

CONST关键字与模板

主讲：陈笑沙

目录

- 11.1 const 关键字
- 11.2 const 与类
- 11.3 为什么需要模板？
- 11.4 如何定义模板？
- 11.5 迭代器

11.1 CONST 关键字

`const` 关键字表示变量、函数或者参数是不可更改的。

11.1 CONST 关键字

```
1 std::vector<int> vec{1, 2, 3};
2 const std::vector<int> c_vec{7, 8};
3 std::vector<int>& ref = vec;
4 const std::vector<int>& c_ref = vec;
5
6 vec.push_back(3);
7 c_vec.push_back(3);
8 ref.push_back(3);
9 c_ref.push_back(3);
```

11.1 CONST 关键字

```
1 std::vector<int> vec{1, 2, 3};
2 const std::vector<int> c_vec{7, 8};
3 std::vector<int>& ref = vec;
4 const std::vector<int>& c_ref = vec;
5
6 vec.push_back(3);
7 c_vec.push_back(3);
8 ref.push_back(3);
9 c_ref.push_back(3);
```

11.1 CONST 关键字

```
1 std::vector<int> vec{1, 2, 3};
2 const std::vector<int> c_vec{7, 8};
3 std::vector<int>& ref = vec;
4 const std::vector<int>& c_ref = vec;
5
6 vec.push_back(3);
7 c_vec.push_back(3);
8 ref.push_back(3);
9 c_ref.push_back(3);
```

11.1 CONST 关键字

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`
- 则该方法可以在 `const` 变量、引用、指针中使用。

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`
- 则该方法可以在 `const` 变量、引用、指针中使用。
- 比如：

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`
- 则该方法可以在 `const` 变量、引用、指针中使用。
- 比如：
 - 对于方法 `void f() const`，只能是常量对象调用

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`
- 则该方法可以在 `const` 变量、引用、指针中使用。
- 比如：
 - 对于方法 `void f() const`，只能是常量对象调用
 - 对于方法 `void f()`，常量对象无法调用

11.1 CONST 关键字

- 如果一个类中，某个方法被标明为 `const`
- 则该方法可以在 `const` 变量、引用、指针中使用。
- 比如：
 - 对于方法 `void f() const`，只能是常量对象调用
 - 对于方法 `void f()`，常量对象无法调用
- 成员函数（方法）可以通过常量与非常量进行区分

11.1 CONST 关键字

```
1 class vector {
2     ...
3     iterator begin() {
4         return iterator(data);
5     }
6     iterator end() {
7         return iterator(data + m_size);
8     }
9     const_iterator begin() const {
10        return const_iterator(data);
11    }
12    const_iterator end() const {
13        return const_iterator(data + m_size);
14    }
15    const_iterator cbegin() const {
```

11.1 CONST 关键字

```
1 class vector {
2     ...
3     iterator begin() {
4         return iterator(data);
5     }
6     iterator end() {
7         return iterator(data + m_size);
8     }
9     const_iterator begin() const {
10        return const_iterator(data);
11    }
12    const_iterator end() const {
13        return const_iterator(data + m_size);
14    }
15    const_iterator cbegin() const {
```

11.1 CONST 关键字

```
1  return iterator(data);  
5  }  
6  iterator end() {  
7      return iterator(data + m_size);  
8  }  
9  const_iterator begin() const {  
10     return const_iterator(data);  
11 }  
12 const_iterator end() const {  
13     return const_iterator(data + m_size);  
14 }  
15 const_iterator cbegin() const {  
16     return const_iterator(data);  
17 }  
18 const_iterator cend() const {  
19     return const_iterator(data + m_size);  
20 }
```


11.1 CONST 关键字

```
8     }
9     const_iterator begin() const {
10         return const_iterator(data);
11     }
12     const_iterator end() const {
13         return const_iterator(data + m_size);
14     }
15     const_iterator cbegin() const {
16         return const_iterator(data);
17     }
18     const_iterator cend() const {
19         return const_iterator(data + m_size);
20     }
21     ...
22 };
23 const_iterator cbegin() const {
```

11.1 CONST 关键字

使用 const 的好处

```
1 void f(int x, int y) {  
2     if ((x == 2 && y == 3) || (x == 1))  
3         cout << 'a' << endl;  
4     if (y = -1) // in most time, this is not we want  
5         cout << 'b' << endl;  
6     if ((x == 3) && (y == 2 * x))  
7         cout << 'c' << endl;  
8 }
```

11.1 CONST 关键字

使用 const 的好处

```
1 void f(const int x, const int y) {  
2     if ((x == 2 && y == 3) || (x == 1))  
3         cout << 'a' << endl;  
4     if (y = -1) // compile error!  
5         cout << 'b' << endl;  
6     if ((x == 3) && (y == 2 * x))  
7         cout << 'c' << endl;  
8 }
```

11.2 CONST 与类

11.2 CONST 与类

```
1 class Student {
2 public:
3     std::string getName();
4     void setName(const std::string& name);
5     int getAge();
6     void setAge(int age);
7 private:
8     std::string name;
9     int age;
10 };
```

11.2 CONST 与类

```
1 std::ostream& operator<<(std::ostream& out, const Student& stu)
2     out << stu.getName << ": " << stu.getAge() << endl;
3 }
```

11.2 CONST 与类

```
1 std::ostream& operator<<(std::ostream& out, const Student& stu)
2     out << stu.getName << ": " << stu.getAge() << endl;
3 }
```

编译错误!

11.2 CONST 与类

```
1 std::ostream& operator<<(std::ostream& out, const Student& stu)
2     out << stu.getName << ": " << stu.getAge() << endl;
3 }
```

编译器不知道 getName() 与 getAge() 方法不会改变 Student

11.2 CONST 与类

11.2 CONST 与类

```
1 class Student {  
2 public:  
3     std::string getName() const;  
4     void setName(const std::string& name);  
5     int getAge() const;  
6     void setAge(int age);  
7 private:  
8     std::string name;  
9     int age;  
10 };
```

11.2 CONST 与类

相关成员函数实现时，也需要加 `const`

```
1 std::string Student::getName() const {  
2     return name;  
3 }  
4 void Student::setName(const std::string& name) {  
5     this->name = name;  
6 }
```

11.3 为什么需要模板？

11.3 为什么需要模板？

```
1 class IntVector {  
2     private:  
3         int* ptr;  
4         unsigned int size;  
5     public:  
6         ...  
7 };
```

11.3 为什么需要模板？

```
1 class IntVector {  
2     private:  
3         int* ptr;  
4         unsigned int size;  
5     public:  
6         ...  
7 };
```

需要为不同的元素类型创建不同的容器类。

11.3 为什么需要模板？

```
1 class Vector {  
2     using T = int;  
3     private:  
4         T* ptr;  
5         unsigned int size;  
6     public:  
7         ...  
8 };
```

11.3 为什么需要模板？

```
1 class Vector {  
2     using T = int;  
3     private:  
4         T* ptr;  
5         unsigned int size;  
6     public:  
7         ...  
8 };
```

每次更改内容，需要改 Vector 里面的代码。

11.3 为什么需要模板？

```
1 #define VECTOR_DEFINE(T) \  
2 typedef T##Vector { \  
3 private:\  
4     T* data; \  
5     size_t size; \  
6     size_t capacity; \  
7     ... \  
8 }; \  

```

11.3 为什么需要模板？

```
1 #define VECTOR_DEFINE(T) \  
2 typedef T##Vector { \  
3 private:\  
4     T* data; \  
5     size_t size; \  
6     size_t capacity; \  
7     ... \  
8 }; \  

```

- 容易出错

11.3 为什么需要模板？

```
1 #define VECTOR_DEFINE(T) \  
2 typedef T##Vector { \  
3 private:\  
4     T* data; \  
5     size_t size; \  
6     size_t capacity; \  
7     ... \  
8 }; \  

```

- 容易出错
- 编程工具不会提供语法检查

11.3 为什么需要模板？

11.3 为什么需要模板？

- 对于基础数据类型，有可能通过创建 `IntVector` `DoubleVector` 等遍历。

11.3 为什么需要模板？

- 对于基础数据类型，有可能通过创建 `IntVector` `DoubleVector` 等遍历。
- 但是如果想要存储用户自定义的类，则不可能穷举所有情况。

11.3 为什么需要模板？

- 对于基础数据类型，有可能通过创建 `IntVector` `DoubleVector` 等遍历。
- 但是如果想要存储用户自定义的类，则不可能穷举所有情况。
- 需要有一种工具，可以由用户自定义容器类的元素类型。

11.4 如何定义模板？

11.4 如何定义模板？

- **模板类**是一个以**类型参数化**为核心的类定义模式。

11.4 如何定义模板？

- **模板类**是一个以**类型参数化**为核心的类定义模式。
- 通过将类中的**某些**具体类型（如 `int`、`string` 等）替换为占位符类型参数（如 `T`），可以在不修改类逻辑的前提下，用同一套代码支持多种数据类型。

11.4 如何定义模板？

- **模板类**是一个以**类型参数化**为核心的类定义模式。
- 通过将类中的**某些**具体类型（如 `int`、`string` 等）替换为占位符类型参数（如 `T`），可以在不修改类逻辑的前提下，用同一套代码支持多种数据类型。
- 编译器会在编译时根据实际类型实例化出具体的类。

11.4 如何定义模板？

```
1 class MyPair {  
2 public:  
3     int getFirst() const;  
4     int getSecond() const;  
5  
6     void setFirst(int f);  
7     void setSecond(int s);  
8  
9 private:  
10    int first;  
11    int second;  
12 };
```

11.4 如何定义模板？

```
1  template<typename First, typename Second>
2  class MyPair {
3  public:
4      First getFirst() const;
5      Second getSecond() const;
6
7      void setFirst(First f);
8      void setSecond(Second s);
9
10 private:
11     First first;
12     Second second;
13 };
```

11.4 如何定义模板？

```
1  template<typename First, typename Second>
2  class MyPair {
3  public:
4      First getFirst() const;
5      Second getSecond() const;
6
7      void setFirst(First f);
8      void setSecond(Second s);
9
10 private:
11     First first;
12     Second second;
13 };
```

- First 和 Second 是泛型类型，可以是任意类型！

11.4 如何定义模板？

- `typename` 也可以被写作 `class`
- 大部分情况两者等价
- 更推荐使用 `typename`，语义更加明确

11.4 如何定义模板？

- 除了类以外，还有模板函数

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void swapValues(T &a, T &b) {
6     T temp = a;
7     a = b;
8     b = temp;
9 }
10
11 template <typename T>
12 void print(const string &prompt, const T &a, const T &b) {
13     cout << prompt << ": " << a << ", " << b << endl;
14 }
15
```


11.4 如何定义模板？

- 除了类以外，还有模板函数

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void swapValues(T &a, T &b) {
6     T temp = a;
7     a = b;
8     b = temp;
9 }
10
11 template <typename T>
12 void print(const string &prompt, const T &a, const T &b) {
13     cout << prompt << ": " << a << ", " << b << endl;
14 }
15
```

11.4 如何定义模板？

- 除了类以外，还有模板函数

```
5 void swapValues(T &a, T &b) {  
6     T temp = a;  
7     a = b;  
8     b = temp;  
9 }  
10  
11 template <typename T>  
12 void print(const string &prompt, const T &a, const T &b) {  
13     cout << prompt << ": " << a << ", " << b << endl;  
14 }  
15  
16 int main() {  
17     int x = 10, y = 20;  
18     print("before swap", x, y);  
19     swapValues(x, y);  
20     print("after swap", x, y);  
}
```

11.4 如何定义模板？

- 除了类以外，还有模板函数

```
12 void print(const string &prompt, const T &a, const T &b) {  
13     cout << prompt << ": " << a << ", " << b << endl;  
14 }  
15  
16 int main() {  
17     int x = 10, y = 20;  
18     print("before swap", x, y);  
19     swapValues(x, y);  
20     print("after swap", x, y);  
21     cout << endl;  
22  
23     double a = 3.14, b = 2.718;  
24     print("before swap", a, b);  
25     swapValues(a, b);  
26     print("after swap", a, b);  
27 }
```

11.4 如何定义模板？

- 除了类以外，还有模板函数

```
18  print("before swap", x, y);
19  swapValues(x, y);
20  print("after swap", x, y);
21  cout << endl;
22
23  double a = 3.14, b = 2.718;
24  print("before swap", a, b);
25  swapValues(a, b);
26  print("after swap", a, b);
27  cout << endl;
28
29  string s1 = "Hello", s2 = "World";
30  print("before swap", s1, s2);
31  swapValues(s1, s2);
32  print("after swap", s1, s2);
33
```

11.4 如何定义模板？

- 除了类以外，还有模板函数

```
21 cout << endl;  
22  
23 double a = 3.14, b = 2.718;  
24 print("before swap", a, b);  
25 swapValues(a, b);  
26 print("after swap", a, b);  
27 cout << endl;  
28  
29 string s1 = "Hello", s2 = "World";  
30 print("before swap", s1, s2);  
31 swapValues(s1, s2);  
32 print("after swap", s1, s2);  
33  
34 return 0;  
35 }
```

11.4 如何定义模板？

11.4 如何定义模板？

- 编译器需要根据模板的定义，生成对应不同类型的类并编译。

11.4 如何定义模板？

- 编译器需要根据模板的定义，生成对应不同类型的类并编译。
- 因此，不要将模板类的定义放在源文件中。

11.4 如何定义模板？

- 编译器需要根据模板的定义，生成对应不同类型的类并编译。
- 因此，不要将模板类的定义放在源文件中。
- 而是需要与模板类的声明一起放在头文件中。

11.4 如何定义模板？

- 编译器需要根据模板的定义，生成对应不同类型的类并编译。
- 因此，不要将模板类的定义放在源文件中。
- 而是需要与模板类的声明一起放在头文件中。
- 模板函数也一样。

11.4 如何定义模板？

- 模板是一个非常复杂的 C++ 编程分支，在确认熟悉之前，尽量不要亲自实现复杂的模板库。
- 尝试编译以下代码：

```
1 #include <algorithm>
2 #include <vector>
3
4 int main() {
5     int a;
6     std::vector<std::vector<int>> v;
7     std::vector<std::vector<int>>::const_iterator it =
8         std::find(v.begin(), v.end(), a);
9     return 0;
10 }
```

11.4 如何定义模板？

- 模板还有非常多的内容：

11.4 如何定义模板？

- 模板还有非常多的内容：
- 甚至是一门图灵完备的语言

11.4 如何定义模板？

- 模板还有非常多的内容：
- 甚至是一门图灵完备的语言
- 内容过于繁杂，本课程不深入讨论

11.4 如何定义模板？

- 模板还有非常多的内容：
- 甚至是一门图灵完备的语言
- 内容过于繁杂，本课程不深入讨论
- 深入学习可以参考：[C++ Templates 2nd](#)

11.4 如何定义模板？

- 模板还有非常多的内容：
- 甚至是一门图灵完备的语言
- 内容过于繁杂，本课程不深入讨论
- 深入学习可以参考：[Cpp Templates 2nd](#)
- 推荐阅读：《C++20高级编程》（罗能，机械工业出版社）

11.4 如何定义模板？

- 模板还有非常多的内容：
- 甚至是一门图灵完备的语言
- 内容过于繁杂，本课程不深入讨论
- 深入学习可以参考：[Cpp Templates 2nd](#)
- 推荐阅读：《C++20高级编程》（罗能，机械工业出版社）
- 在完全掌握模板之前，不建议**过度**使用

11.5 迭代器

11.5 迭代器

- 如果想要给我们自定义的容器添加 `foreach` 语法支持：

11.5 迭代器

- 如果想要给我们自定义的容器添加 `foreach` 语法支持：
- 即： `for(auto& elem : myContainer)`

11.5 迭代器

- 如果想要给我们自定义的容器添加 `foreach` 语法支持：
- 即：`for(auto& elem : myContainer)`
- 则需要实现**迭代器**

11.5 迭代器

11.5 迭代器

- 迭代器的作用：

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。
- 迭代器的统一接口：

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。
- 迭代器的统一接口：
 - 容器： `begin()`， `end()`

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。
- 迭代器的统一接口：
 - 容器：begin(), end()
 - 迭代器：++, *, ≠

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。
- 迭代器的统一接口：
 - 容器：begin(), end()
 - 迭代器：++, *, ≠
 - 迭代器类型：value_type、difference_type、pointer、reference

11.5 迭代器

- 迭代器的作用：
 - 统一遍历方法，更改容器类型不用更改遍历代码。
 - 免去下标检查，避免越界错误。
- 迭代器的统一接口：
 - 容器：begin(), end()
 - 迭代器：++, *, ≠
 - 迭代器类型：value_type、difference_type、pointer、reference
 - 迭代器范畴：iterator_category

11.5 迭代器

11.5 迭代器

- `iterator_category` 包含:

11.5 迭代器

- `iterator_category` 包含:
 - `std::input_iterator`: `istream` 独有

11.5 迭代器

- `iterator_category` 包含:
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有

11.5 迭代器

- `iterator_category` 包含:
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有
 - `std::forward_iterator`

11.5 迭代器

- `iterator_category` 包含:
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有
 - `std::forward_iterator`
 - `std::bidirectional_iterator`

11.5 迭代器

- `iterator_category` 包含:
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有
 - `std::forward_iterator`
 - `std::bidirectional_iterator`
 - `std::random_access_iterator`

11.5 迭代器

- `iterator_category` 包含：
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有
 - `std::forward_iterator`
 - `std::bidirectional_iterator`
 - `std::random_access_iterator`
- 所有相关定义，在 `<iterator>` 头文件中

11.5 迭代器

- `iterator_category` 包含：
 - `std::input_iterator`: `istream` 独有
 - `std::output_iterator`: `ostream` 独有
 - `std::forward_iterator`
 - `std::bidirectional_iterator`
 - `std::random_access_iterator`
- 所有相关定义，在 `<iterator>` 头文件中
- 由于 C++ 不支持接口，因此编译器不提供编译期语法检查。正确性由程序员保证。

11.5 迭代器

`/example/lec11/vector`

11.5 迭代器

Iterator 类型	是否可以自增自减?	是否可以改变指向的内容?
iterator	是	是
const_iterator	是	否
const iterator	否	是
const const_iterator	否	否

11.5 迭代器

Iterator 类型	与指针类型对比
iterator	int*
const_iterator	const int*
const iterator	int const*
const const_iterator	const int const*