lecture9

what：Template classes
what more:template classes/const correctness

what templates:

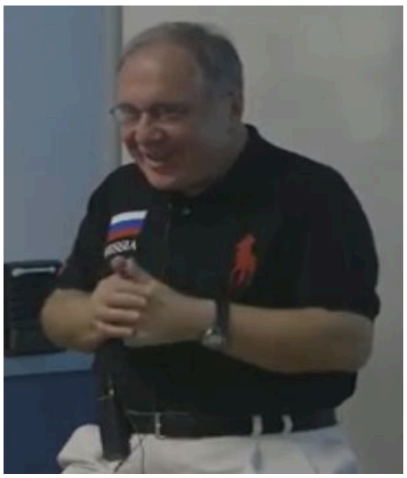
引入：设计一个vector
用类设计一个vector

Recall: IntVector

```
// Implements a sequence of strings
class IntVector {
public:
    IntVector();
    ~IntVector();

    size_t size();
    bool empty();

    void push_back(const int& elem);
    int& operator[](size_t index);
};
```

但是这个只适用于int,其他数据类型还得重写

You realize you need to handle...



Alexander Stepanov
Creator of STL

Vector of doubles?

Vector of std::string?

Vector of vector of strings?

Vector of custom type I haven't even thought of yet?

把这些不同数据类型的“类”组合起来就是模板（templates）

What are templates?

```
class IntVector {
};
```

```
class DoubleVector {
};
```

```
class StringVector {
    // Code to store
    // a list of
    // strings…
};
```

```
template <typename T>
class vector {
    // So satisfying.
};

vector<int> v1;
vector<double> v2;
vector<string> v3;
```

templates history
class-Preprocessor Macro-code generation

2025Fall-09-TemplateClasses, p.26

> Key Idea: Templates automate code generation

template instantiation

Templates have come a long way

```
template <typename T>
class Vector {
public:
    T& at(size_t index);
    void push_back(const T& elem);
private:
    T* elems;
};
```

Template Declaration

Vector is a template that takes in *the name of a type* T

T gets replaced when Vector is **instantiated**

把T实例化为其他类型

Template Instantiation

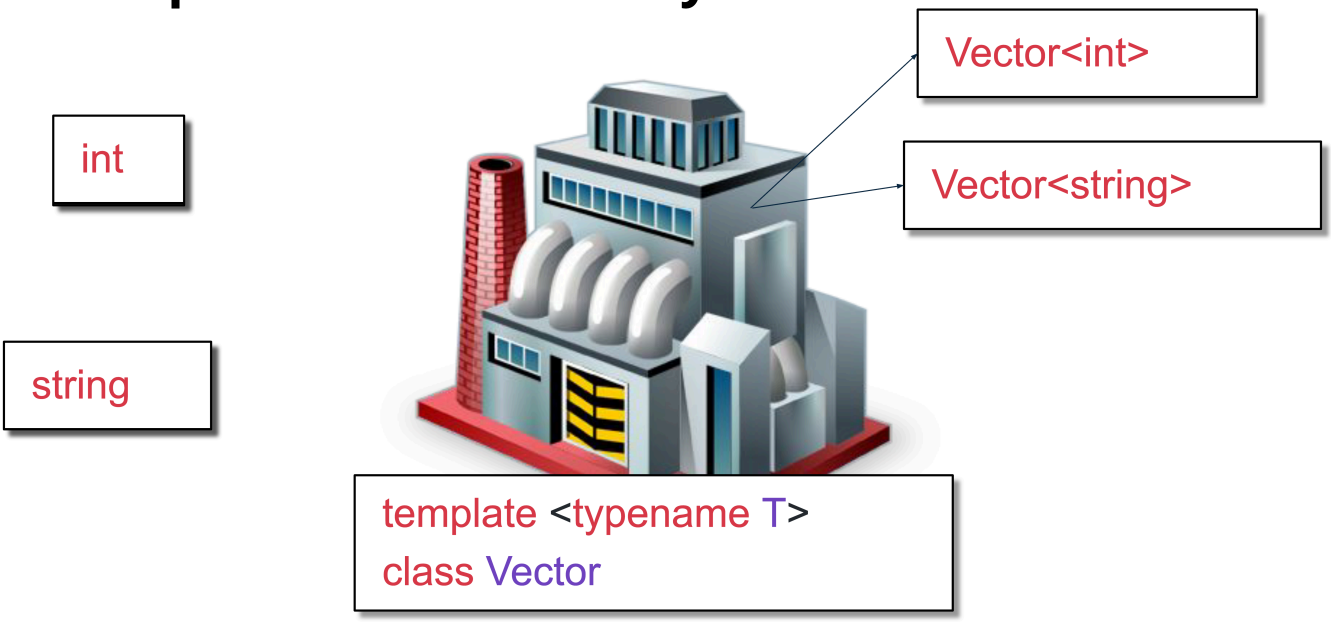
```
Vector<int> intVec;
Vector<double> doubleVec;
Vector<std::string> strVec;

Vector<Vector<int>> vecVec;

struct MyCustomType {};
Vector<MyCustomType> structVec;
```

Template Instantiation

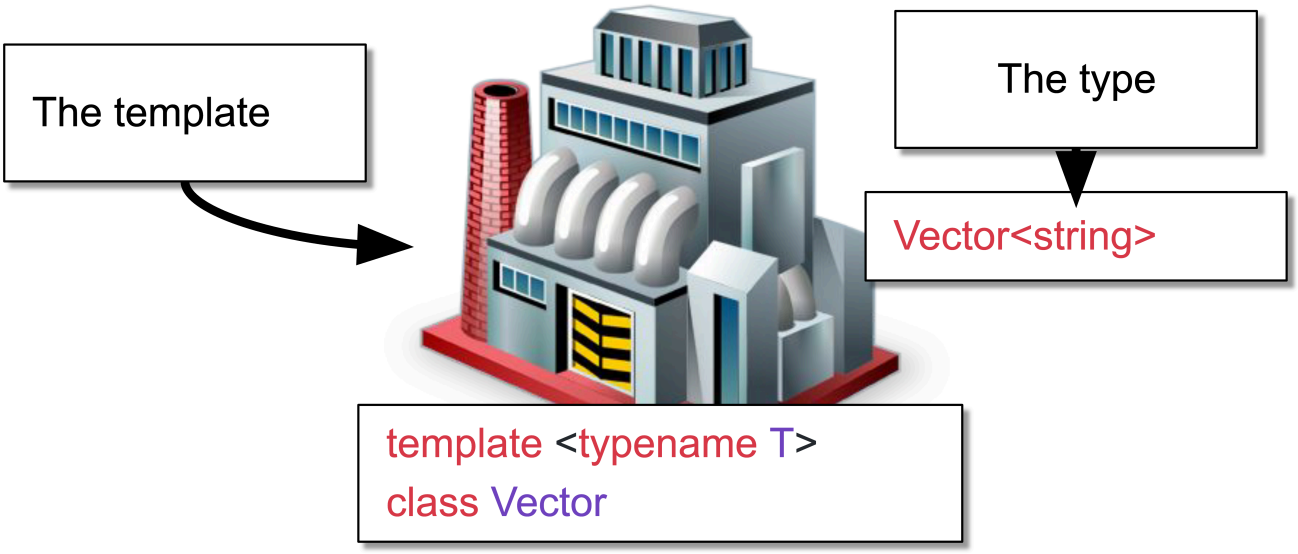
Code for a specific type is generated on-demand, when you use it

template的比喻

A template is like a factory



Templates vs. Types



2025Fall-09-TemplateClasses, p.40

> (1) Must copy template <…> syntax in .cpp

头文件与源文件的配置

2025Fall-09-TemplateClasses, p.44

> (2) .h must include .cpp at bottom of file

Template class implementation

For template classes, the .h file includes the .cpp file

```
// Vector.h

template <typename T>
class Vector {
public:
    T& at(size_t i);
};

#include "Vector.cpp"
```

```
// Vector.cpp

template <typename T>
T& Vector<T>::at(size_t i) {
    // Implementation...
}
```

注意，与类不同的地方在于，这个引入文件需要在头文件导入源文件，与类反了，看类

Normal class implementation

For non-template classes, the .cpp file includes the .h file

```
// StrVector.h

class StrVector {
public:
    string& at(size_t i);
};
```

```
// StrVector.cpp

#include "StrVector.h"

string& StrVector::at(size_t i)
{
    // Implementation...
}
```

2025Fall-09-TemplateClasses, p.48

> (3) typename is the same as class

可以把模板里的typename替换为class

(3) typename is the same as class

All of the following are identical:

```
template <typename K, typename V>
struct pair;
```

```
template <class K, class V>
struct pair;
```

```
template <class K, typename V>
struct pair;
```

```
template <class K, typename V>
struct pair;
```

const correctness

这个内容很关键，就是涉及到模板里仅可读内容

原本的vector

Back to our Vector class!

```
void printVec(const Vector<int>& v) {
    for (size_t i = 0; i < v.size(); i++) {
        std::cout << v.at(i) << " ";
    }
    std::cout << std::endl;
}
```

Compiler: "🦁 const Vector<int> has no size, at!!!"

目前这俩函数无法识别const类型

Back to our Vector class!

```
template<class T>
class Vector {
public:
    size_t size();
    bool empty();

    T& operator[] (size_t index);
    T& at(size_t index);
    void push_back(const T& elem);
};
```

Let's add **const** to the methods which don't modify **Vector**

现在参数都是可变的，如果传入参数是const 类型
那么找不到这样的函数，因为size()这些函数是可以修改内部的，const不允许，需要重写函数

Back to our Vector class!

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    T& at(size_t index) const;
    void push_back(const T& elem);
};
```

Let's add **const** to the methods which don't modify **Vector**

这样就找到const变量了

Back to our Vector class!

```
void printVec(const Vector<int>& v) {
    for (size_t i = 0; i < v.size(); i++) {
        std::cout << v.at(i) << " ";
    }
    std::cout << std::endl;
}
```

Compiler: "✅ Everything looks good to me!"

2025Fall-09-TemplateClasses, p.72

> Problem #1: const consumers can modify!

但是还没完，虽然这俩函数找到了const,但是const意味着这种函数类型也必须是const的，需要把T&改为const T&

Solution: return a const reference

```
template<class T>
class Vector {
public:
    size_t size() const;
    bool empty() const;

    T& operator[] (size_t index);
    const T& at(size_t index) const;
    void push_back(const T& elem);
};
```

Hmm... There's still a problem here

2025Fall-09-TemplateClasses, p.74

> Problem #2: non-const consumers can't modify!

这就是第二个问题。
但是如果在vector里直接加const，那么未来传入一个正常可以修改的参数，又找不到原先这样的函数了，我们可以写两个这样的

头文件

Solution: const overloading!

- Let's define two versions of our at method
- One version gets called for const instances
- ...And another that gets called for non-const instances

```
template<class T>
class Vector {
public:
    const T& at(size_t index) const;
    T& at(size_t index);
};
```

cpp文件

Solution: const overloading (.cpp file)!

```
template <class T>
const T& Vector<T>::at(size_t index) const {
    return elems[index];
}

template <class T>
T& Vector<T>::at(size_t index) {
    return elems[index];
}
```

类似的，我们将引入一个新的关键词来简化这些redundant
起因：设计了个新函数，发现redundant

Implementing findElement

```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}

template <typename T>
const T& Vector<T>::findElement(const T& value) cor
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}
```

This works, but it's super redundant. There must be a better way!

引入关键字：const_cast 打补丁

A slight (but useful) aside

- Casting: the process of converting one type to another
  - There are *many* ways to cast in C++
- const_cast allows us to "cast away" the const-ness of a variable
  - Usage: const_cast<target_type>(expression)
  - So why is this useful?

化简结果

Implementing findElement

```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[
    }
    throw std::out_of_range("Element not
}

template <typename T>
const T& Vector<T>::findElement(const T& value) const {
    return const_cast<Vector<T>&>(*this).findElement(value);
}
```

Ahh no more redundancy…
But what in the Bjarne is going on here?

这句话每个位置都有含义

const_cast casts away the const

this dereferences a const Vector<T>, giving us a const-ref

```
const_cast<Vector<T>&>(*this).findElement(value);
```

Vector<T>& is a **non-const** reference, the type we would like

Phew... This is the non-const version of findElement

当然未来在写这样的模板函数的时候，在声明阶段是不变的，但是在具体写执行和实现过程，可以用这种关键词来简化
如：

const_cast forces compiler to pick right overload

```
template<class T>
class Vector {
public:
    T& at(size_t index);
    const T& at(size_t index) const;
    T& findElement(const T& value);
    const T& findElement(const T& value) const;
};
```

Implementing findElement

```
template <typename T>
T& Vector<T>::findElement(const T& value) {
    for (size_t i = 0; i < logical_size; i++) {
        if (elems[i] == elem) return elems[i];
    }
    throw std::out_of_range("Element not found");
}

template <typename T>
const T& Vector<T>::findElement(const T& value) const {
    return const_cast<Vector<T>&>(*this).findElement(value);
}
```

但是这个const_cast最好不要用，除非你知道自己明确要使用它

When to use const_cast?

- Short answer: just about never
- const_cast tells the compiler: “don’t worry I’ve got this”
- If you need a mutable value, just don’t add const in the first place
- Valid uses of const_cast are few and far between

原因如下：
简单直接的回答是：**除非你百分之百确定自己在做什么，否则永远不要用它。**

这张图传达的是 C++ 开发者中一个非常普遍的共识：`const_cast` 通常是代码设计缺陷的“遮羞布”。

---

1. 为什么图里建议“几乎从不”使用？

- **破坏了编译器的承诺**：`const_cast` 的主要作用是去除变量的 `const`（常量）属性。当你给一个变量加上 `const` 时，你是在告诉编译器和你的同事“这个值不会改变”。使用 `const_cast` 就像是当面反悔。
- **未定义行为（危险！）**：如果你试图用 `const_cast` 去修改一个**本质上就是常量**的变量（例如 `const int x = 10;`），程序可能会直接崩溃或出现莫名其妙的错误，这属于 C++ 中的“未定义行为”。
- **更好的选择是“一开始就不加”**：如果你发现一个变量需要被修改，最正确的做法是直接去掉它的 `const` 声明，而不是在后面用强制类型转换来补救。

---

2. 什么时候“必须”或“可以”用它？

虽然图中说“有效用途寥寥无几”，但在现实开发中，有两种情况比较常见：

A. 对接老旧的第三方库（Legacy API）

有些老旧的 C 语言库，它们的函数参数虽然不修改数据，但由于历史原因没写 `const` 。

C++

```
void old_c_api(char* str); // 其实它不改字符串，但没写const

const char* my_text = "hello";
old_c_api(const_cast<char*>(my_text)); // 这种情况下，你不得不妥协
```

B. 常量/非常量重载的去重（Avoid Code Duplication）

当一个成员函数有 `const` 和非 `const` 两个版本，且逻辑完全一样时，为了避免写两遍同样的代码，可以用 `const_cast` 让其中一个调用另一个。

---

3. 总结建议

情况	建议
如果你觉得需要修改自己的 `const` 变量	**不要用**。去检查你的逻辑，把变量声明处的 `const` 删掉。
如果你为了走捷径绕过编译器报错	**不要用**。这通常会埋下难以排查的 Bug。
如果你在处理没法修改的第三方库接口	**可以用**。这是它存在的少数合法理由之一。

一句话总结： `const_cast` 是用来“修补”别人写错的接口的，而不是用来“逃避”你自己设定的规则的。

新的关键字：mutable

2025Fall-09-TemplateClasses, p.95

> A C++ party trick: mutable keyword

A C++ party trick: mutable keyword

Like const_cast, mutable circumvents const protections. Use it carefully!

```
struct MutableStruct {
    int dontTouchThis;
    mutable double iCanChange;
};

const MutableStruct cm;
// cm.dontTouchThis = 42;  // ❌ Not allowed, cm is const
cm.iCanChange = 3.14;           // ✅ Ok, iCanChange is mutable
```

我个人感觉mutable和前面的很像，这个代码简单来说就是定义这个struct全是const,正常你就不能修改，但是如果前面又mutable修饰，那么你可以修改，类似于补丁吧

说白了在这个mutable是更好的设计

你的感觉非常准，`mutable` 确实可以被看作是一个“合法的补丁”，但它和 `const_cast` 在使用意图上有本质的区别。

简单来说：**`const_cast` 是在“强扭瓜”，而 `mutable` 是在“留后门”。**

以下是针对你提供的代码和概念的详细拆解：

---

1. `mutable` 的核心逻辑：逻辑常量 vs 物理常量

在 C++ 中，`const` 函数通常意味着“这个操作不会改变对象的**状态**”。但有时候，为了实现这个“不改变状态”的操作，你不得不修改一些后台数据（比如缓存、日志计数、互斥锁）。

- **`const_cast` 的逻辑：** “我知道这是个常量，但我不管，我就要强行改它。”
- **`mutable` 的逻辑：** “虽然整个对象是常量，但**这个特定成员**无论何时都是可以修改的，即使在 `const` 函数里也不例外。”

---

2. 代码解析：谁能动，谁不能动?

根据你上传的图片示例：

C++

```
struct MutableStruct {
    int dontTouchThis;           // 普通成员
    mutable double iCanChange;   // 带有 mutable 补丁的成员
};

const MutableStruct cm;          // 定义一个常量对象
```

- **`cm.dontTouchThis = 42;` ❌ 报错：** 因为 `cm` 被声明为 `const`，它的普通成员变量也被锁死了。
- **`cm.iCanChange = 3.14;` ✅ 允许：** 即使 `cm` 是 `const`，`mutable` 关键字就像给了这个变量一个“免死金牌”，允许它被修改。

---

3. 为什么说它比 `const_cast` 更专业？

虽然两者都规避了 `const` 保护，但 `mutable` 被认为是更优雅的设计，原因如下：

特性	const_cast	mutable
透明度	**隐藏的**。阅读类定义时看不出会被改，但在实现文件里偷偷转类型。	**显性的**。在类定义时就告诉所有人：这个变量是允许变动的。
安全性	**高风险**。可能导致未定义行为（如果你改了真正的只读内存）。	**安全的**。编译器会预留出修改的空间。
典型用途	适配老旧库、不规范的接口。	**实现逻辑：** 比如 `const` 接口内部的 `std::mutex` 加锁，或者计算结果的缓存。

---

4. 总结：什么时候该用这个“补丁”？

你应该在满足以下描述时使用 `mutable`：

1. **逻辑上它是常量：** 用户调用你的 `getValue()` 函数，觉得对象没变。
2. **物理上它需要变：** 为了让 `getValue()` 更快，你需要在内部更新一个 `cache` 变量，或者更新一个“访问次数”计数器。

如果你发现自己想用 `const_cast` 来修改类成员，请停下来，考虑把那个成员改成 `mutable`。
这会让你的代码意图更清晰，也更符合 C++ 的设计哲学。