

其他语法

@M了个J

<https://weibo.com/exceptions>

<https://github.com/CoderMJLee>



实力IT教育 www.520it.com

运算符重载 (operator overload)

- 运算符重载（操作符重载）：可以为运算符增加一些新的功能

```
class Point {  
public:  
    int m_x;  
    int m_y;  
    Point(int x, int y) :m_x(x), m_y(y) { }  
    Point(const Point &point) :m_x(point.m_x), m_y(point.m_y) { }  
    void display() {  
        cout << "x = " << this->m_x << ", y = " << this->m_y << endl;  
    }  
};  
  
Point operator+(const Point &p1, const Point &p2) {  
    return Point(p1.m_x + p2.m_x, p1.m_y + p2.m_y);  
}
```

```
Point p1(10, 20);  
Point p2(20, 30);
```

```
Point p3 = p1 + p2;  
p3.display();
```

```
x = 30, y = 50
```

- 全局函数、成员函数都支持运算符重载
- 课堂练习：Point、String类

临时String

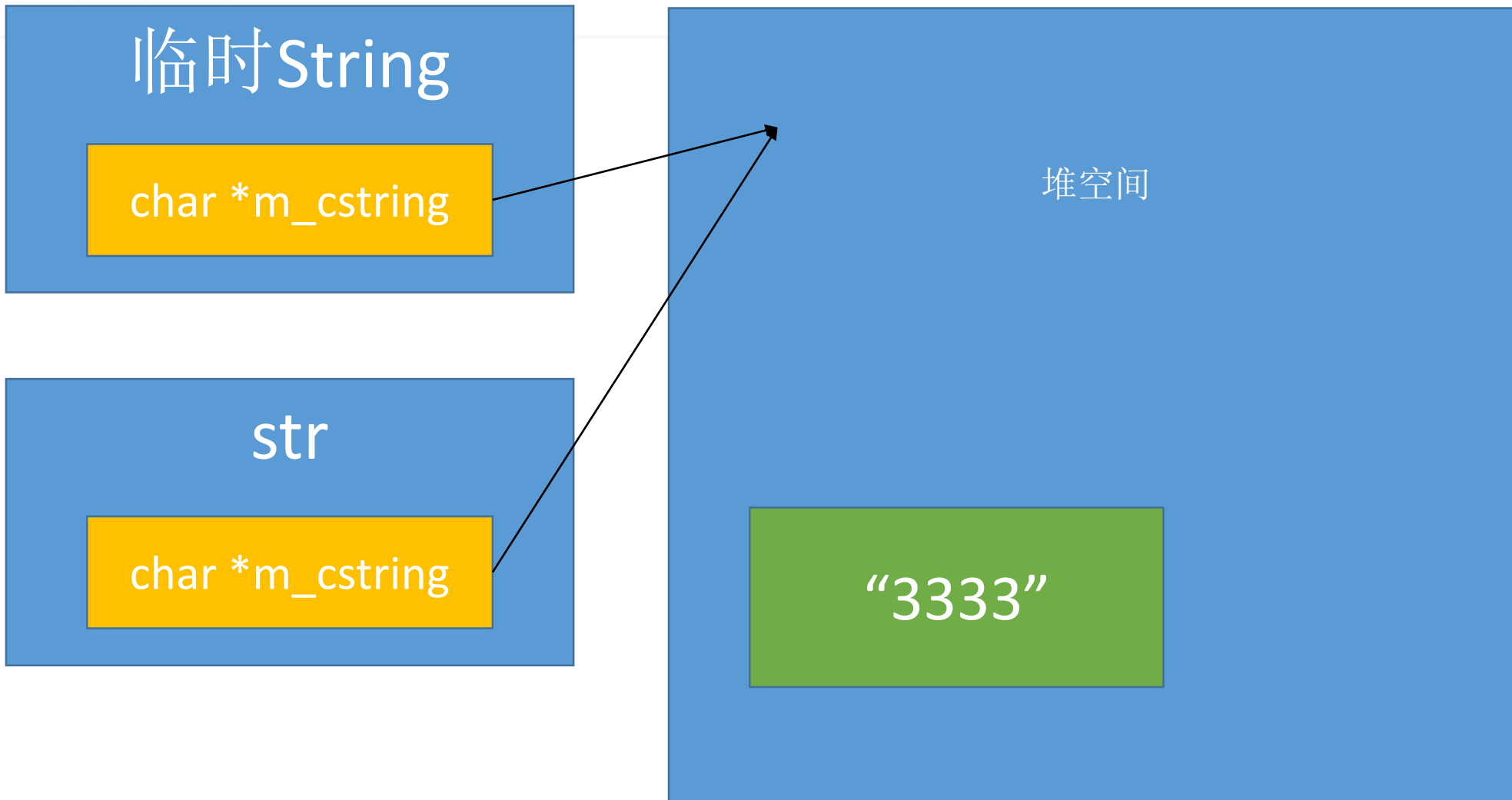
`char *m_cstring`

str

`char *m_cstring`

堆空间

“3333”



“444”

str

char *m_cstring

“333444”

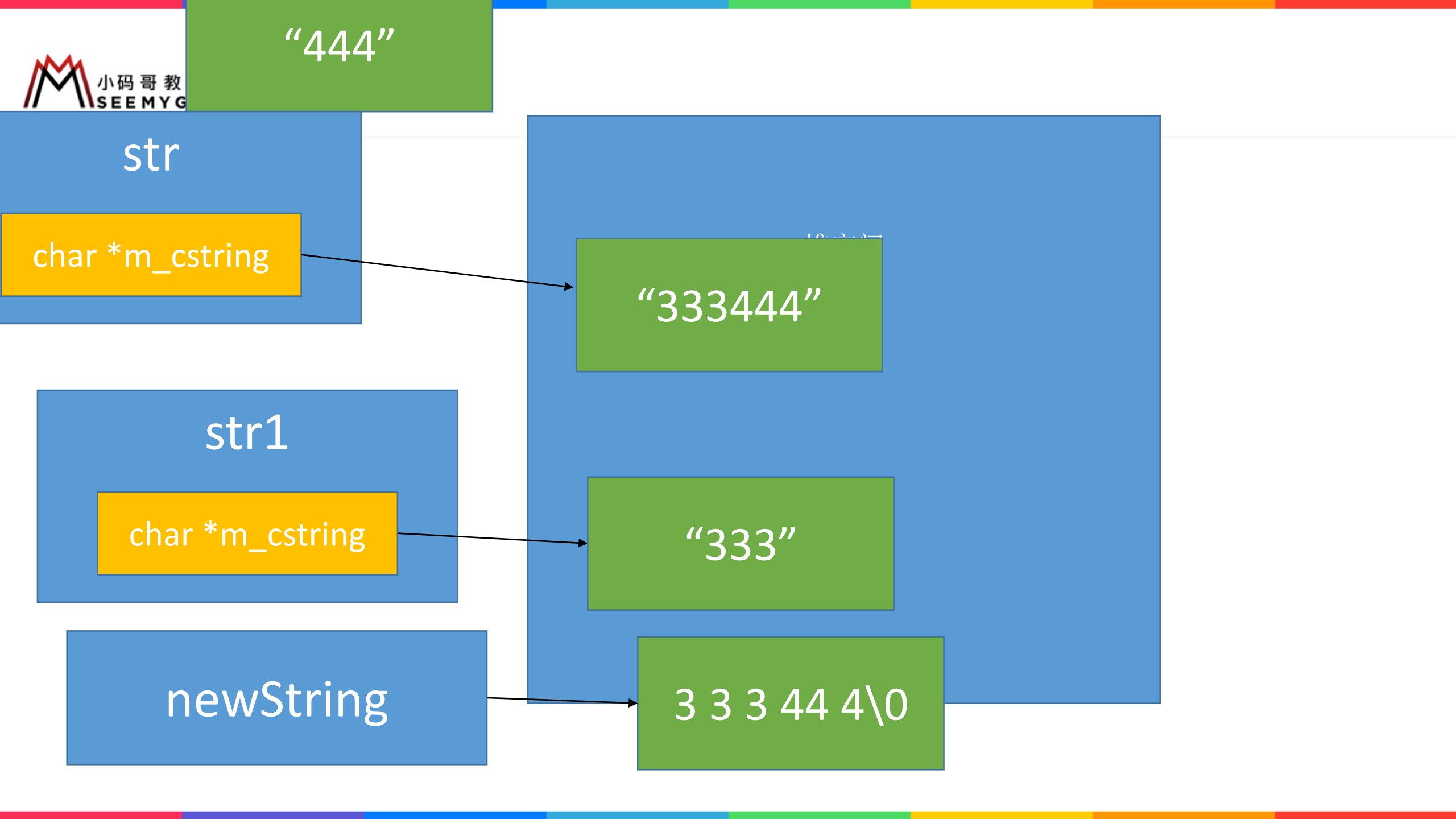
str1

char *m_cstring

“333”

newString

3 3 3 44 4\0



调用父类的运算符重载函数

```
class Person {  
    int m_age;  
public:  
    Person &operator=(const Person &person) {  
        this->m_age = person.m_age;  
    }  
};  
  
class Student : public Person {  
    int m_score;  
public:  
    Student &operator=(const Student &student) {  
        Person::operator=(student);  
  
        this->m_score = student.m_score;  
    }  
};
```

单例模式补充

```
class Rocket {  
public:  
    static Rocket * sharedRocket() {  
        if (ms_rocket) return ms_rocket;  
        return ms_rocket = new Rocket();  
    }  
    static void deleteRocket() {  
        if (!ms_rocket) return;  
        delete ms_rocket;  
        ms_rocket = NULL;  
    }  
private:  
    static Rocket *ms_rocket;  
    Rocket() { }  
    Rocket(const Rocket &rocket) { }  
    Rocket &operator=(const Rocket &rocket) { }  
};  
  
Rocket *Rocket::ms_rocket = NULL;
```

仿函数（函数对象）

- 仿函数：将一个对象当作一个函数一样来使用

```
class Sum {  
public:  
    int operator()(int a, int b) {  
        return a + b;  
    }  
};
```

```
Sum sum;  
cout << sum(20, 30) << endl;
```

- 对比普通函数，它作为对象可以保存状态

运算符重载注意点

■ 有些运算符不可以被重载，比如

□ 对象成员访问运算符：.

□ 域运算符：::

□ 三目运算符：?:

□ sizeof

■ 有些运算符只能重载为成员函数，比如

□ 赋值运算符：=

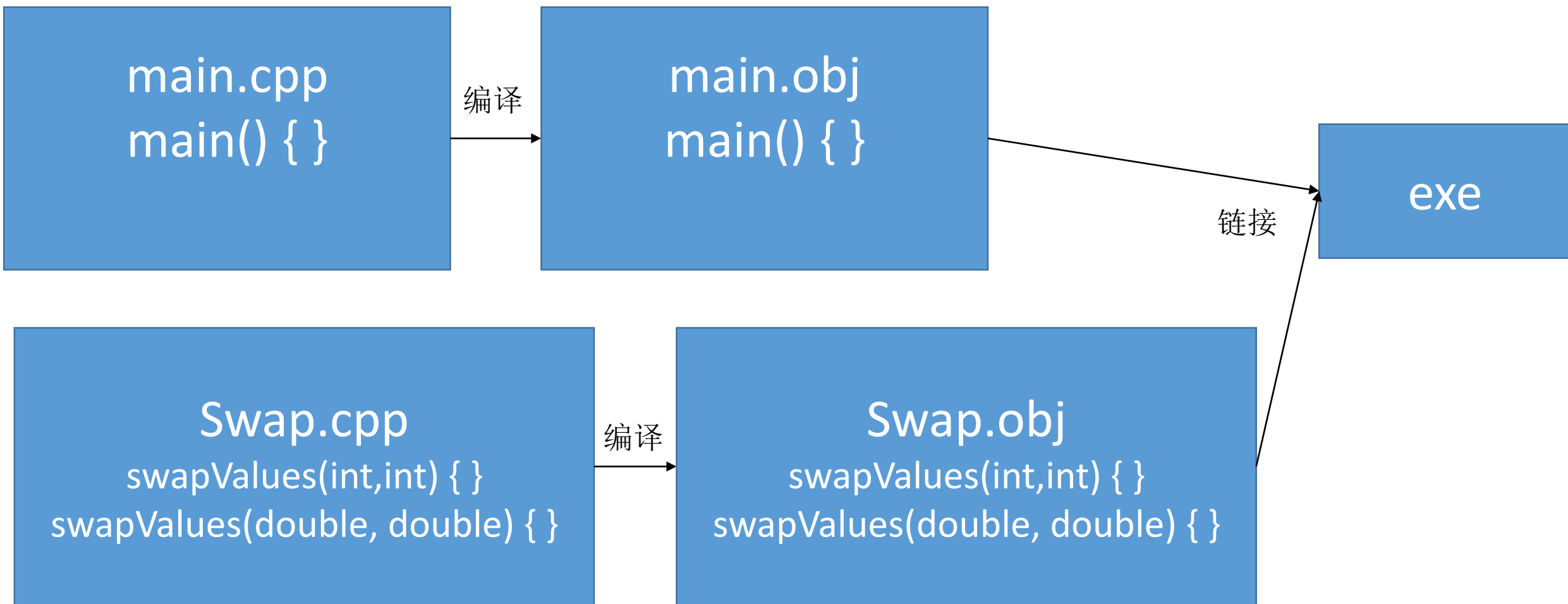
□ 下标运算符：[]

□ 函数运算符：()

□ 指针访问成员：->

模板 (template)

- 泛型，是一种将类型参数化以达到代码复用的技术，C++中使用模板来实现泛型
- 模板的使用格式如下
 - `template <typename\class T>`
 - `typename`和`class`是等价的
- 模板没有被使用时，是不会被实例化出来的
- 模板的声明和实现如果分离到.h和.cpp中，会导致链接错误
- 一般将模板的声明和实现统一放到一个.hpp文件中



main.cpp
main() { }

编译

main.obj
main() { }

链接

exe

Swap.cpp
template

编译

Swap.obj
没有函数实现

```
template <class T> void swapValues(T &v1, T &v2);
```

```
template <class T>
void swapValues(T &v1, T &v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
int a = 10;
int b = 20;
swapValues<int>(a, b);
swapValues(a, b);
```

```
template <class T1, class T2>
void display(const T1 &v1, const T2 &v2) {
    cout << v1 << endl;
    cout << v2 << endl;
}
```

```
display(20, 1.7);
```

```
template <class Item>
class List {
    int m_size;
    int m_capacity;
    Item *m_data;
public:
    List(int capacity = 0);
    ~List();
    void add(Item value);
    Item get(int index);
    int size();
    void display();
};
```

```
template <class Item>
void List<Item>::add(Item value) {
    if (this->m_size == this->m_capacity) {
        cout << "数组已满" << endl;
        return;
    }

    this->m_data[this->m_size++] = value;
}

template <class Item>
Item List<Item>::get(int index) {
    if (index < 0 || index >= this->m_size) return NULL;

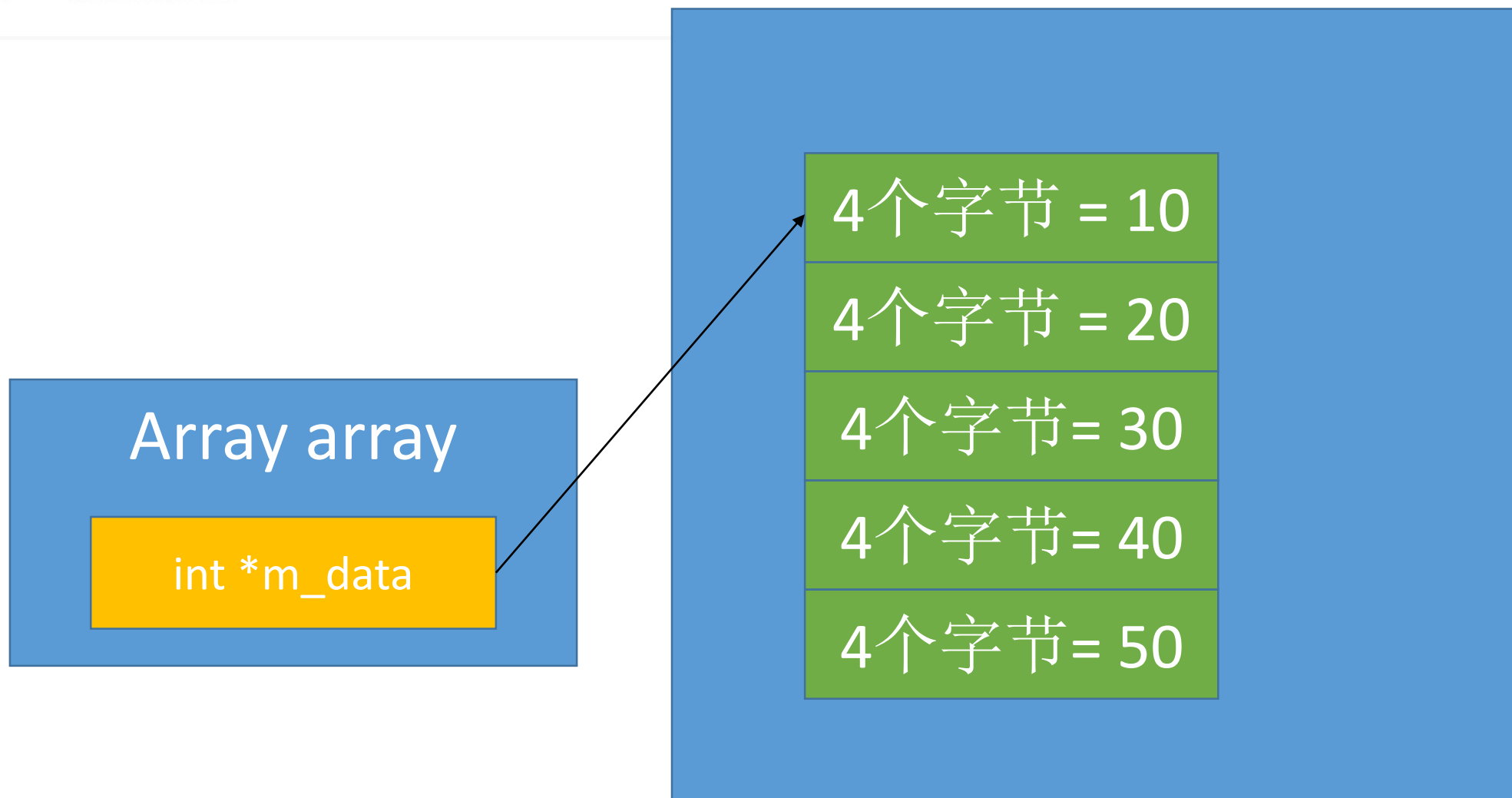
    return this->m_data[index];
}

template <class Item>
int List<Item>::size() {
    return this->m_size;
}
```

类模板中的友元函数

```
template <class Item>
class Array {
    friend ostream &operator<<<>(ostream &, const Array<Item> &);
    int m_size = 0;
    int m_capacity = 0;
    Item *m_data = NULL;
```

```
template <class Item>
ostream &operator<<<>(ostream &cout, const Array<Item> &array) {
    cout << "[";
    for (int i = 0; i < array.m_size; i++) {
        cout << array.m_data[i];
        if (i != array.m_size - 1) {
            cout << ", ";
        }
    }
    return cout << "];"
}
```



■ C语言风格的类型转换符

- `(type) expression`

- `type (expression)`

■ C++中有4个类型转换符

- `static_cast`

- `dynamic_cast`

- `reinterpret_cast`

- `const_cast`

- 使用格式: `xx_cast<type>(expression)`

const_cast

- 一般用于去除const属性，将const转换成非const

```
const Person *p1 = new Person();  
p1->m_age = 10;  
  
Person *p2 = const_cast<Person *>(p1);  
p2->m_age = 20;
```

dynamic_cast

- 一般用于多态类型的转换，有运行时安全检测

```
class Person {  
    virtual void run() { }  
};  
  
class Student : public Person { };  
  
class Car { };
```

```
Person *p1 = new Person();  
Person *p2 = new Student();  
  
Student *stu1 = dynamic_cast<Student *>(p1); // NULL  
Student *stu2 = dynamic_cast<Student *>(p2);  
Car *car = dynamic_cast<Car *>(p1); // NULL
```

static_cast

- 对比dynamic_cast，缺乏运行时安全检测
- 不能交叉转换（不是同一继承体系的，无法转换）
- 常用于基本数据类型的转换、非const转成const
- 使用范围较广

```
Person *p1 = new Person();  
Person *p2 = new Student();  
Student *stu1 = static_cast<Student *>(p1);  
Student *stu2 = static_cast<Student *>(p2);  
Car *car = static_cast<Car *>(p1);
```

reinterpret_cast

- 属于比较底层的强制转换，没有任何类型检查和格式转换，仅仅是简单的二进制数据拷贝
- 可以交叉转换
- 可以将指针和整数互相转换

```
Person *p1 = new Person();
Person *p2 = new Student();
Student *stu1 = reinterpret_cast<Student *>(p1);
Student *stu2 = reinterpret_cast<Student *>(p2);
Car *car = reinterpret_cast<Car *>(p1);

int *p = reinterpret_cast<int *>(100);
int num = reinterpret_cast<int>(p);

int i = 10;
double d1 = reinterpret_cast<double &>(i);
```

C++标准的发展

年份	C++ 标准	名称
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2011	ISO/IEC 14882:2011	C++11
2014	ISO/IEC 14882:2014	C++14
2017	ISO/IEC 14882:2017	C++17
2020	Yet to be determined	C++20

■ auto

- 可以从初始化表达式中推断出变量的类型，大大简化编程工作
- 属于编译器特性，不影响最终的机器码质量，不影响运行效率

```
auto i = 10; // int
auto str = "c++"; // const char *
auto p = new Person(); // Person *
p->run();
```

■ decltype

- 可以获取变量的类型

```
int a = 10;
decltype(a) b = 20; // int
```

■ nullptr

- 可以解决NULL的二义性问题

```
func(0);
func(nullptr);
func(NULL);

cout << (NULL == nullptr) << endl; // 1
```

```
void func(int v) {
    cout << "func(int v) - " << v << endl;
}

void func(int *v) {
    cout << "func(int *v) - " << v << endl;
}
```

// 快速遍历

```
int array[] = { 11, 22, 33, 44, 55 };
for (int item : array) {
    cout << item << endl;
}
```

// 更加简洁的初始化方式

```
int array[] { 11, 22, 33, 44, 55 };
```

Lambda表达式

■ Lambda表达式

□ 有点类似于JavaScript中的闭包、iOS中的Block，本质就是函数

□ 完整结构：[capture list] (params list) mutable exception-> return type { function body }

✓ capture list: 捕获外部变量列表

✓ params list: 形参列表，不能使用默认参数，不能省略参数名

✓ mutable: 用来说用是否可以修改捕获的变量

✓ exception: 异常设定

✓ return type: 返回值类型

✓ function body: 函数体

□ 有时可以省略部分结构

✓ [capture list] (params list) -> return type {function body}

✓ [capture list] (params list) {function body}

✓ [capture list] {function body}

Lambda表达式 – 示例

```
int (*p1)(int, int) = [] (int v1, int v2) -> int {  
    return v1 + v2;  
};  
cout << p1(10, 20) << endl;
```

```
auto p3 = [](int v1, int v2) { return v1 - v2; } (20, 10);  
cout << p3 << endl;
```

```
int exec(int a, int b, int(*func)(int, int)) {  
    if (func == nullptr) return 0;  
    return func(a, b);  
}
```

```
cout << exec(20, 10, [] (int v1, int v2) { return v1 + v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 - v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 * v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 / v2; }) << endl;
```

```
auto p2 = [] (int v1, int v2) {  
    return v1 + v2;  
};  
cout << p2(10, 20) << endl;
```

```
auto p4 = [] {  
    cout << "test" << endl;  
};  
p4();
```

Lambda表达式 - 外部变量捕获

```
int a = 10;
int b = 20;
// 值捕获
auto func = [a, b] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 20
```

```
int a = 10;
int b = 20;
// 隐式捕获 (值捕获)
auto func = [=] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 20
```

```
int a = 10;
int b = 20;
// a是值捕获, 其余变量是地址捕获
auto func = [&, a] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 22
```

```
int a = 10;
int b = 20;
// a是引用 (地址) 捕获, b是值捕获
auto func = [&a, b] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 20
```

```
int a = 10;
int b = 20;
// 隐式捕获 (地址捕获)
auto func = [&] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 22
```

```
int a = 10;
int b = 20;
// a是地址捕获, 其余变量是值捕获
auto func = [=, &a] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 20
```

Lambda表达式 - mutable

```
int a = 10;  
auto func = [a]() mutable {  
    cout << ++a << endl;  
};  
func(); // 11  
cout << a << endl; // 10
```

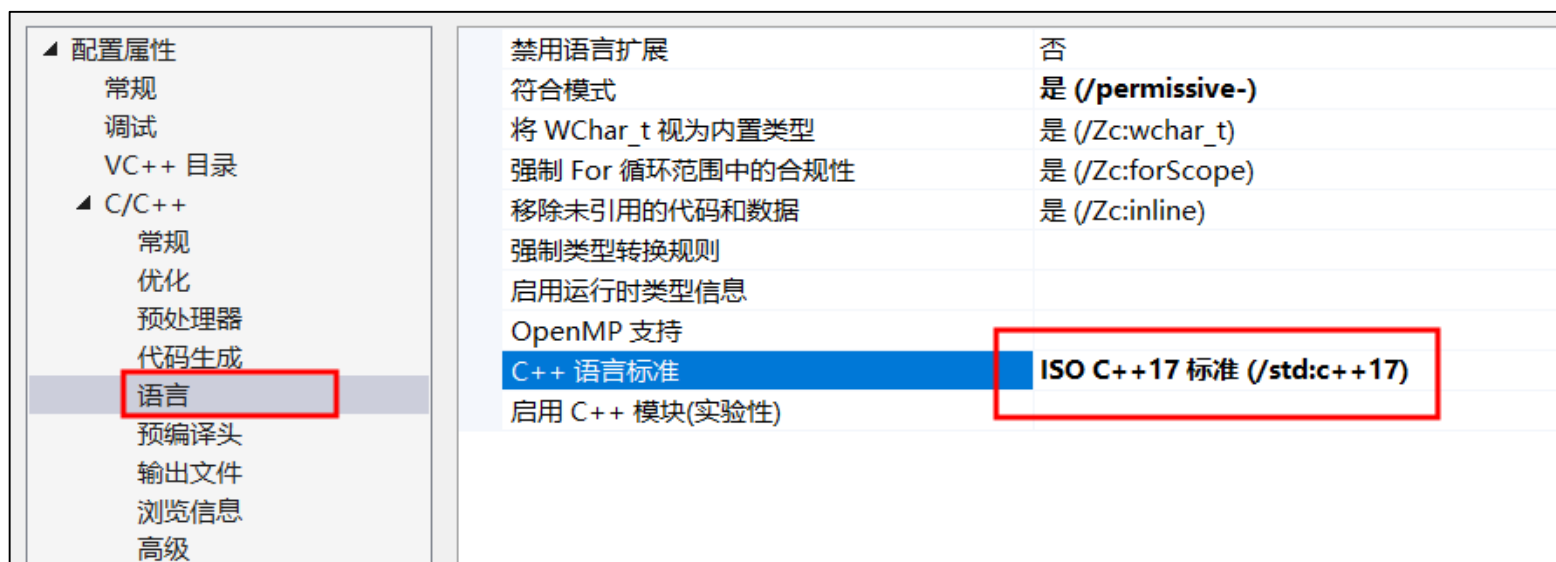
■ 泛型Lambda表达式

```
auto func = [](auto v1, auto v2) { return v1 + v2; };  
cout << func(10, 20.5) << endl;
```

■ 对捕获的变量进行初始化

```
int a;  
auto func = [a = 10]() {  
    cout << a << endl;  
};  
func();  
// 这里仍然是未初始化  
cout << a << endl;
```

■ 设置C++标准



■ 可以进行初始化的if、switch语句

```
// 变量a、b的作用域是它所在的if语句、以及其后面的if-else语句
if (int a = 10; a > 10) {
    a = 1;
} else if (int b = 20; a > 5 && b > 10) {
    b = 2;
    a = 2;
} else if (0) {
    b = 3;
    a = 3;
} else {
    b = 4;
    a = 4;
}
```

```
// 变量a的作用域是它所在switch语句
switch (int a = 10; a) {
    case 1:
        break;
    case 5:
        break;
    case 10:
        break;
    default:
        break;
}
```

- 编程过程中的常见错误类型
 - 语法错误
 - 逻辑错误
 - 异常
 -

- 异常是一种在程序运行过程中发生的不好预测的错误（比如内存不够）
- 异常没有被处理，会导致程序终止

```
int divide(int v1, int v2) {  
    if (v2 == 0) throw "不能除以0";  
    return v1 / v2;  
}
```

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (const char *exception) {  
    cout << exception << endl;  
}  
  
cout << "runing..." << endl;
```

```
try {  
    // 被检测的代码  
} catch (异常类型 [变量名]) {  
    // 异常处理代码  
} catch (异常类型 [变量名]) {  
    // 异常处理代码  
} ...
```

- `throw`异常后，会在当前函数中查找匹配的`catch`，找不到就终止当前函数代码，去上一层函数中查找。如果最终都找不到匹配的`catch`，整个程序就会终止

- 为了增强可读性和方便团队协作，如果函数内部可能会抛出异常，建议函数声明一下异常类型

```
// 抛出任意可能的异常
void func1() {
}

// 不抛出任何异常
void func2() throw() {
}

// 只抛出int、double类型的异常
void func3() throw(int, double) {
}
```

自定义异常类型

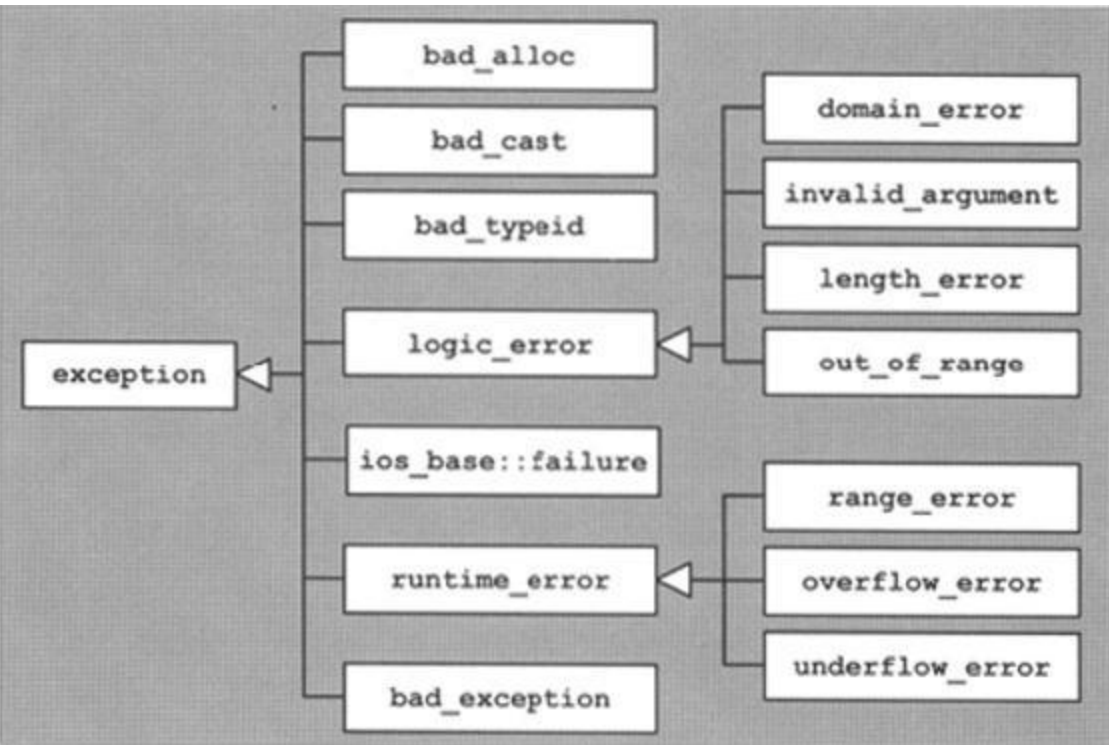
```
class Exception {  
public:  
    virtual string what() const = 0;  
};  
  
class DivideException : public Exception {  
public:  
    string what() const { return "不能除以0"; }  
};  
  
int divide(int v1, int v2) throw(Exception) {  
    if (v2 == 0) throw DivideException();  
    return v1 / v2;  
}
```

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (const Exception &exception) {  
    cout << exception.what() << endl;  
}  
  
cout << "runing..." << endl;
```

拦截所有类型的异常

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (...) {  
    cout << "出现异常" << endl;  
}
```

标准异常 (std)



异常	描述
std::exception	该异常是所有标准 C++ 异常的父类。
std::bad_alloc	该异常可以通过 new 抛出。
std::bad_cast	该异常可以通过 dynamic_cast 抛出。
std::bad_exception	这在处理 C++ 程序中无法预期的异常时非常有用。
std::bad_typeid	该异常可以通过 typeid 抛出。
std::logic_error	理论上可以通过读取代码来检测到的异常。
std::domain_error	当使用了一个无效的数学域时，会抛出该异常。
std::invalid_argument	当使用了无效的参数时，会抛出该异常。
std::length_error	当创建了太长的 std::string 时，会抛出该异常。
std::out_of_range	该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator[]()。
std::runtime_error	理论上不可以通过读取代码来检测到的异常。
std::overflow_error	当发生数学上溢时，会抛出该异常。
std::range_error	当尝试存储超出范围的值时，会抛出该异常。
std::underflow_error	当发生数学下溢时，会抛出该异常。

标准异常 (std)

```
try {  
    int size = 999999;  
    for (size_t i = 0; i < size; i++) {  
        int *p = new int[size];  
    }  
} catch (std::bad_alloc exception) {  
    cout << exception.what() << endl;  
}  
  
cout << "runing..." << endl;
```

智能指针 (Smart Pointer)

■ 传统指针存在的问题

- 需要手动管理内存
- 容易发生内存泄露 (忘记释放、出现异常等)
- 释放之后产生野指针

■ 智能指针就是为了解决传统指针存在的问题

- `auto_ptr`: 属于C++98标准, 在C++11中已经不推荐使用 (有缺陷, 比如不能用于数组)
- `shared_ptr`: 属于C++11标准
- `unique_ptr`: 属于C++11标准

智能指针的简单自实现

```
template<class T>
class SmartPointer {
    T *m_pointer;
public:
    SmartPointer(T *pointer) :m_pointer(pointer) { }

    ~SmartPointer() {
        if (m_pointer == nullptr) return;
        delete m_pointer;
    }

    T *operator->() {
        return m_pointer;
    }
};
```

■ shared_ptr的设计理念

□ 多个shared_ptr可以指向同一个对象，当最后一个shared_ptr在作用域范围内结束时，对象才会被自动释放

■ 针对数组的用法

```
shared_ptr<Person> ptr1(new Person[5]{} , [](Person *p) { delete[] p; });
```

```
shared_ptr<Person[]> persons(new Person[5]{});
```


shared_ptr的原理

- 一个shared_ptr会对一个对象产生强引用 (strong reference)
- 每个对象都有个与之对应的强引用计数，记录着当前对象被多少个shared_ptr强引用着
 - 可以通过shared_ptr的use_count函数获得强引用计数
- 当有一个新的shared_ptr指向对象时，对象的强引用计数就会+1
- 当有一个shared_ptr销毁时（比如作用域结束），对象的强引用计数就会-1
- 当一个对象的强引用计数为0时（没有任何shared_ptr指向对象时），对象就会自动销毁（析构）

shared_ptr的注意事项

- 不要使用裸指针来初始化智能指针，比如以下代码

```
Person *p = new Person(); // Person()

{
    shared_ptr<Person> p1(p);
} // ~Person()

{
    shared_ptr<Person> p2(p);
} // ~Person()
```

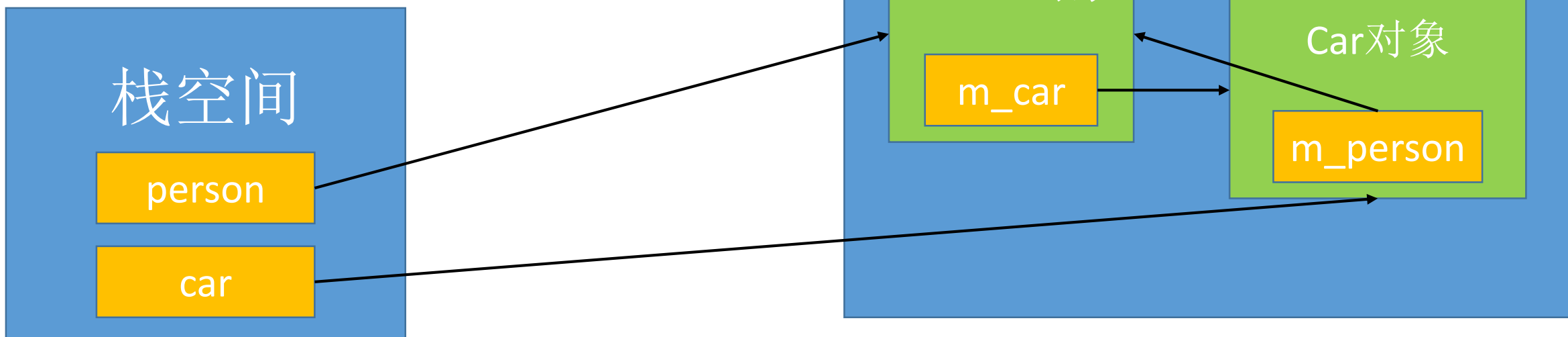
- 可以通过一个已存在的智能指针初始化一个新的智能指针

```
shared_ptr<Person> p1(new Person());
shared_ptr<Person> p2(p1);
```

shared_ptr的循环引用

```
class Car {  
public:  
    shared_ptr<Person> m_person = nullptr;  
};  
  
class Person {  
public:  
    shared_ptr<Car> m_car = nullptr;  
};
```

```
shared_ptr<Person> person(new Person());  
shared_ptr<Car> car(new Car());  
person->m_car = car;  
car->m_person = person;
```



weak_ptr

■ weak_ptr 会对一个对象产生弱引用,

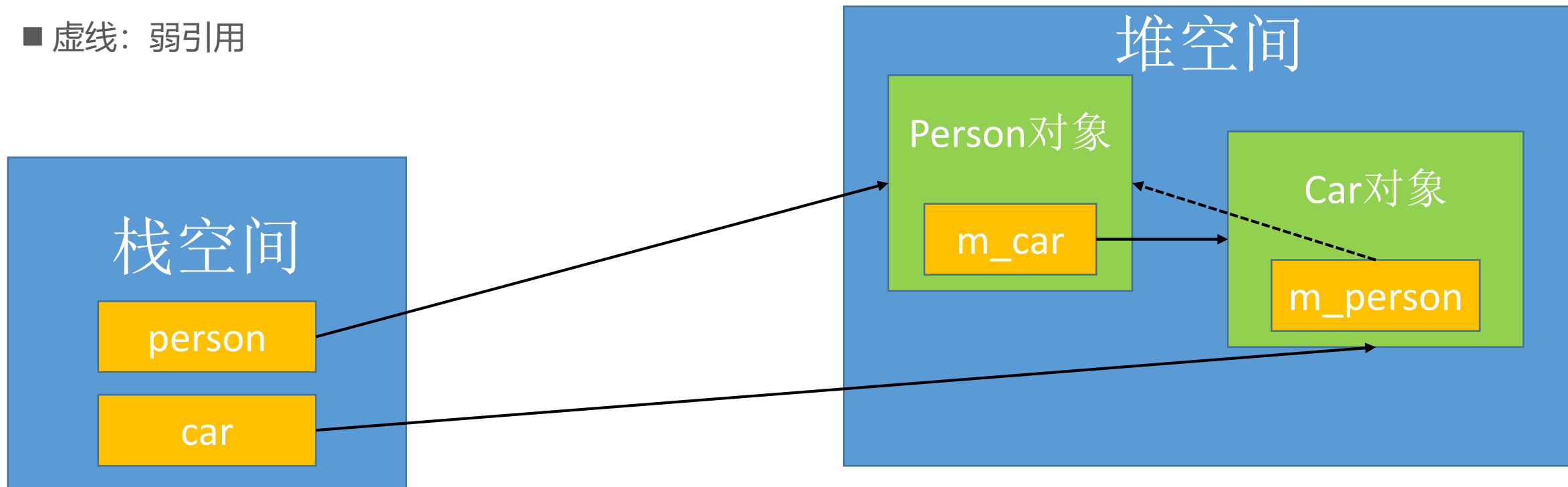
■ weak_ptr 可以指向对象解决 shared_ptr 的循环引用问题

```
class Car {  
public:  
    weak_ptr<Person> m_person;  
};  
  
class Person {  
public:  
    shared_ptr<Car> m_car = nullptr;  
};
```

```
shared_ptr<Person> person(new Person());  
shared_ptr<Car> car(new Car());  
person->m_car = car;  
car->m_person = person;
```

weak_ptr解决循环引用

- 实线：强引用
- 虚线：弱引用



unique_ptr

- `unique_ptr`也会对一个对象产生强引用，它可以确保同一时间只有1个指针指向对象
- 当`unique_ptr`销毁时（作用域结束时），其指向的对象也就自动销毁了
- 可以使用`std::move`函数转移`unique_ptr`的所有权

```
// ptr1强引用着Person对象
unique_ptr<Person> ptr1(new Person());
// 转移之后, ptr2强引用着Person对象
unique_ptr<Person> ptr2 = std::move(ptr1);
```