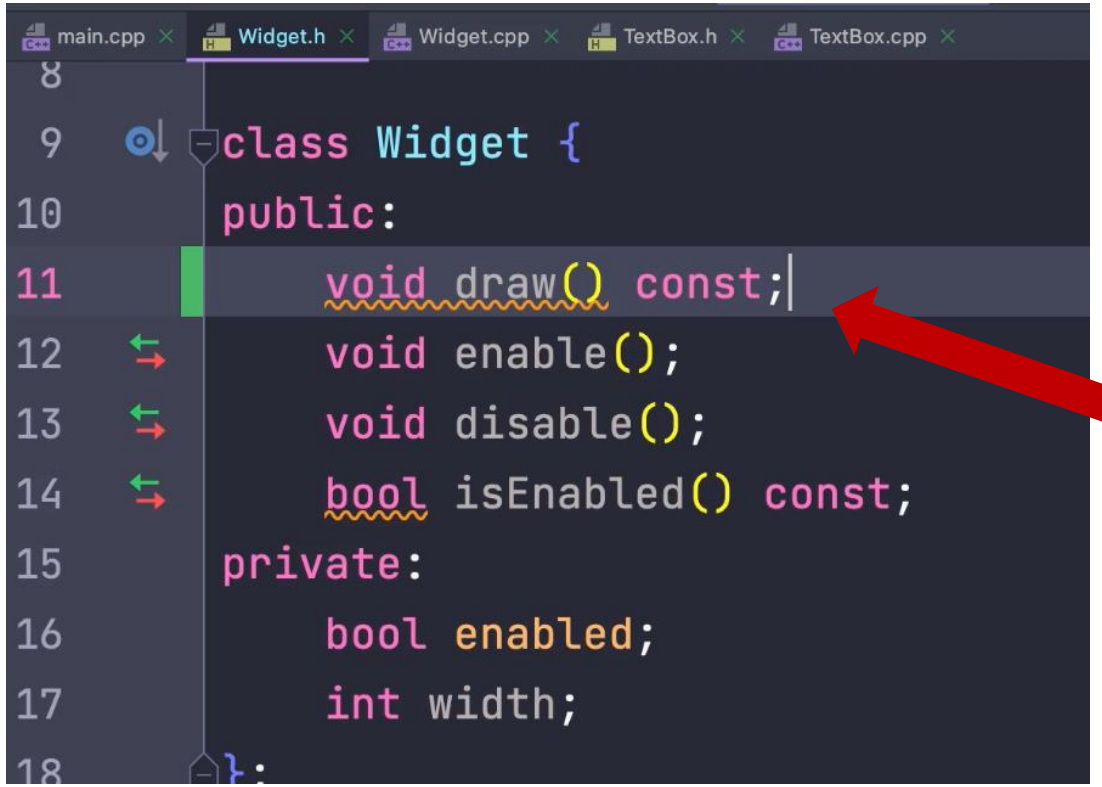
Conversion Between Base and Derived Classes

- 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.

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

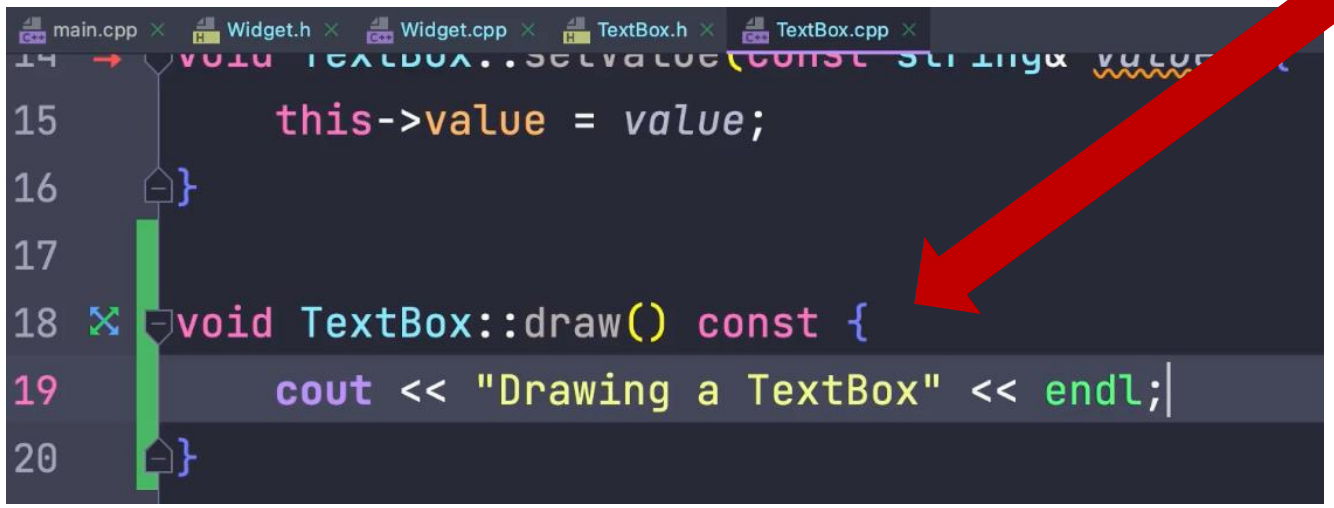
Overriding Methods



- Adding draw() function to Widget class.

Overriding Methods

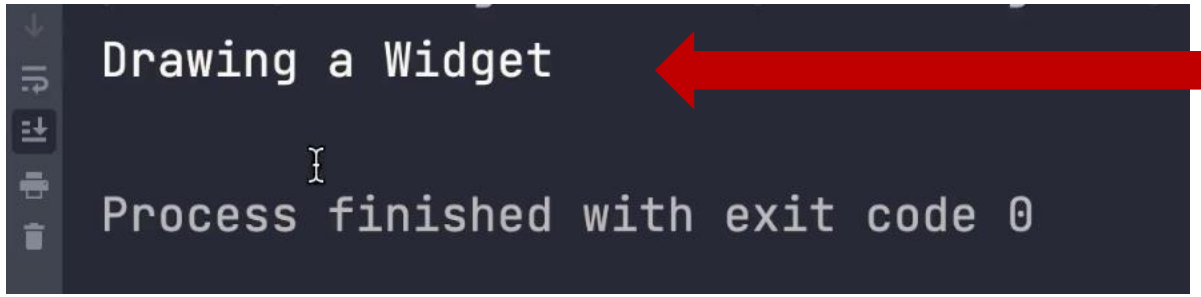
- TextBox should have its own draw() function.
- Adding draw() function to TextBox class.



Overriding Methods

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

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

- 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

main.cpp | Widget.h | Widget.cpp | TextBox.h | TextBox.cpp

```
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  };
```

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

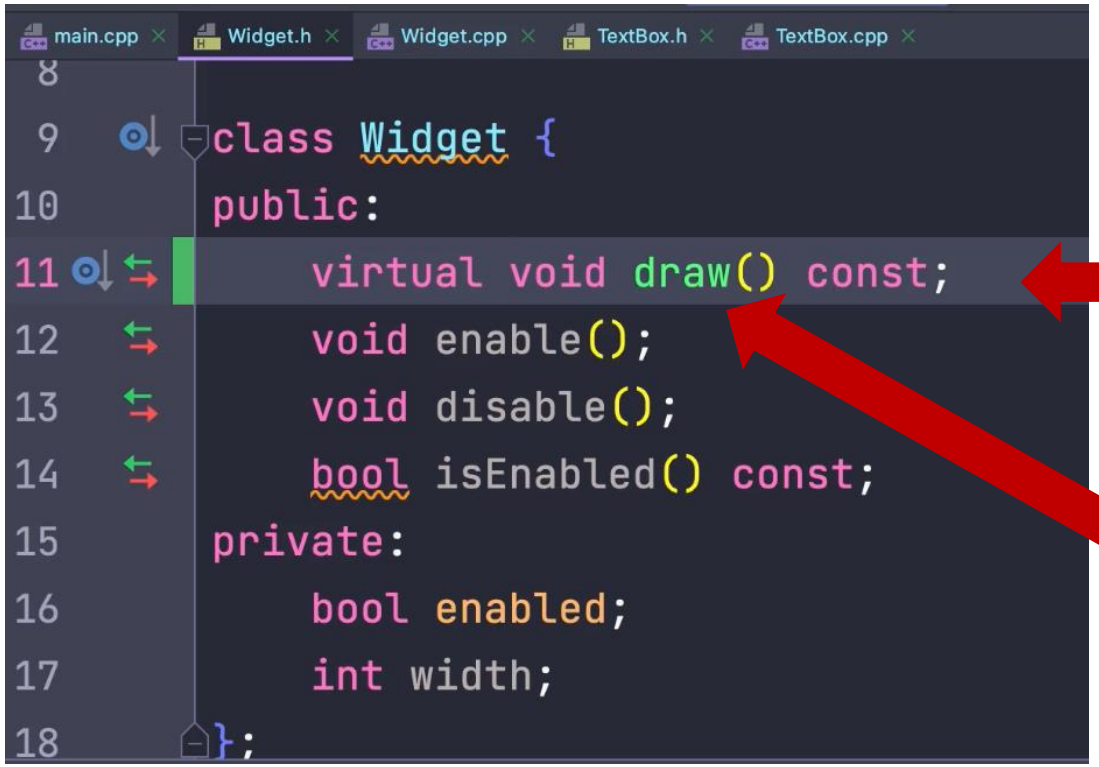
main.cpp | Widget.h | Widget.cpp | TextBox.h | TextBox.cpp

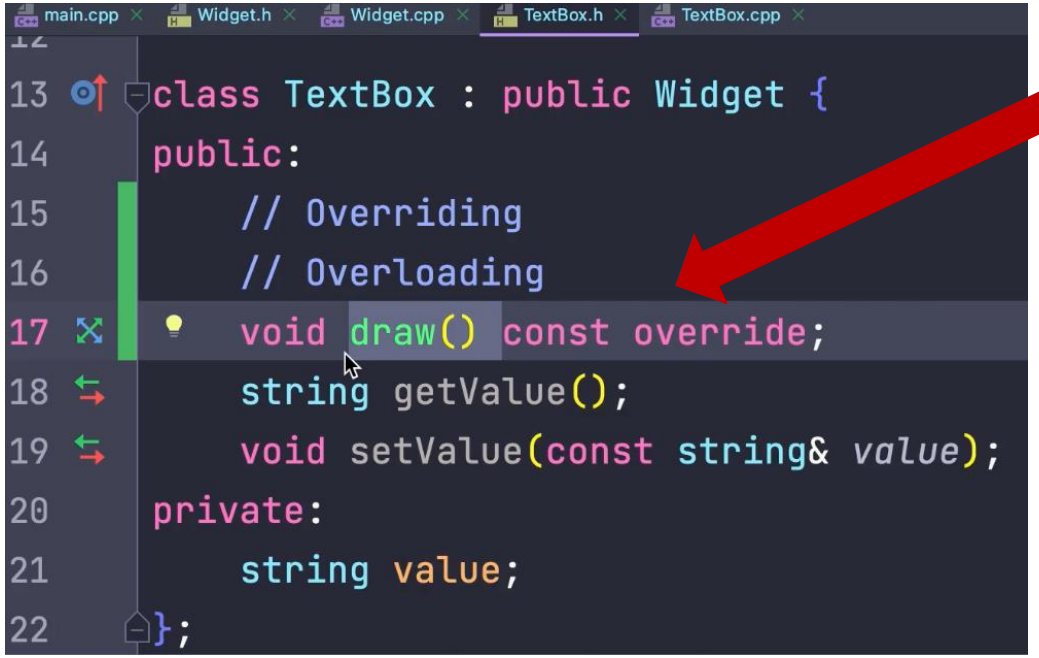
```
6
7   void showWidget(Widget& widget) {
8       widget.draw();
9   }
10
11  int main() {
12      TextBox box;
13      showWidget(&box);
14
15      return 0;
```

- Draw() method of the TextBox is called.

```
Drawing a TextBox

Process finished with exit code 0
```

Overriding Methods

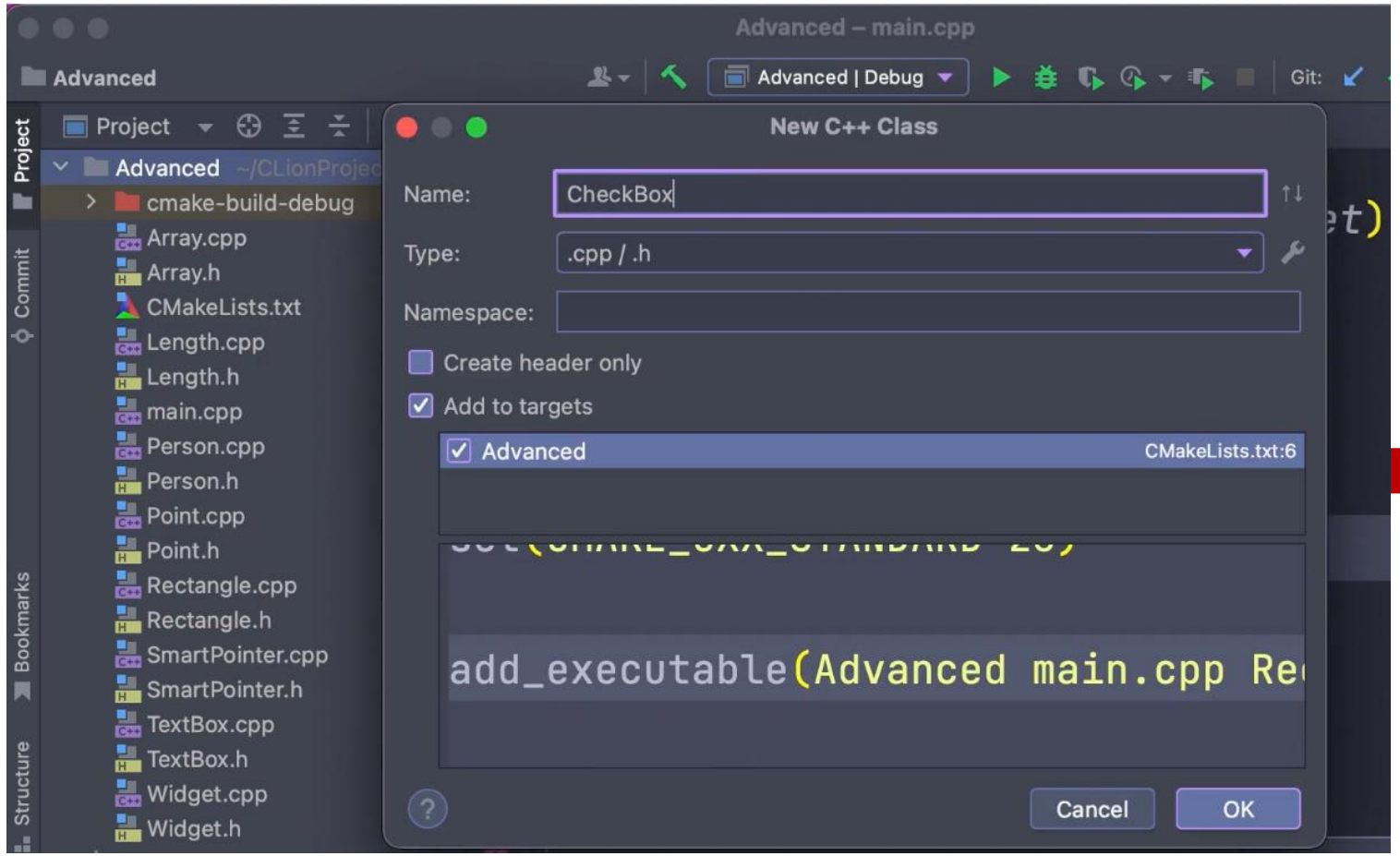




- If we want to use the draw() function of TextBox, the we use “virtual” keyword.
- The compiler uses “dynamic (late) binding” technique.
- Any method that is overridden by the inheriting class should be declared as virtual in the base class.
- Overriding is different than overloading.
- **Overloading:** 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.



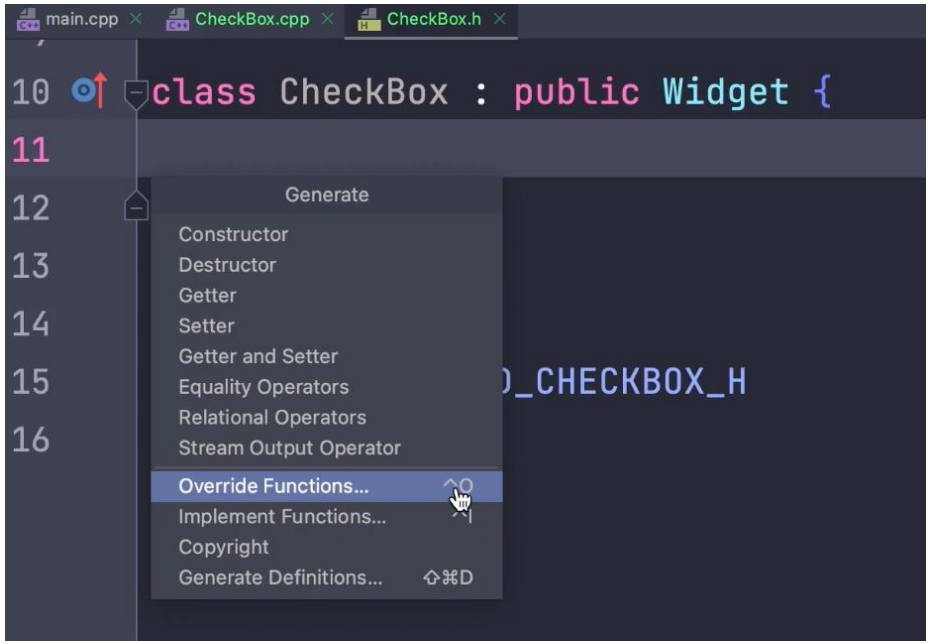
Polymorphism

Many form

Polymorphism

An object taking many forms

- Create a class called CheckBox



- Press CTRL+ENTER
- Select Override Functions

Polymorphism

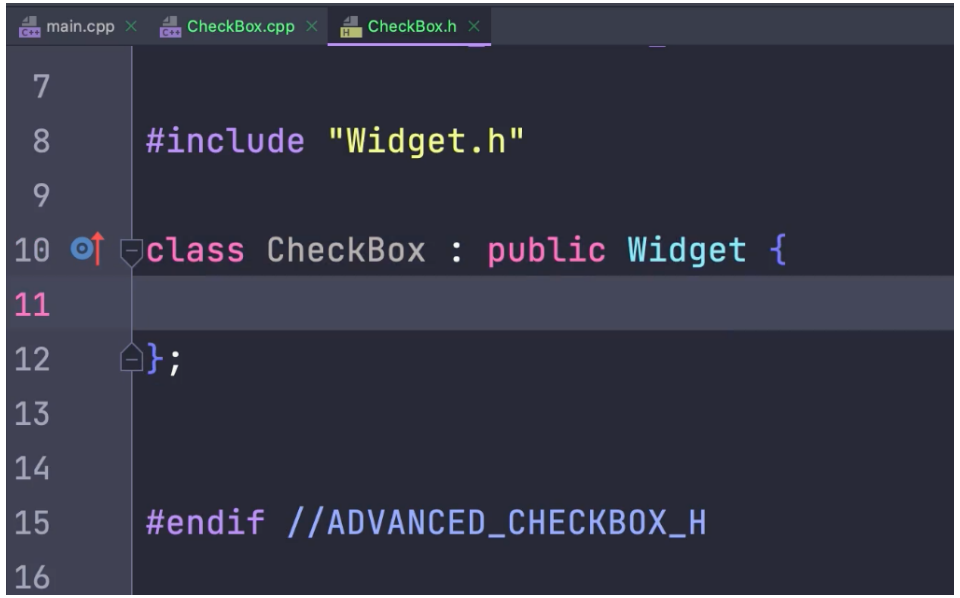
- One of the core principles of OOP.
- Allows us to build applications that can easily be extended.

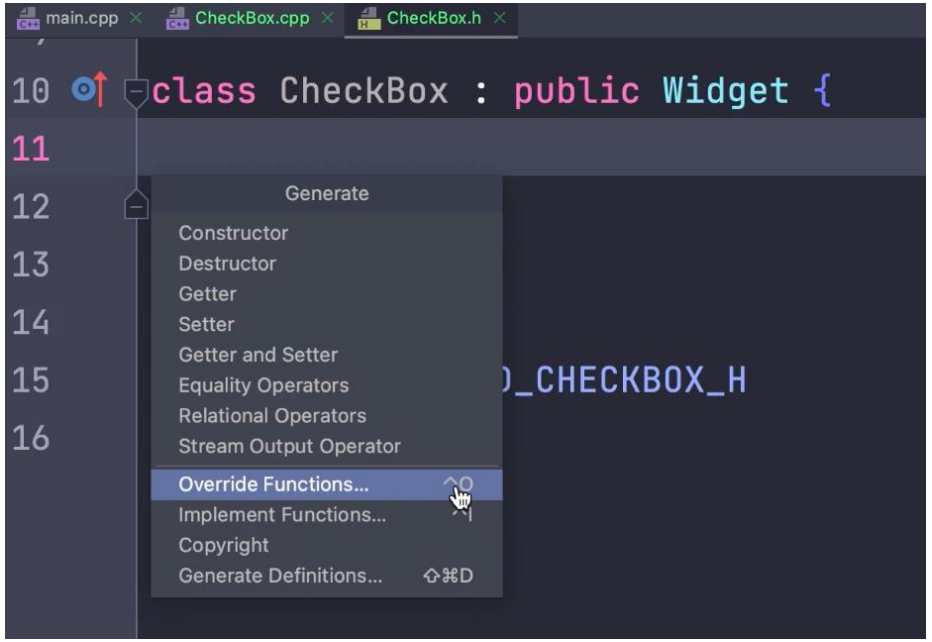
Polymorphism

Many form

Polymorphism

An object taking many forms





- ❑ Press CTRL+ENTER
- ❑ Select Override Functions

Polymorphism

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

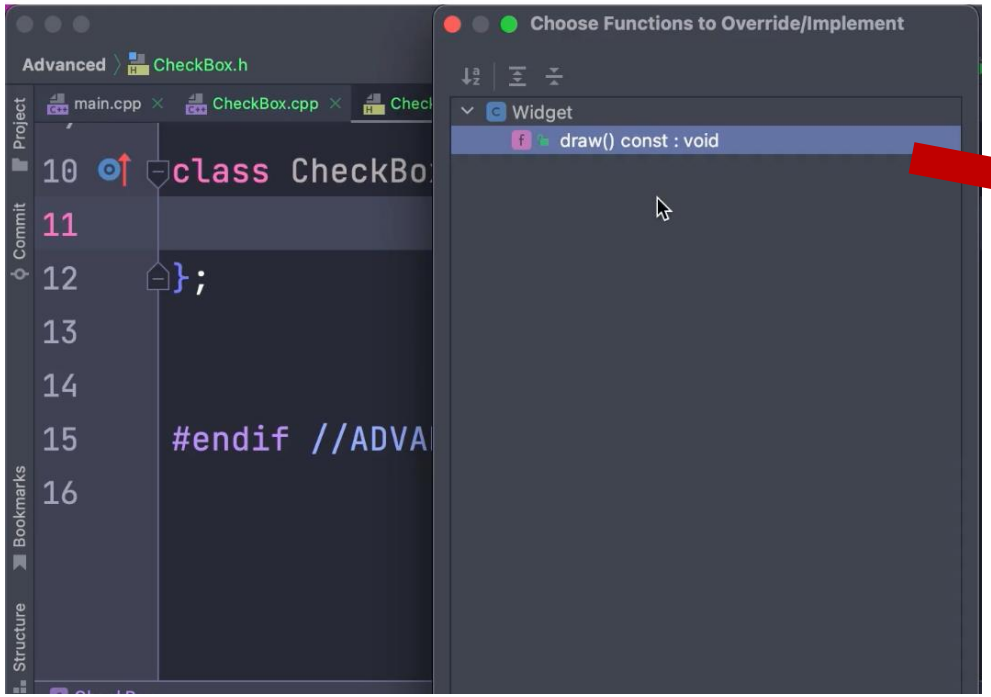
Polymorphism

Many form

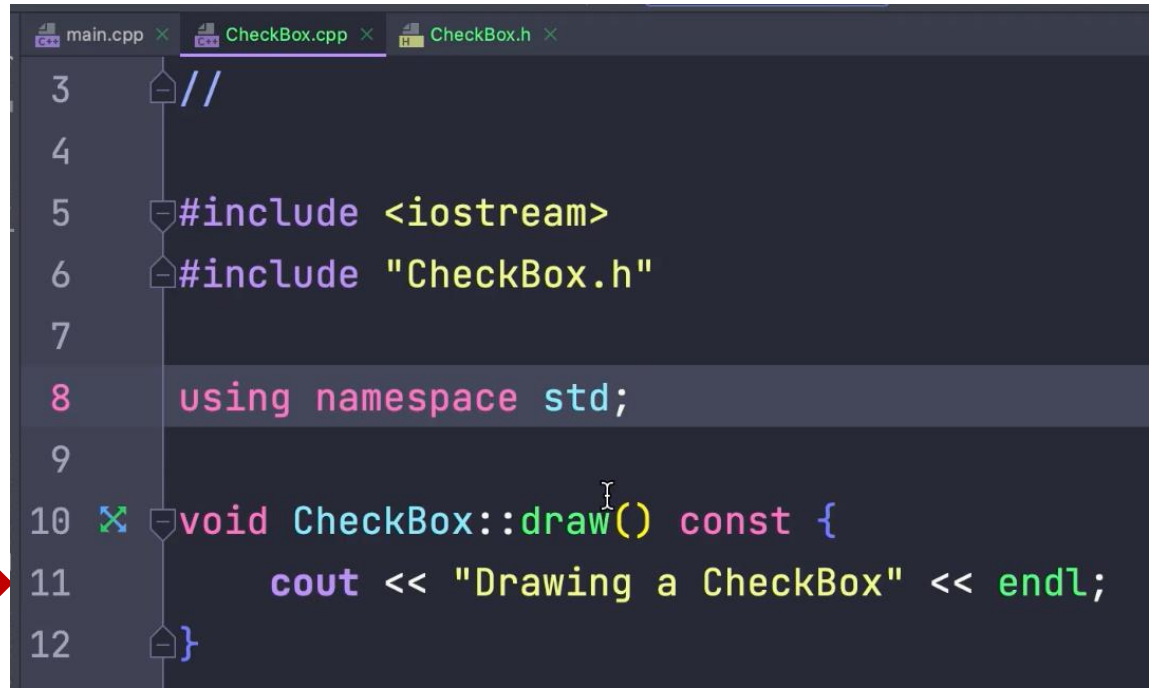
Polymorphism

An object taking many forms

Polymorphism



Select draw method



Update the draw method.

Polymorphism

main.cpp | CheckBox.cpp | CheckBox.h

```cpp
#include <iostream>
#include "CheckBox.h"
#include "TextBox.h"
#include "Widget.h"

using namespace std;

void showWidget(Widget& widget) {
    widget.draw();
}
```

```cpp
int main() {
    TextBox box;
    showWidget( &box);

    CheckBox checkBox;
    showWidget(checkBox);

    return 0;
}
```

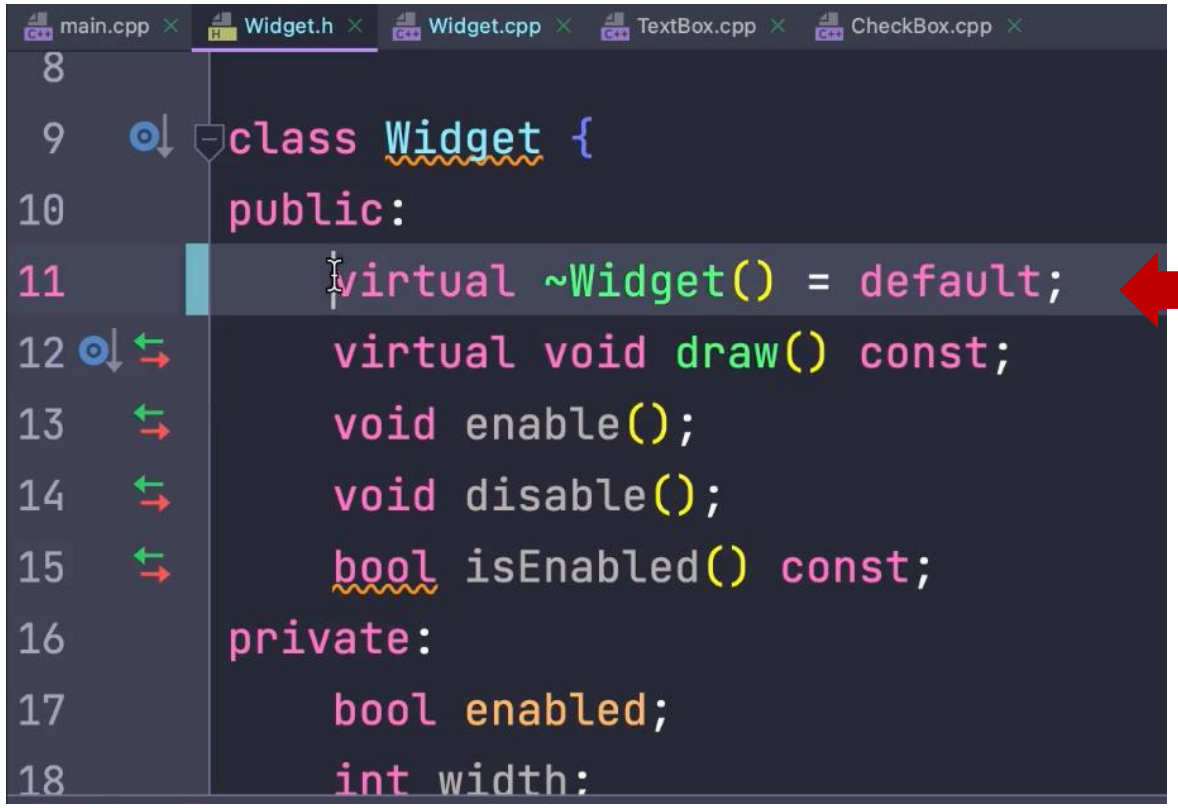
Widget takes the form of a TextBox

Widget takes the form of a CheckBox

```
/Users/moshfeghhamedani/CLionProje
Drawing a TextBox
Drawing a CheckBox

Process finished with exit code 0
```

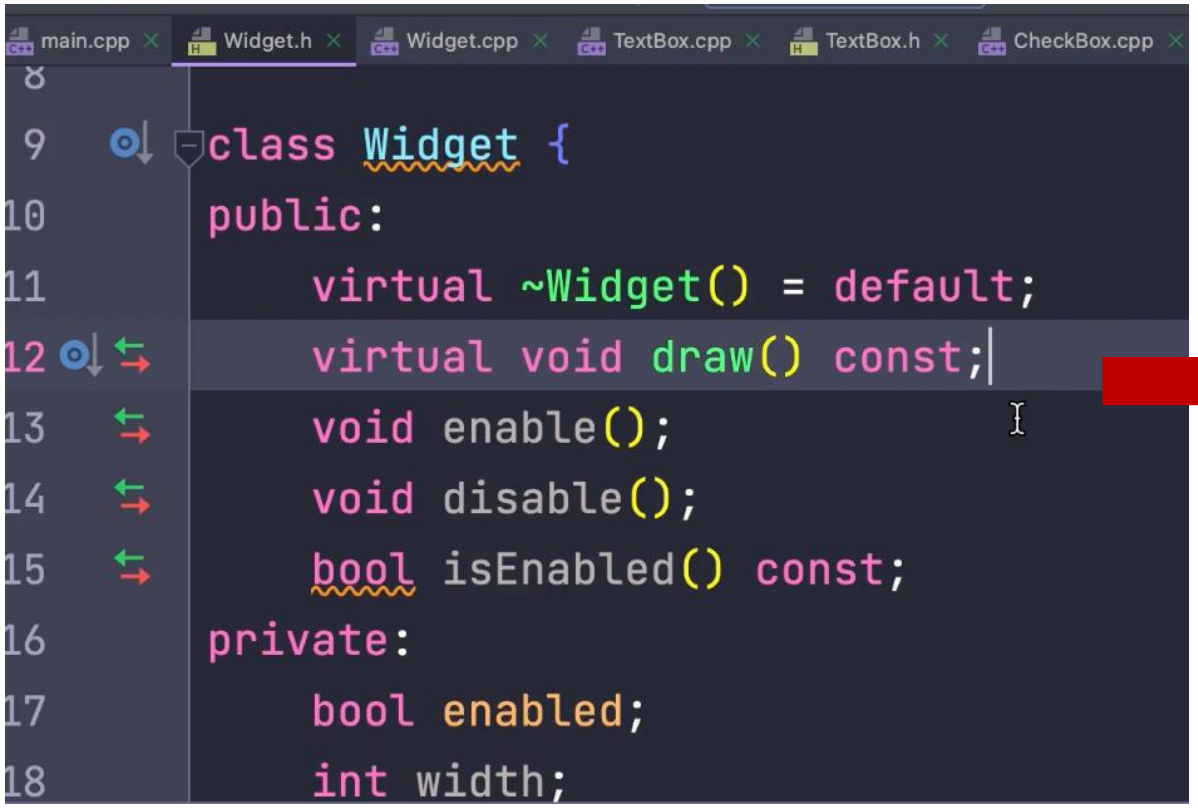
Output of the main function.

Virtual Destructors

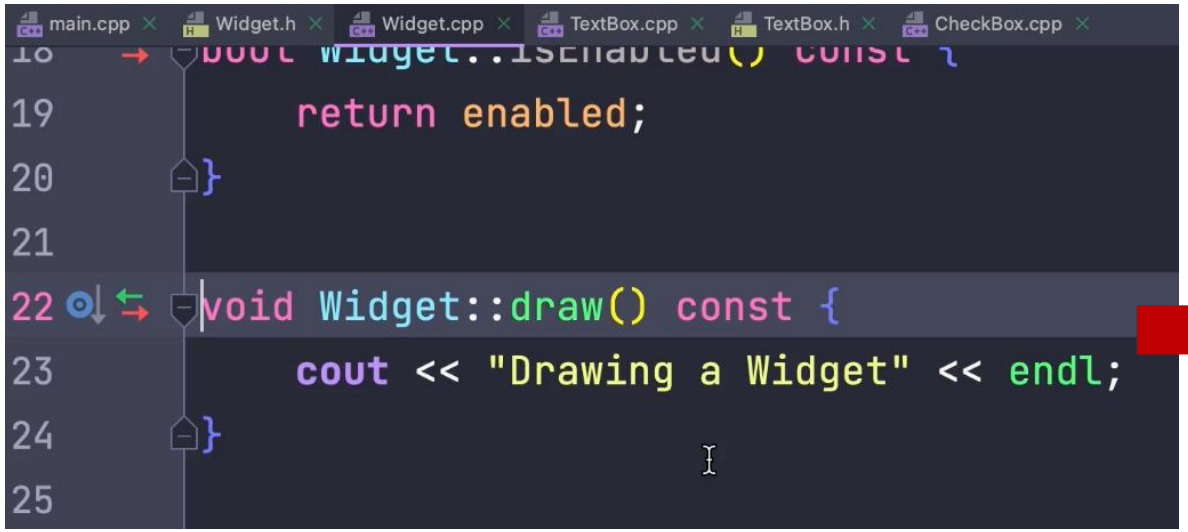


- When we have virtual methods in a class, then we need to create a virtual destructor as well.
- If we have an empty destructor (we still need to declare it in the header file), then we do not need to define it; the compiler will generate for us.

Abstract Classes



- 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.

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;
```

main.cpp | Widget.h | Widget.cpp | TextBox.cpp | TextBox.h | CheckBox.cpp

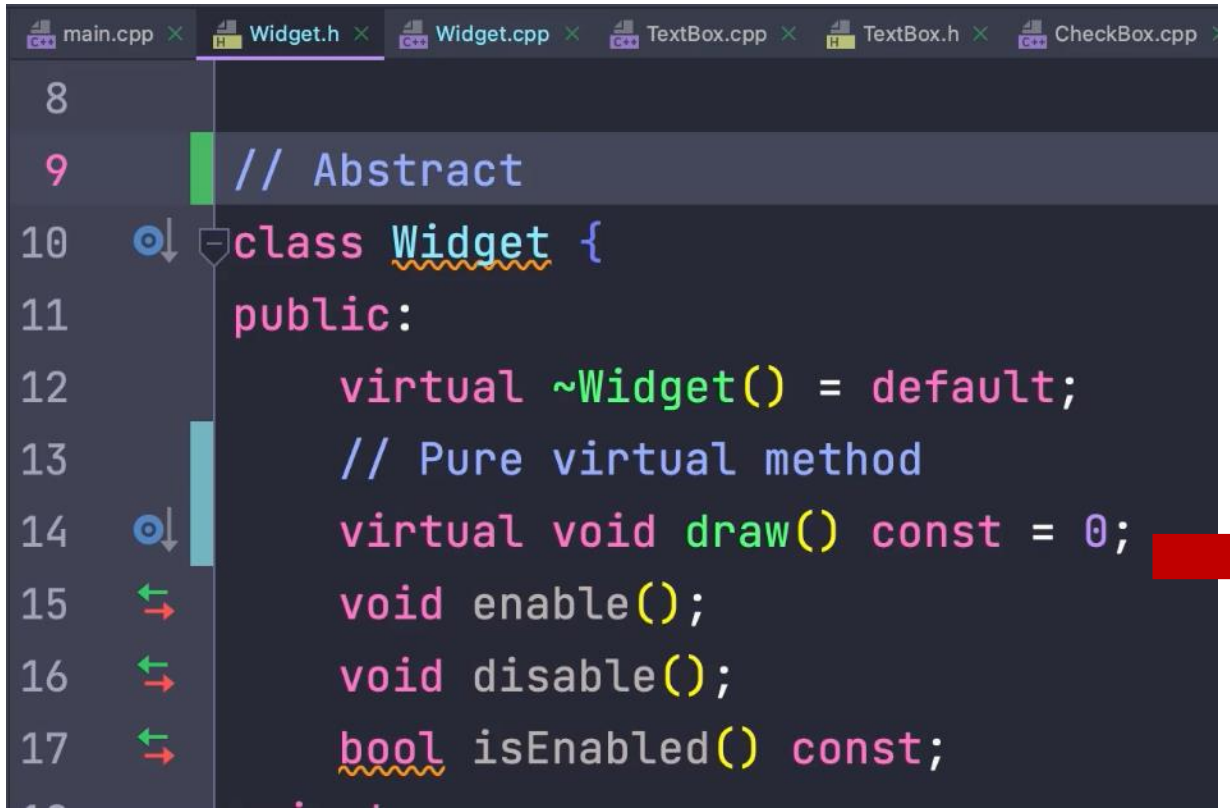
```
19      return enabled;
20  }
21
22  void Widget::draw() const {
23      cout << "Drawing a Widget" << endl;
24  }
25
```

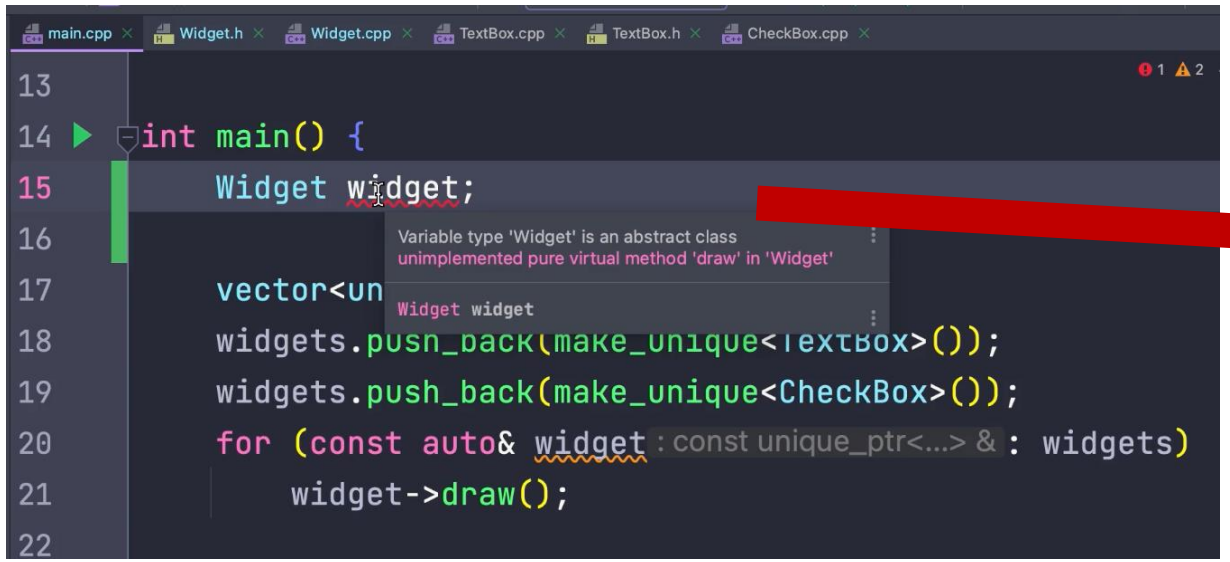
Abstract Classes

- 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.

Abstract Classes

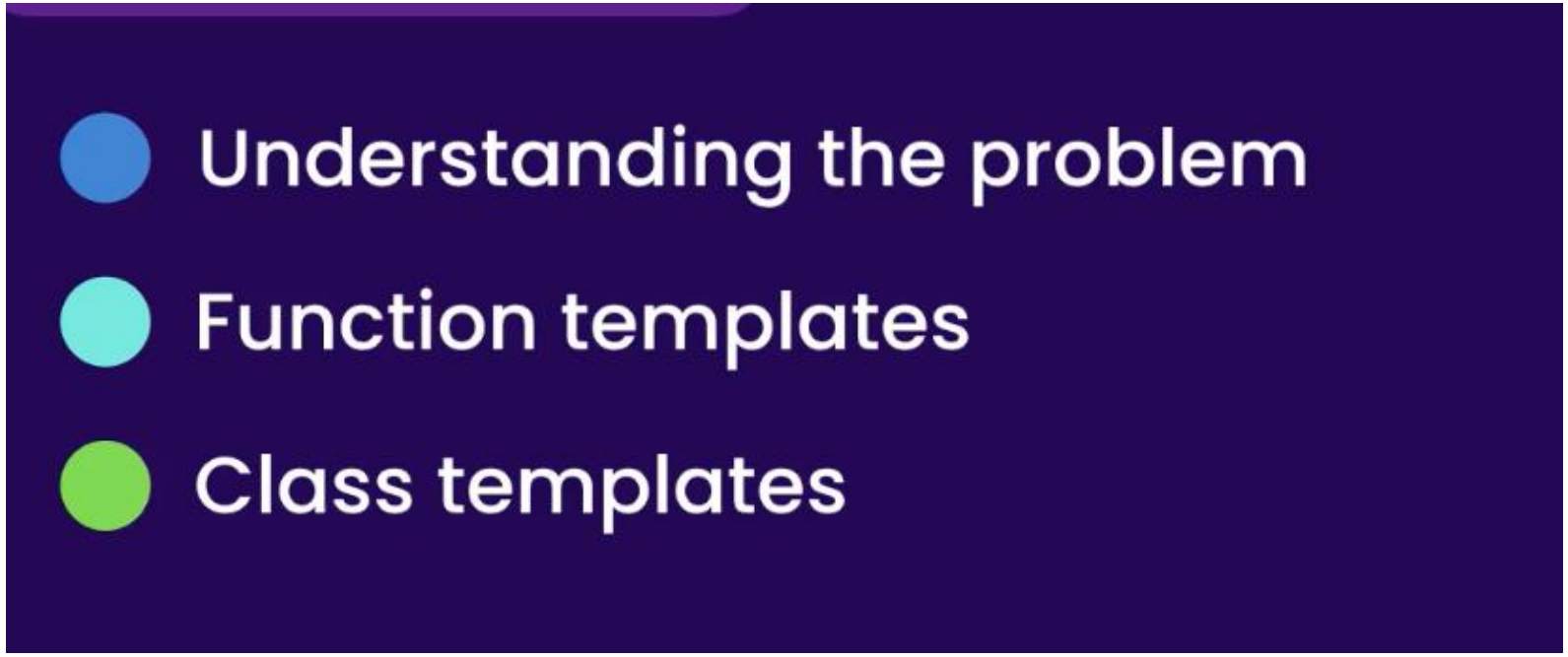




- The draw() method in the Widget class has no actual implementation.
- We declare our virtual method in the Widget class as **pure virtual method**.
- If a class has at least one virtual method, the the class is called "**Abstract Class**".
- Abstract classes cannot be instantiated; they exists purely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

Templates

- Templates help us write generic functions and classes.



Defining a Function Template

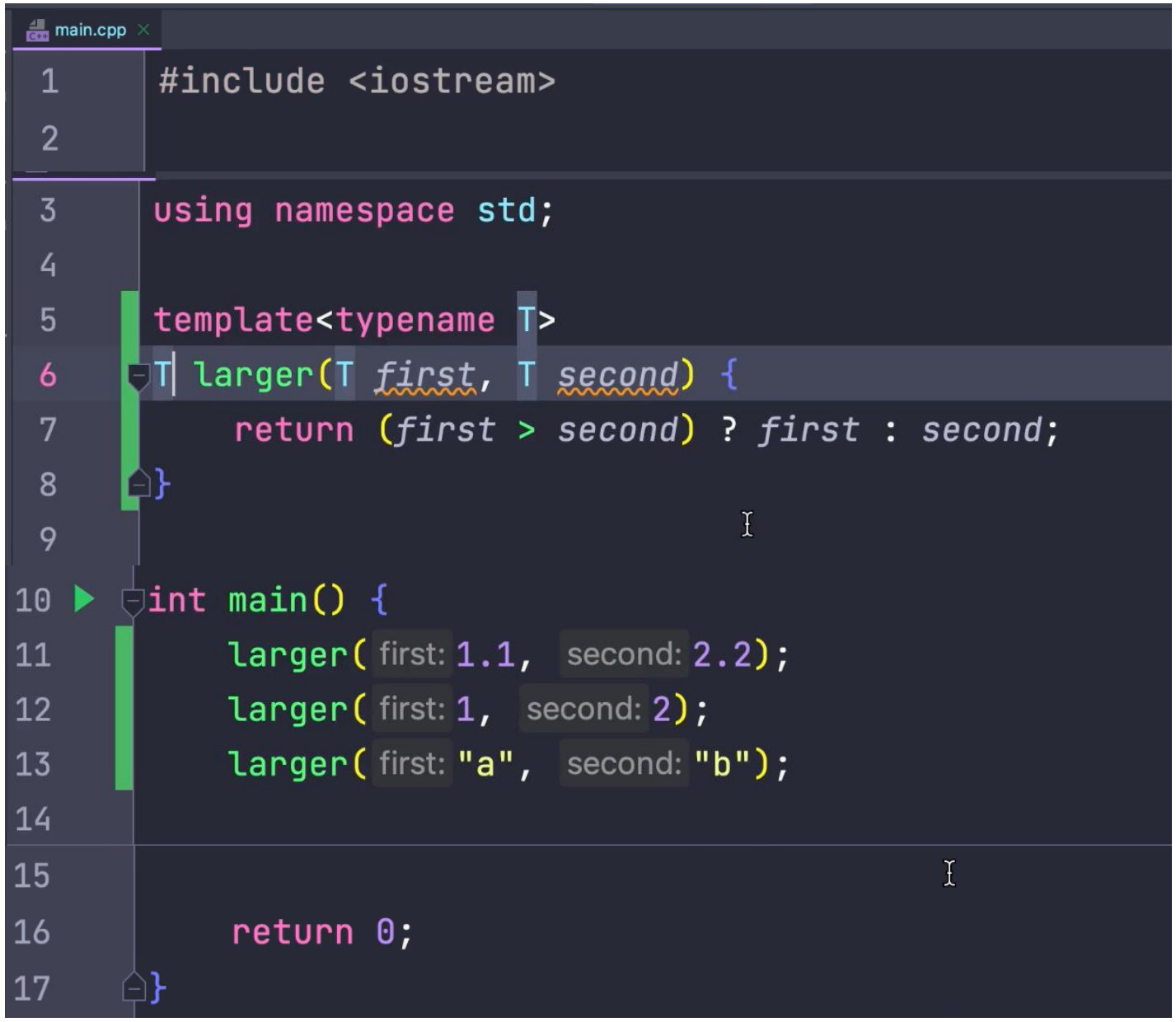
```cpp
#include <iostream>

using namespace std;

int larger(int first, int second) {
    return (first > second) ? first : second;
}

int main() {
    larger( first: 1.1,  second: 2.2);

    return 0;
}
```

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

Defining a Function Template

```cpp
#include <iostream>

using namespace std;

int larger(int first, int second) {
    return (first > second) ? first : second;
}

double larger(double first, double second) {
    return (first > second) ? first : second;
}

int main() {
    larger( first: 1.1,  second: 2.2);

    return 0;
}
```

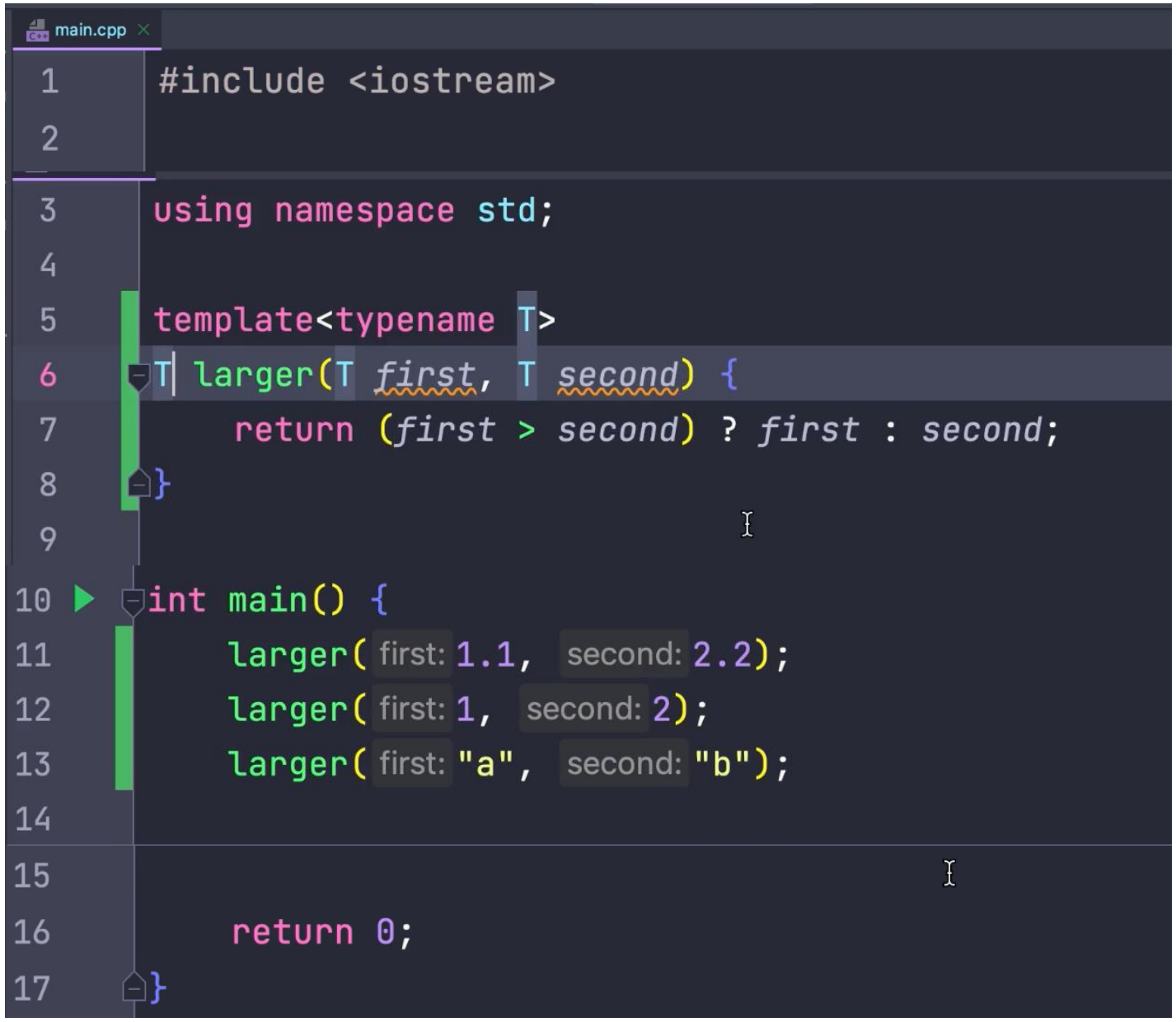
- **Solution:** Use overloading and implement same function with type double.
  - **Problem:** We need to create overloading functions for each type.
  - Code will get large and hard to maintain.



Defining a Function Template

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

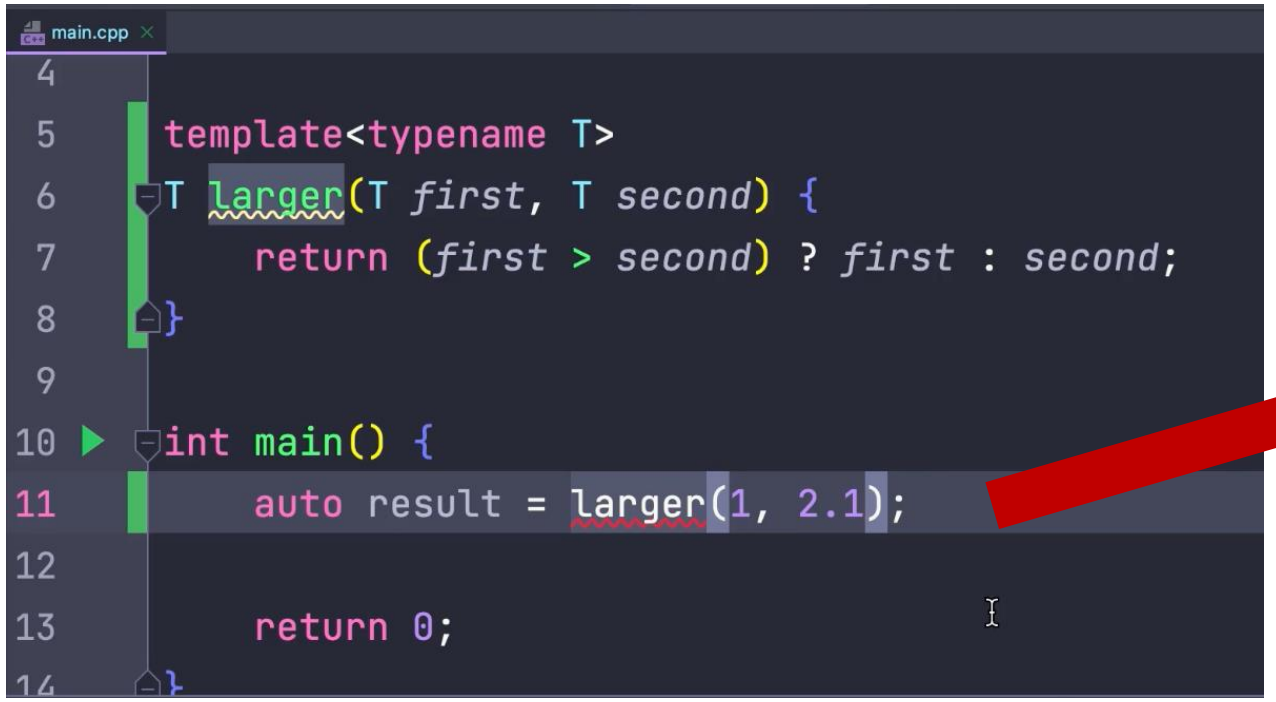
- Instead of **template<typename T>**, **template<class T>** can also be used.



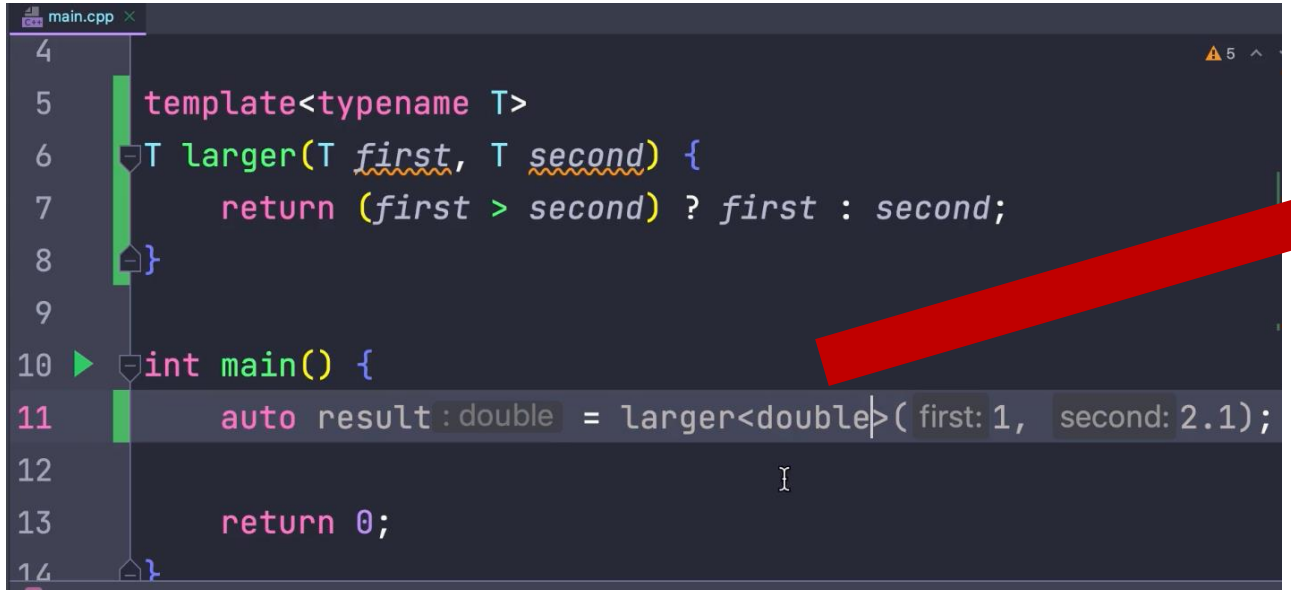
Defining a Function Template

- **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.

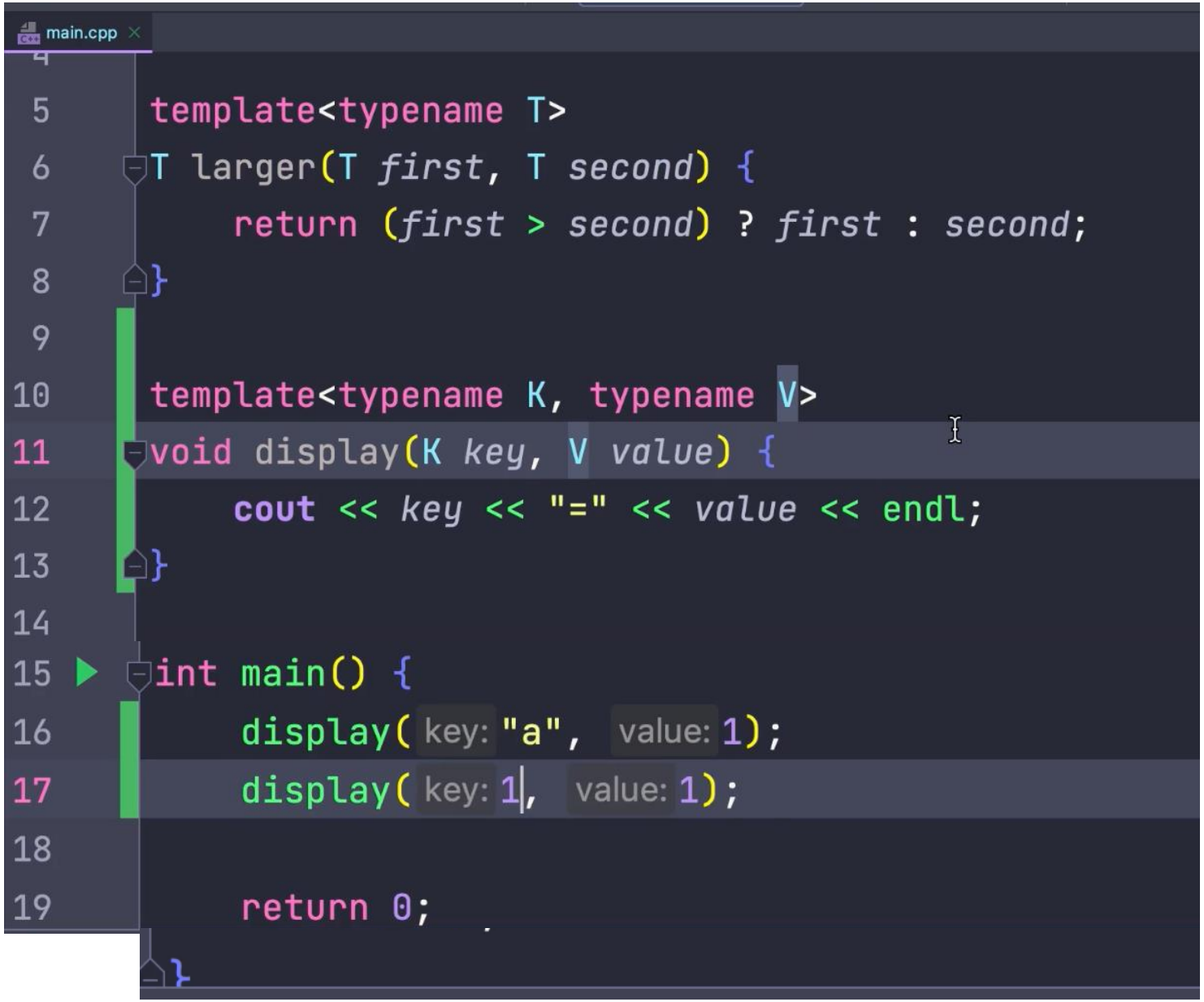
Explicit Type Arguments



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

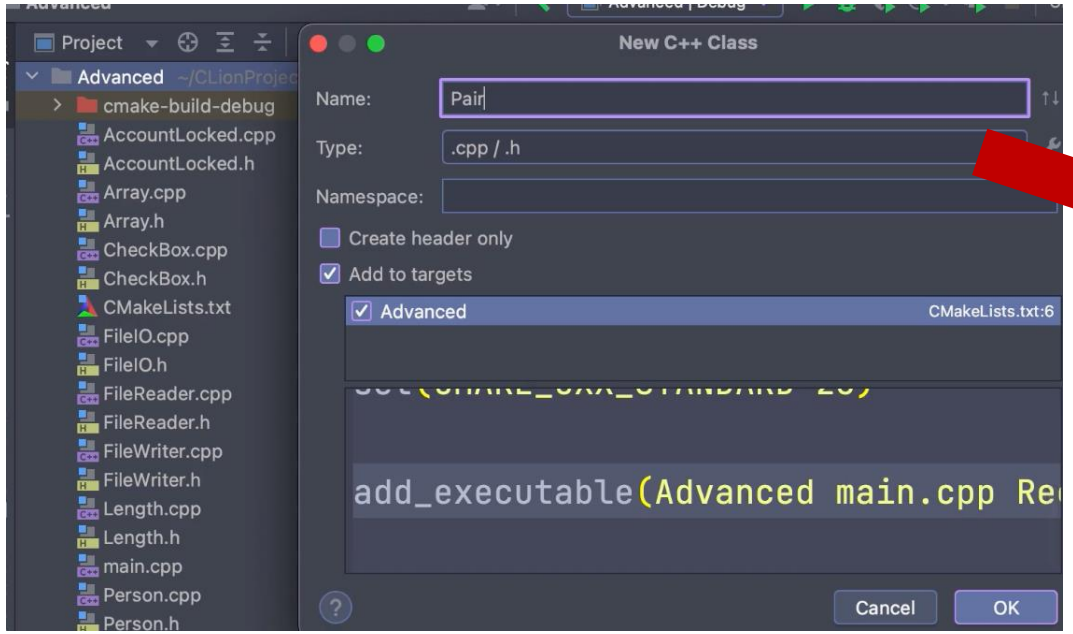


- **Solution:** We explicitly specify the type of the arguments.

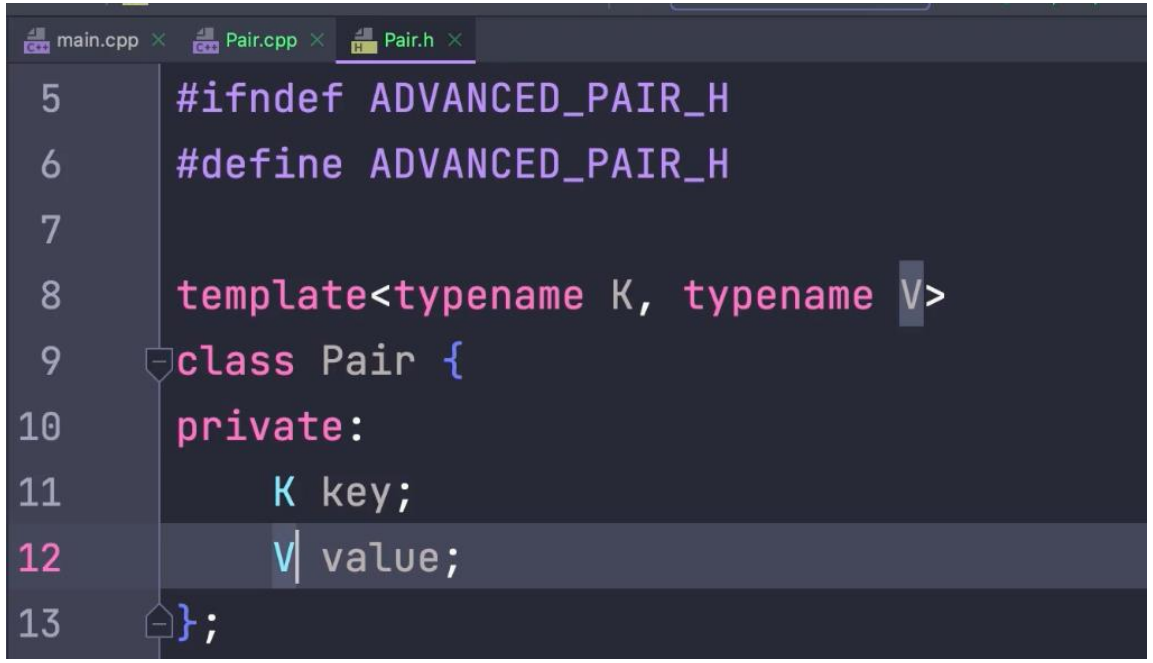


Templates with Multiple Parameters

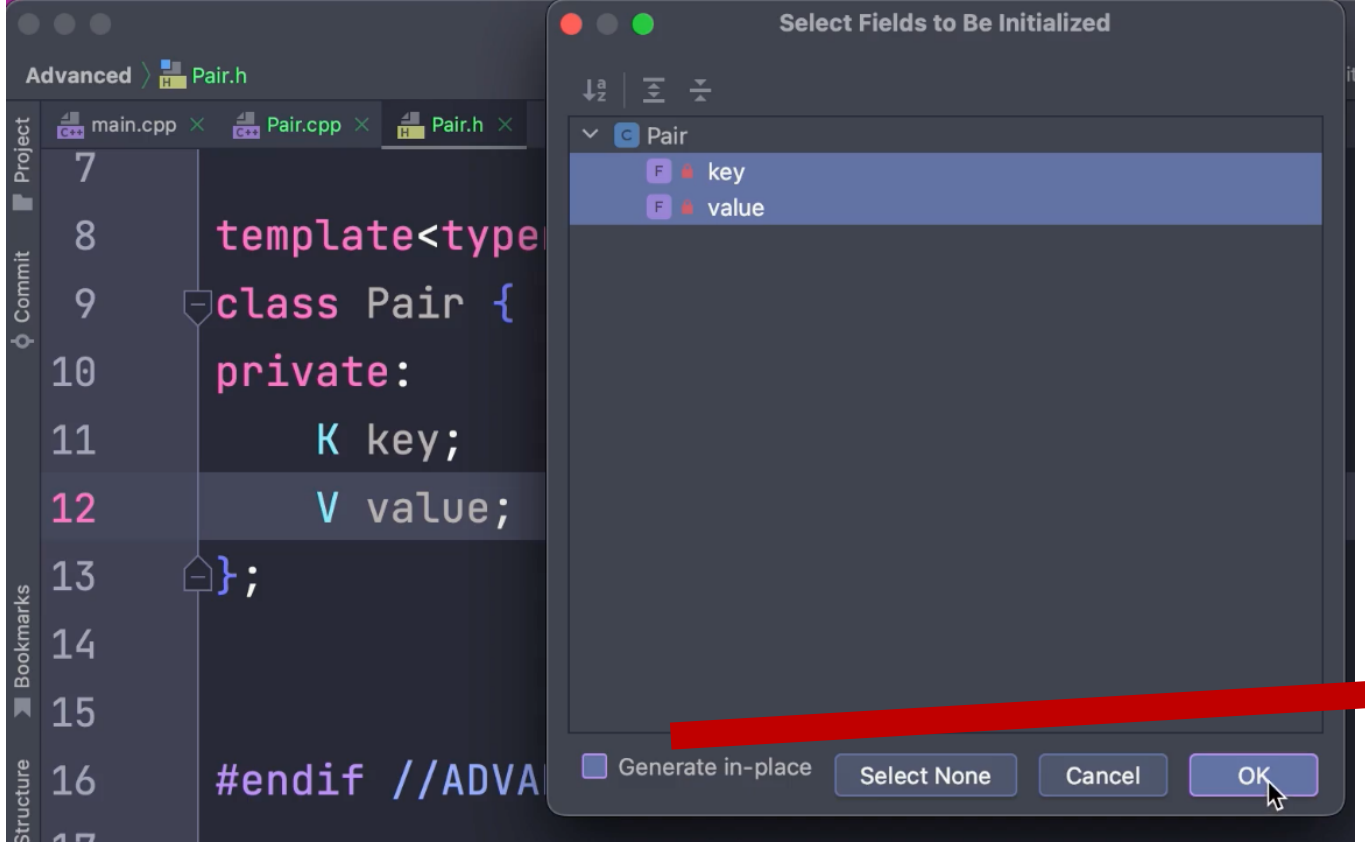
Defining a Class Template



- ❑ Create a new class called Pair.

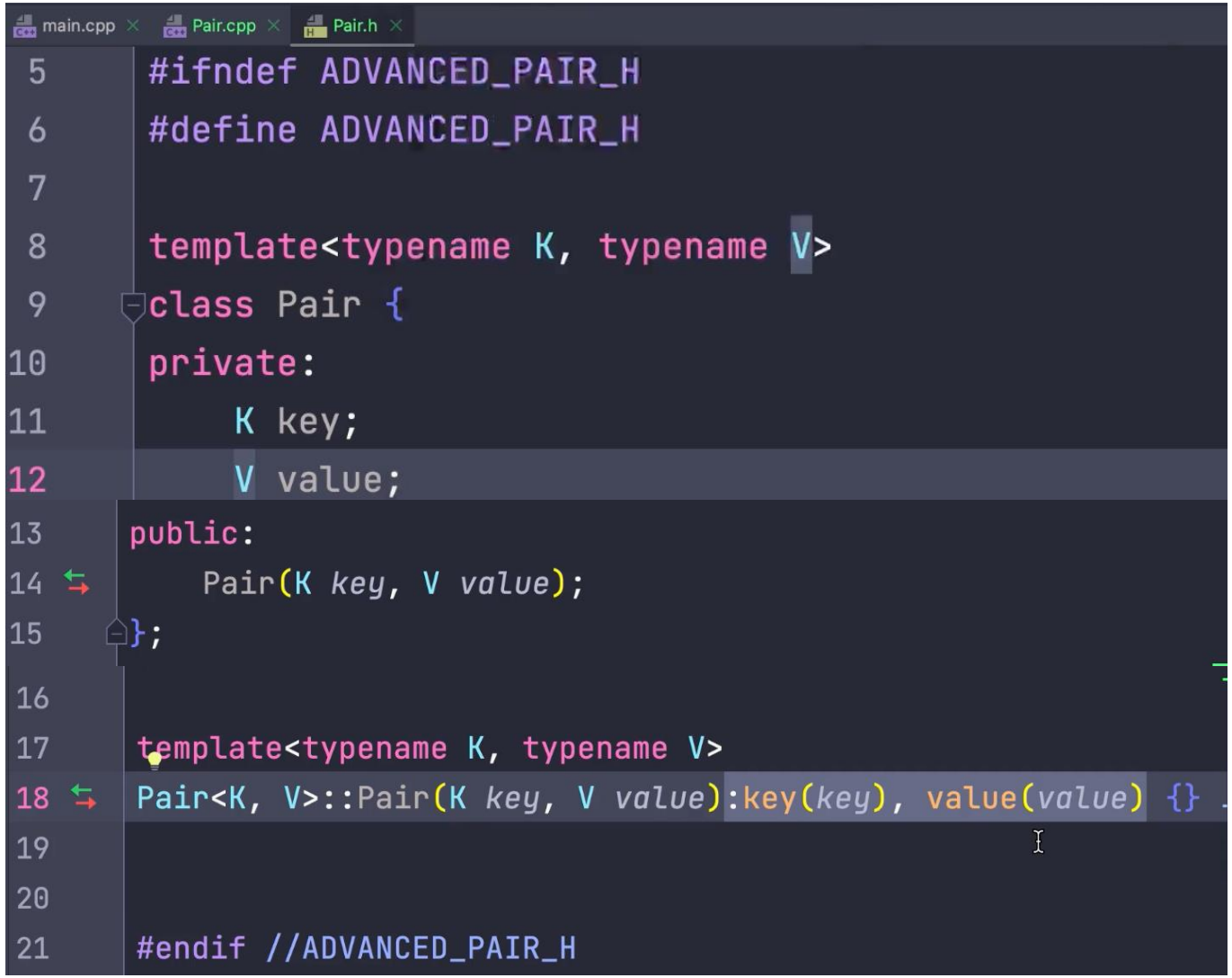


Defining a Class Template

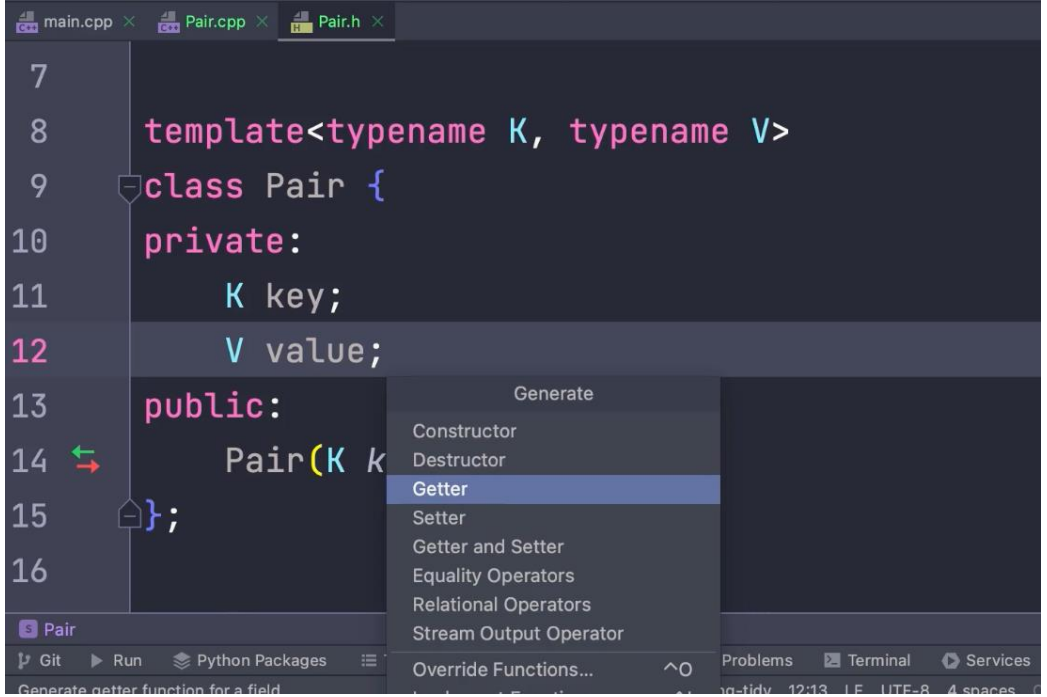


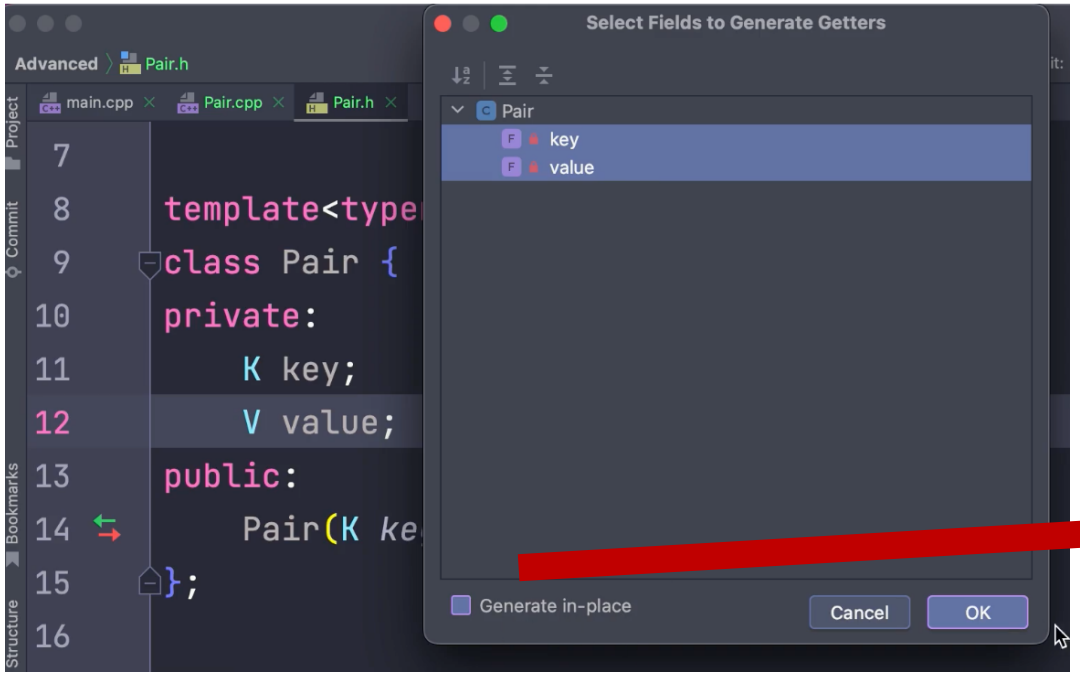
- ❑ Press CTRL-N (COMMAND-N) to bring the generate palate.
- ❑ Uncheck this box.

Defining a Class Template



Defining a Class Template





- Create Getter for key and value.
- CTRL-N, then select Getter.
- Make sure the box is unchecked.

Defining a Class Template

```
main.cpp  Pair.cpp  Pair.h
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 {
```

```
main.cpp  Pair.cpp  Pair.h
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.