深蓝学院
shenlanxueyuan.com

C++ 基础

第 11 章：类

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

结构体与对象聚合

```
//定义,给出内存的组织结构
struct Str {
  int x;
  int y;
};
```

```
//声明,告诉编译器如果以后看到Str,那么这是个结构体
struct Str;
```

```
typedef struct Str {
  int x;
  int y;
} MStr;
// 表示有一个结构体,我们为它起了个别名,叫MStr
```

- 结构体：对基本数据结构进行扩展，将多个对象放置在一起视为一个整体
  - 结构体的声明与定义（注意定义后面要跟分号来表示结束）
  - 仅有声明的结构体是不完全类型（incomplete type）
  - 结构体（以及类）的一处定义原则：翻译单元级别

对于不完全的类型，我们不能声明一个对象（因为我们不知道这个对象需要占用多大的内存），我们还是可以声明其对应的指针。因为指针的大小是一定的。

翻译单元级别意味着，我可以在source.cpp中定义一个Str,然后在main.cpp中定义一个Str.

```
struct Str {
//结构体声明
  int x;//数据成员定义
  int y;
};
```

- 数据成员（数据域）的声明与初始化

声明：告诉编译器有这个东西，定义：运行到这的时候需要分配相应的内存。结构体定义内部包含的是数据成员的声明。

  - （C++11）数据成员可以使用 decltype 来声明其类型，但不能使用 auto
  - 数据成员声明时可以引入 const 、引用等限定
  - 数据成员会在构造类对象时定义
  - （C++11）类内成员初始化
  - 聚合初始化：从初始化列表到指派初始化器

```
struct Str {
  decltype(3) x;
  int y;
  const int z = 3;
  int a = 3;
// 此时不会分配内存
};
int main() {
  Str m_str;
  // 此时才会分配内存
}
```

```
//指派初始化
struct Str {
  int x;
  int y;
  int z;
};
int main() {
  Str m_str{.x=3, .y=4};
  std::cout << m_str.y;
}
```

- mutable 限定符

修改

```
struct Str {
  mutable int x = 0;
  int y = 1; };
int main() {
  const Str m_str;
  m_str.x = 3;
  //加入mutable，即使str是const,也可以改变x的值
```

结构体与对象聚合（续 1）

```
//类外定义
struct Str {
  static int x;
  int y;};
int Str::x;
//类外定义，这句话
必须写，不写会报错
```

```
//类内初始化
struct Str {
  const static int array_Size = 100;
  int buffer[array_Size];
};
//有限制。你只能使用array_Size的值，
不能调用array_Size的地址
array_Size是一个编译期的行为，在编译期
array_Size都会被替换为100.我并没有构造
一块内存和array_Size关联起来。
改进：加入定义,有定义才能进行相应的内存分配
const int Str::array_Size;
```

- 静态数据成员—多个对象之间共享的数据成员
  - 定义方式的衍化
    - C++98： 类外定义， const 静态成员的类内初始化
    - C++17： 内联静态成员的初始化

```
//内联静态成员初始化
struct Str {
  inline const static int array_Size=100;
  int buffer[array_Size];
};
加入Inline，那么不需要这个静态成员是const
类型。也可行
```

内联函数，虽然出现多次定义，但是编译器在最后链接的时候，只选择其中的一个。

  - 可以使用 auto 推导类型
- 静态数据成员的访问
  - “.” 与“ ->” 操作符

```
Str* ptr = &m_str1;
ptr->array_Size;
Str::array_Size;
```

  - “::” 操作符
- 在类的内部声明相同类型的静态数据成员

```
struce Str {
  Str x; // error
  static Str x;
//可以通过编译。它是静态数据成
员，那么它一定
不属于任何一个对象。所以编译器
在计算str所占的大小的时候，不会
考虑static str x，所以可以通过
};
Str Str::x; //引入定义。
```

```
如果使用内联静态初始化，就会报错
struct Str {
  inline static Str x;
};
//原因，编译器是按行解析，当我们
解析到这一行的时候，编译器就要产生
一个x的定义，但是此时编译器不能对
Str有一个全面的了解，所以会报错。
```

成员函数（方法）

- 可以在结构体中定义函数，作为其成员的一部分：对内操作数据成员，对外提供调用接口
  - 在结构体中将数据与相关的成员函数组合在一起将形成类，是 C++ 在 C 基础上引入的概念
  - 关键字 class
  - 类可视为一种抽象数据类型，通过相应的接口（成员函数）进行交互
  - 类本身形成域，称为类域
- 成员函数的声明与定义
  - 类内定义（隐式内联）
  - 类内声明 + 类外定义 在类外定义就不是内联的了
  - 类与编译期的两遍处理
  - 成员函数与尾随返回类型（ trail returning type ）

```
Str::MyRes Str::fun()
{
    return x;
}
```

```
auto Str::fun() -> MyRes
{
    return x;
}
```

//trail returning type
如果觉得左边太啰嗦，可以使用右边的方法。当编译器看到Str::fun()的时候，编译器就知道我们要处理Str这个类，就会尝试在Str这个域里找

```
struct Str
{
    MyRes fun()
    {
        return x;
    }
    using MyRes = int;
    int x;
};
```

虽然有两遍处理，这个还是会报错，因为不知道MyRes是什么

```
struct Str
{
    void fun()
    {
        std::cout << x << std::endl;
    }
    int x;
};
```

编译器在处理代码的时候，一般是从上到下依次处理的。可是在这里，在处理fun的时候，就会遇到x，可是它不知道x的含义是什么。因此这里就涉及到类在编译期会进行两遍处理。
对于一个类来说，是函数比较重要，是接口，对于别人来说，可能并不关注你的类的数据有什么。通常来说，大家会把比较重要的东西放到前面。
=》需要两遍处理
会先通看一遍（除了函数内部的内容），然后再去处理函数内部的内容

成员函数（方法）——续 1

- 成员函数与 this 指针
  - 使用 this 指针引用当前对象
  - 基于 const 的成员函数重载
- 成员函数的名称查找与隐藏关系
  - 函数内部（包括形参名称）隐藏函数外部
  - 类内部名称隐藏类外部
  - 使用 this 或域操作符引入依赖型名称查找
- 静态成员函数
  - 在静态成员函数中返回静态数据成员
- 成员函数基于引用限定符的重载（C++11）

ref-qualified member functions

https://kukuruku.co/post/ref-qualified-member-functions/

```
struct Str {
  void fun() // 隐藏参数Str* this
  {
    std::cout << this << std::endl; }
  int x;
};
```

```
//如果我们不希望改变类的数据成员
struct Str {
  void fun(int x) const { //在后面写一个const
  //如果不写这个const，this的类型是Str* const
  // 如果加一个const，this的类型是 const Str* const
    ... }
  int x;
};
```

如果是静态函数static，
那么就不会传入隐藏参数

```
struct Str {
  static void fun() {
    // ,,, }
  int x;
};
```

调用方式：

```
Str::fun();
str m_str1;
m_str1.fun();
```

用处：描述和类很相关的东西，又不需要描述对象，

```
class some_type {
  void foo() &; // 左值引用
  void foo() &&;//右值引用
  void foo() const &;//常量左值引用
  void foo() const &&;//常量右值引用,
我们一般不会用常量右值，因为一般右值，
我们都为了修改。
  //左值和右值指的是调用对象的性质
};
int main() {
  some_type t; // t是一个左值
  t.foo(); // foo()&
  some_type().foo(); // some_type()
相当于构造了一个纯右值,foo()&&
```

```
struct Str {
  void fun(int x) {
  // 相当于隐式插入了一个变量Str*
const this
  }
  void fun(int x) {
  // 相当于隐式插入了一个变量const
Str* const this
  }
  int x;
};
```

同样名的函数，有的函数可能不改变数据成员，有的函数可能会改变

```
struct Str {
  static auto& instance() {
    static Str x;
    return x; }
};
//可以使用如上的方式在静态成员
函数中返回静态数据成员。可以通
过这种方法来实现单例模式或者其
它功能
```

注意：基于const的成员函数重载和基于引用限定符的重载不要混用。

访问限定符与友元

- 使用 public/private/protected 限定类成员的访问权限
  - 访问权限的引入使得可以对抽象数据类型进行封装
  - 类与结构体缺省访问权限的区别
- 使用友元打破访问权限限制—关键字 friend
  - 声明某个类或某个函数是当前类的友元—慎用！因为封装是很有用的，所以要慎用。friend打破封装是非常彻底的
  - 在类内首次声明友元类或友元函数
    - 注意使用限定名称引入友元并非友元类（友元函数）的声明
  - 友元函数的类内外定义与类内定义
  - 隐藏友元（hidden friend）：常规名称查找无法找到（参考文献）
    - 好处：减轻编译器负担，防止误用
    - 改变隐藏友元的缺省行为：在类外声明或定义函数

```
int main();
class Str2;
class Str {
  friend int main();
//授权main去访问Str内的所有东西
  friend Str2;
  inline static int x;
}
class Str2 {
  void fun() {
    std::cout << Str::x << std::endl;}
};
int main() {
  // ...
}
```

```
//在类内首次声明友元类或友元函数
class Str {
  inline static int x;
  int y;
  friend void fun();//在类内首次声明友元类或友元函数
  friend class Str2;
// friend void ::fun(); // 限定名称，不会被编译器认为为声明
//不能使用限定名称引入（首次声明）
};
```

```
void fun();//加声明可以改变隐藏友元的缺省行为
class Str {
  friend void fun() {
    //这个fun函数的作用域是全局。
    //但是这个fun函数是个隐藏友元
//但是我们在main()函数中找不到这个fun函数，会报错undeclare type(无声明)
}
//adl argument depend lookup
```

友元函数的类内定义

```
class Str { // 解决常规名称无法查找到隐藏友元，这个才是正确打开方式
  friend void fun(const Str& val) { //.. };
};
int main() {
  Str val;
  fun(val); // adl argument depend lookup实参类型依赖查找
```

构造、析构与复制成员函数

- 构造函数：构造对象时调用的函数
  - 名称与类名相同，无返回值，可以包含多个版本（重载）
  - （C++11）代理构造函数
- 初始化列表：区分数据成员的初始化与赋值
  - 通常情况下可以提升系统性能
  - 一些情况下必须使用初始化列表（如类中包含引用成员）
  - 注意元素的初始化顺序与其声明顺序相关，与初始化列表中的顺序无关
  - 使用初始化列表覆盖类内成员初始化的行为
- 缺省构造函数：不需要提供实际参数就可以调用的构造函数
  - 如果类中没有提供任何构造函数，那么在条件允许的情况下，编译器会合成一个缺省构造函数
  - 合成的缺省构造函数会使用缺省初始化来初始化其数据成员
  - 调用缺省构造函数时避免 most vexing parse
  - 使用 default 关键字定义缺省构造函数

```
class Str {
public:
  Str() {
    x = 3;}
  Str(int input) {
    x = input;}
//为了避免对于x赋值的重复提出了改进方法
private:
  x = input;
}
```

```
Str(cosnt std::string& val) {
  //此时x已经被构造出来了，使用缺省构造的方式
  x = val;
}//这种方式性能不是很好。首先要使用缺省构造的方式构造，然后对x进行赋值。
//好的方式，直接在初始化的时候赋值
Str(const std::string val) : x(val),y(0) {}
```

```
改进方法：
方法一：使用缺省方法
Str(int input = 3) {
  x = input;}
//缺点，对于多个参数不太实用。因为缺省实参只能放在右边
//方法二：代理构造函数
Str() : Str(3) {} //Str(3)就叫代理函数
//代理构造函数包含了两个部分，一个是Str(3),一个是括号内的函数体。那么就涉及到这两个部分哪个先执行，哪个后执行。系统会限制性Str(3)内的内容，然后再执行括号内的
Str(int input) { x = input;}
```

```
Str() : x(50) {}
private:
  size_t x = 3;//初始化列表优先级高于类内成员初始化
```

```
Str(const std::string& val):x(val), y(x.size()) {};
private:
  std::string x;
  size_t y; //合理的，x会被先初始化x，再初始化y
```

这个功能也引出了一个书写的建议：在初始化赋值的时候，最好也根据声明顺序赋值

如果有引用存在，那么就没有办法合成缺省的构造函数

```
Str m(3); //编译器会把这个语句当作一个函数的声明，这种情况就叫most vexing parse.//改进方法：换成大括号
Str m{3};
```

```
Str() {}
Str(const std::string& input)
: x(input) {}
std::string x;
```

缺省构造函数内部的逻辑，和编译器所合成的缺省构造函数的逻辑是一样的

```
左边改进写法：//11之后引入的行为
Str() = default;
Str(const std::string& input)
:x(input) {}
```

构造、析构与复制成员函数（续 1）

```
//单一参数构造函数，可以视为一种类型转换函数，因此下面的写法也是合法的
Str m = static_cast<Str>(3);

 //copy constructor
 Str(const Str& x) : val(x.val) {};
```

```
//可以使用explicit避免求值过程中隐式转换
struct Str {
  explicit Str(int x) : val(x) {}//你必须引入一个显示的，而不是一个隐式的转换
  int val;
};
int main() { Str m=3;//此时这个函数就会报错，因为它不允许你使用拷贝初始化的方式，把3赋给m
```

- 单一参数构造函数
  - 可以视为一种类型转换函数
  - 可以使用 explicit 关键字避免求值过程中的隐式转换
- 拷贝构造函数：接收一个当前类对象的构造函数
  - 会在涉及到拷贝初始化的场景被调用，比如：参数传递。因此要注意拷贝构造函数的形参类型
  - 如果未显式提供，那么编译器会自动合成一个，合成的版本会依次对每个数据成员调用拷贝构造

参数传递一般是const& 类型的

如果只是想依次调用拷贝构造，那么可以写default来构造一个 Str(const Str&) = default;

- 移动构造函数 (C++11)：接收一个当前类右值引用对象的构造函数
  - 可以从输入对象中“偷窃”资源，只要确保传入对象处于合法状态即可
  - 当某些特殊成员函数（如拷贝构造）未定义时，编译器可以合成一个
  - 通常声明为不可抛出异常的函数
  - 注意右值引用对象用做表达式时是左值！

要确保偷窃之后，传入对象还处于合法状态：还可以调用m.fun(),还可以调用析构等函数

```
Str(Str&& x) // 移动构造函数
:val(x.val),a(std::move(x.a)) {}
int val = 3;
std::string a = "abc";

int main() {
  Str m;
  m.fun();
  Str m2 = std::move(m);
  m.fun();
}
```

如果Str中移动构造函数是有效的，并且Str中包含了某个数据成员Str2，并且Str2中只有复制构造函数（没有移动），这个时候Str2的default移动构造函数会调用Str2的复制构造函数来完成移动构造函数

```
//右值引用对象做表达式时左值
Str(Str&& x) noexcept {
  std::string tmp = x;
//虽然输入的时候时右值，但是在函数内部做表达式使用的时候，它就变成左值了。如果想要当作右值使用，就需要写std::move(x)
```

Str(Str&&) noexcept{} 推荐使用这种写法。因为异常一般需要引入一些额外的资源，会造成一点资源的浪费 还有一点，很多类型在使用移动构造函数的时候，都会在保证该函数不会抛出异常的时候才会使用它。例如：vector.在使用push_back的时候，只有在移动构造函数不会产生异常的时候，才会选择移动构造函数

构造、析构与复制成员函数（续 2）

- 拷贝赋值与移动赋值函数（operator = ）
  - 注意赋值函数不能使用初始化列表
  - 通常来说返回当前类型的引用
  - 注意处理给自身赋值的情况
  - 在一些情况下编译器会自动合成
- 析构函数
  - 函数名：“～”加当前类型，无参数，无返回值
  - 用于释放资源
  - 注意内存回收是在调用完析构函数时才进行
  - 除非显式声明，否则编译器会自动合成一个，其内部逻辑为平凡的
  - 析构函数通常不能抛出异常 `~Str() noexcept {};`

```
Str& operator= (const Str& x)
{ //拷贝赋值函数
  return *this;
}
Str& operator= (Str&& x)
{ //移动赋值函数
  val = std::move(x.val);
  a = std::move(x.a);
  return *this;}

Str m, m2;
m2 = m; //调用拷贝赋值函数，在c++ insights中被翻译成
m2.operator=(m);
m2 = std::move(m);//移动赋值
```

```
//注意处理给自身赋值的情况
Str& operator= (Str&& x) {
  delete ptr;
  ptr = x.ptr;
  x.ptr = nullptr;
  val = std::move(x.val);
  a = std::move(x.a);
  return *this;
}
//一般情况下这个函数是没问题，但是再m移动赋值给m的时候，就会有问题例如：
m=std::move(m); 改进：在程序前加这句话
if (&x == this) { return *this;}
```

构造、析构与复制成员函数（续 3 ）

- 通常来说，一个类：
  - 如果需要定义析构函数，那么也需要定义拷贝构造与拷贝赋值函数
  - 如果需要定义拷贝构造函数，那么也需要定义拷贝赋值函数
  - 如果需要定义拷贝构造（赋值）函数，那么也要考虑定义移动构造（赋值）函数
- 示例：包含指针的类
- default 关键字
  - 只对特殊成员函数有效

```
class Str {
public:
  Str() : ptr(new int()) {
  // 构造函数}
  ~Str() {//析构函数
    delete ptr;}
  Str(const Str& val)
  //拷贝构造
    : ptr(new int()) {
      *ptr = *(val.ptr); }
  Str& operator= (const Str& val) {
    *ptr = *(val.ptr); }
  Str& operator=(const Str&val) {
    *ptr = *(val.ptr);
    return *this; }
  Str(Str&& val) noexcept : ptr(val.ptr){
    val.ptr = nullptr;}//移动构造

  int& Data(){
    return *ptr;}
private:
  int* ptr;
};
```

```
int main() {
  Str a;
  a.Data() = 3;
  Str b(a);
}
```

构造、析构与复制成员函数（续 3）

- delete 关键字
  - 对所有函数都有效
  - 注意其与未声明的区别
  - 注意不要为移动构造（移动赋值）函数引入 delete 限定符
    - 如果只需要拷贝行为，那么引入拷贝构造即可
    - 如果不需要拷贝行为，那么将拷贝构造声明为 delete 函数即可
    - 注意 delete 移动构造（移动赋值）对 C++17 的新影响

```
void fun(int) = delete;//这个函数被删除了，不能被调用了
void fun(doule a) { ...}
int main() {
    fun(3);//此时会报错。因为int已经被删除了。而且int是
完美匹配，所以会优先选择这个。
}
和未声明相比报错的结果不同。
```

构造、析构与复制成员函数（续 4 ）

https://www.youtube.com/watch?v=vLinb2fgkHk&ab_channel=InsideBloomberg
老师推荐看

- 特殊成员的合成行为列表（红框表示支持但可能会废除的行为）

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

一行代表user declares用户声明. Nothing：类的声明者并不会声明任何东西，那么compiler会给你合成所有的，每一种都是default

如何查最新的表呢？在cppreference中查询useful resources,列出的免费的c++标准打开需要的pdf，我们可以找到Implicit declaration of copy function

字面值类，成员指针与 bind 交互

- 字面值类：可以构造编译期常量的类型 我们将可以构造编译期常量的类型叫做字面值类
  - 其数据成员需要是字面值类型
  - 提供 constexpr / consteval 构造函数 （小心使用 consteval ） 如果使用了consteval，那么编译器就只会返回编译期常量
  - 平凡的析构函数
  - 提供 constexpr / consteval 成员函数 （小心使用 consteval ）
  - 注意：从 C++14 起 constexpr / consteval 成员函数非 const 成员函数

```
//编译器常量
constexpr int a = 3;
//提供constexpt/consteval构造函数
class Str{
public:
  constexpr Str(int val) : x(val) {}
private:
  int x;
};
constexpr Str a;
```

```
class Str {
public:
  constexpr Str(int val)
    : x(val) {}
  constexpr void inc() {
    x = x+1;}
  constexpr void read() const {
    return x; }
private:
   int x; };
constexpr int MyFun() {
  Str x(10);
  x.inc();
  x.inc();
  x.inc();
  return x.read();}
int main() { return MyFun();}
```

字面值类，成员指针与 bind 交互

- 成员指针 应用相对来说并不是很多，和字面值类一样，应用范围比较窄
  - 数据成员指针类型示例： int A::*;
  - 成员函数指针类型示例： int (A::*)(double);
  - 成员指针对象赋值： auto ptr = &A::x;
    - 注意域操作符子表达式不能加小括号（否则 A::x 一定要有意义）
  - 成员指针的使用：
    - 对象 .* 成员指针
    - 对象指针 ->* 成员指针
- bind 交互
  - 使用 bind + 成员指针构造可调用对象
  - 注意这种方法也可以基于数据成员指针构造可调用对象

int Str::*ptr;//这是一个数据成员指针，ptr可以指向str域里面的一个成员。这个成员的类型是Int
void (Str::* ptr_fun)();//成员函数指针类型,ptr_fun是个指针，这个指针可以指向Str域内的一个东西，这个东西是一个接受0个输入，返回值是void的函数

```
class Str {
  int x;
  int y;
};
class Str2 {
};
int main() {
  int Str::*ptr;
  void (Str::* ptr_fun)();
  int Str2::* ptr2;
  //ptr2和ptr是两个不同的指针

  int Str::*ptr = &Str::x;
  int Str::*ptr2 = &Str::y;

  *ptr;//不能对ptr进行解引用，因为class Str中int x是x的声明，而不是x的定义，如果想解引用的话，就需要定义一个对象
  Str obj;
  obj.x = 3;
  obj.*ptr;//理解方式：1。是数据成员操作符，我操作的是*ptr所指向的成员，2.*是一个整体，在cppreference（operator_precedence）中。*是一个特殊的操作符,叫Pointer-to-member,
  Str* ptr_obj = &obj;
  ptr_obj->*ptr;
}
```

深蓝学院
shenlanxueyuan.com

```
//使用bind
class Str{
  int x;
  int y;
  void fun(doule x) {
  }
};
int main() {
  auto ptr = &Str::fun;
  Str obj;
  (obj.*ptr)(100);

  //bind基本用法
  auto x = std::bind(ptr, obj, 100.0);
  //要注意别忘记了写obj,这个对象
  x();

  auto ptr2 = &Str::x;
  obj.*ptr2 = 3;

  //使用bind
  auto x = std::bind(ptr2,ob);
  x();//如果打印会打印出3
}
```

感谢聆听！

Thanks for Listening