Practice of Programming 5
Point and Reference

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

内容提纲

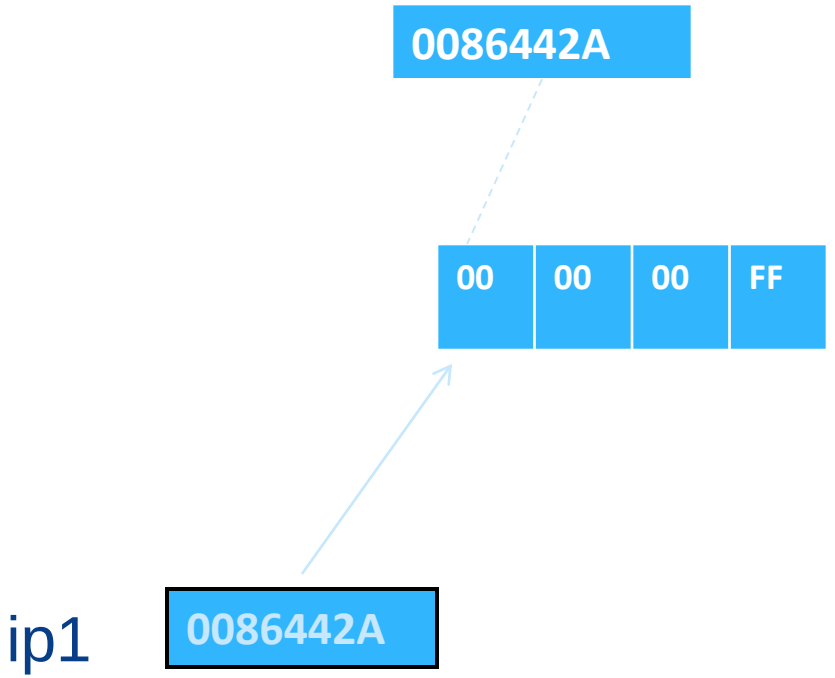
- 回顾
  - 指针类型
  - 引用类型
  - 数组类型
- 设计原则
  - 以传递常量引用方式替代传值方式
  - 返回引用时要谨慎

指针类型

- C++可以在运行时获取变量或者对象的地址，并且通过地址操纵数据

- 指针定义

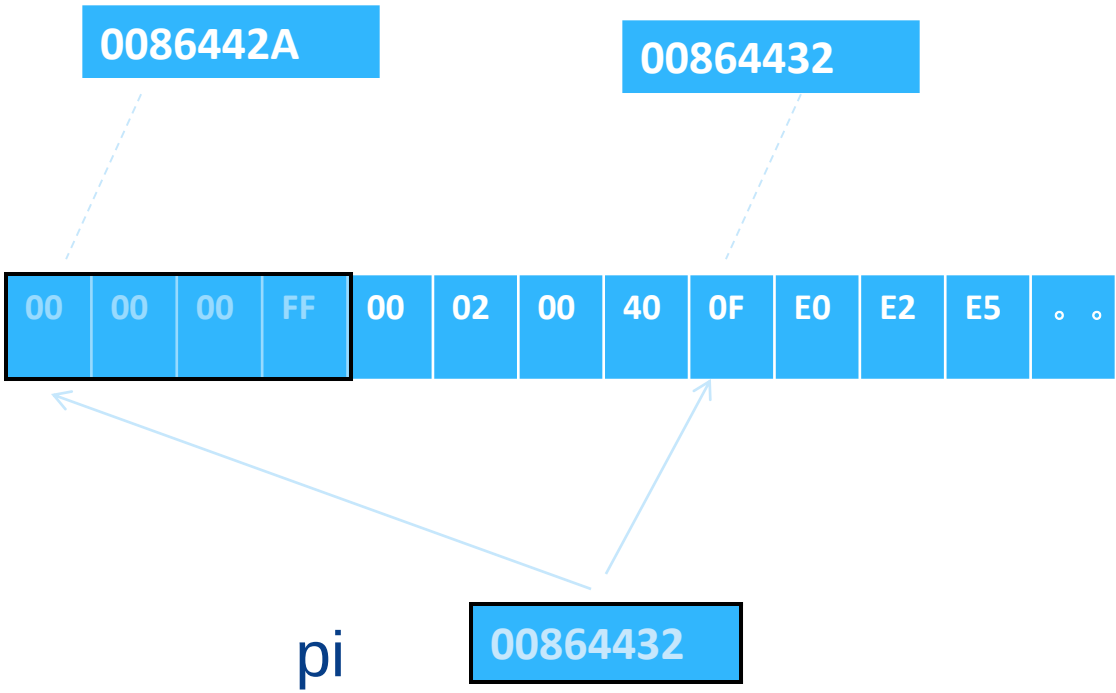
```
int *ip1, *ip2;
string *pstring;
double *dp;
long *lp, lp2;

int *ip1;
int i = 255;
ip1 = &i;
```



指针类型

Example 1：指针

```
int i=255;
int *pi = &i;
*pi = *pi + 2;
    // i 加2 (i = i + 2)
pi = pi + 2;
    // 加到pi 包含的地址上
```



指针赋值

- 指针不能被初始化或赋值为其他类型对象的地址值（&obj）

```
int  *pi = 0;
double  dval;
pi = &dval;   //error,无效的类型赋值: int* ← double*
```

- 空（void* ）类型指针表明这个是个指针，但是指向的数据类型不明。因此可以被任何数据指针类型的地址值赋值。

```
void *pv = pi;
pv = pd;
```

指针赋值

Example 1：指针

```
int i, j, k;
int *pi = &i;
*pi = *pi + 2;
    // i 加2 (i = i + 2)
pi = pi + 2;
    // 加到pi 包含的地址上
```

Example 2：指针

```
int ia[ 10 ];
int *iter = &ia[0];
int *iter_end = &ia[10];

while ( iter != iter_end ) {
    do_something_with_value( *iter );

    // 现在iter 指向下一个元素
    ++iter;
}
```

常量指针与指针常量

- 指针常量指指针指向不能改变的指针
  - int * const cp
  - int errNumb=0;

    int *const curErr=&errNumb;

    if (*curErr) {

     errorHandler();

    *curErr = 0;} //OK

    curErr = &myErrNumb; //Error
- 常量指针指指向常量的指针
  - const int * cp 或者 int const * cp
  - const int * const cp； //常量指针常量

    *cp=+2；//Error

引用类型

- 引用又叫别名
  - 引用类型由类型标识符和一个取地址操作符&来定义.
- 引用必须被初始化

```
int ival = 1024;
int &refVal = ival;      // ok: refVal 是一个指向ival 的引用
int &refVal2;                    // Error
```

- 引用与指针的主要区别
  - 同一指针可以指向不同实体，而引用在被初始化后，与实体之间的联系则已经建立，不能修改
  - 使用引用访问数据变得更加安全

引用类型


```
int ival = 1024;
int &refVal = ival;
```

ival (refval)

引用类型

- Example: references

```
int ival = 1024;
int &refVal = &ival;          // 错误，refVal 是int 类型, 不是int*
int *pi = &ival;
int *&ptrVal2 = pi;           // ok: refPtr 是一个指向指针的引用

int min_val = 0;
refVal = min_val;             // ival 被设置为min_val 的值
                              //refVal 并没有引用到min_val 上
refVal += 2;

int ii = refVal;              //把与 ival 相关联的值赋给 ii
int  *pi = &refVal;           //用ival 的地址初始化pi
```

引用类型

- 引用类型主要被用作函数的形式参数

```
bool get_next_value( int &next_value );
// 返回访问状态,将值放入参数

Matrix operator+( const Matrix&, const Matrix& );
// 重载加法操作符
```

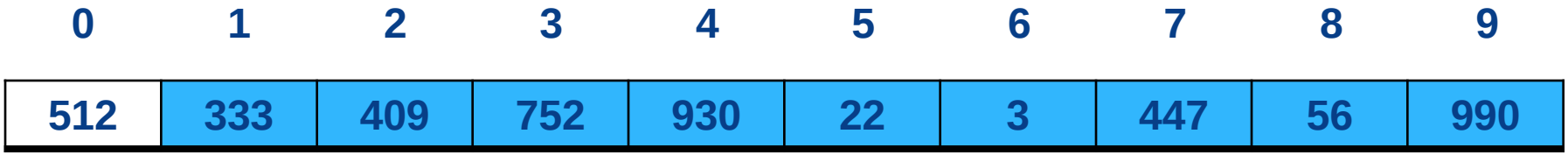
数组类型

- 数组用来存储相同类型的一组数据
  - int test_scores[];

- 数组元素是从0 开始计数的

```
int main()
{
    const int array_size = 10;
    int ia[ array_size ];
    for ( int ix = 0; ix < array_size; ++ix )
        ia[ ix ] = ix;
}
```

数组类型



数组类型

- 维数值必须是常量表达式——即必须能在编译时确定值

```
extern int get_size();
const int buf_size = 512, max_files = 20;
int staff_size = 27;

char input_buffer[ buf_size ];
char *fileTable[ max_files - 3 ];
double salaries[ staff_size ]; // 错误: 非const 变量
int test_scores[ get_size() ]; // 错误: 非const 表达式
```

数组类型

- 数组可以被显式地用一组数来初始化
  - const int array_size = 3;
    int ia[array_size] = { 0, 1, 2 };
  - int ia[] = { 0, 1, 2 };
  - const int array_size = 5;
    int ia[array_size] = { 0, 1, 2 }; // ia[]={ 0, 1, 2, 0, 0 }
  - const char ca1[] = { 'C', '+', '+' }; //维数是3
  - const char ca2[] = "C++"; //维数是4
  - const char ch3[6] = "Daniel"; // 错误: "Daniel"是7 个元素

数组类型

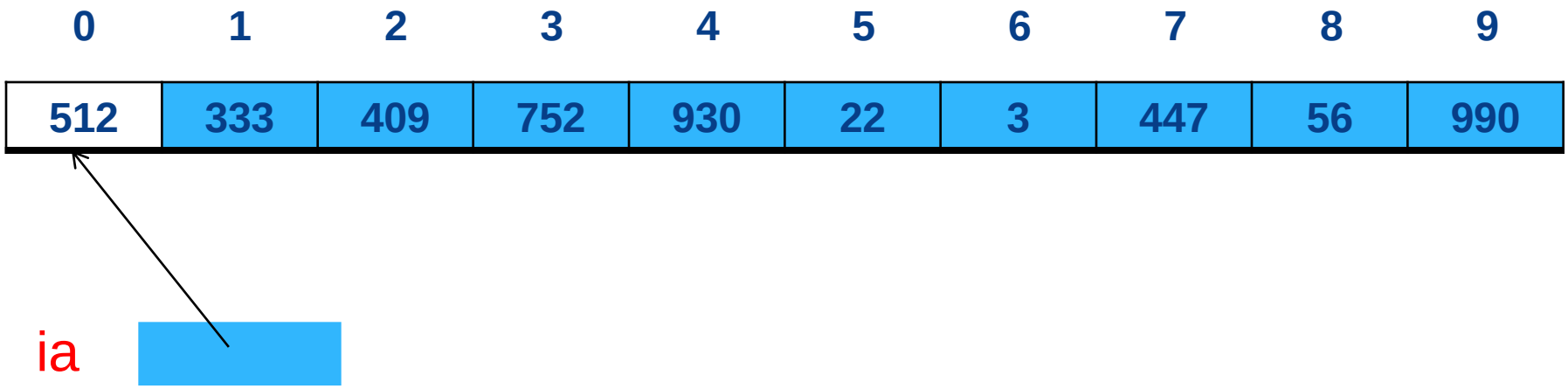
- 一个数组不能被另外一个数组初始化，也不能被赋值给另外一个数组，而且C++不允许声明一个引用数组

```
const int array_size = 3;
int ix, jx, kx;
int *iap [] = { &ix, &jx, &kx };          // ok: 类型为int*的指针的数组
int &iar[] = { ix, jx, kx };              // 错误: 不允许引用数组
int ia2[] = ia;                  // 错误: 不能用另一个数组来初始化一个数组
```

- 要把一个数组拷贝到另一个中去，必须按顺序拷贝每个元素

数组类型

- 数组与指针类型关系
  - int ia[]={...}　　//ia代表了数组的第一个元素的地址
  - *ia==ia[0]
  - ia+1==&ia[1]
  - ia[5] == *(ia+5)



多维数组

- 定义
  - int ia[4][3]={{0},{1},{2},{3}}
  - int ia[4][3]={{0,0,0},{1,0,0},{2,0,0},{3,0,0}}
  - int ia[4][3]={0,0,0,1,0,0,2,0,0,3,0,0}

0	0	0	1	0	0	2	0	0	3	0	0

- 使用
  - **ia[2][2]=1 ➔ *(ia[2]+2)=1**

多维数组


```
int ia[4][3][2]={
{{0,1},{0,3},{0,5}},
{{1,0},{0,2},{0,4}},
{{2,3},{3,0},{6,0}},
{{3,7},{0,9},{0,2}}}
```

动态存储设施

- 这些设施包含两种操作：

- new
  - 为某种类型的对象保留内存空间
  - 初始化对象
  - 并返回指向该对象的指针

- delete
  - 传递给它一个指向用new创建的对象的指针
  - 销毁该对象
  - 并释放该对象先前占用的内存空间

new 和 delete

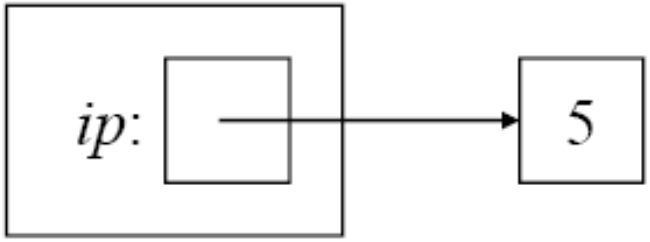
- 下面是一个示例：
  - int *ip = new int;
- 这将为一个整数创建新的内存空间
  - 并返回指向该空间的地址，将其赋给ip
- 注意，我们还未对该整数做任何初始化
  - 任何随机的适合int表示的整数都是合法的值

new 和 delete

- 可以使用“初始化器(initializer)”强制初始化
  - int *ip = new int(5);
- 这样会为新的整数创建内存空间，并将其初始化为5
  - 然后返回指向这个空间的指针
- 这里容易混淆的地方：
  - 在上面的语句中包含两个对象

new 和 delete

1. ip，一个指向整数的指针类型的变量
2. ip指向的对象，它具有整数类型
   - 这是一个动态分配的内存空间
   - 并且存活于某个地方
   - 它没有自己的名字——它就是由ip指向的整数



对象创建

- 当按照这种方式创建类的实例时
  - 构造器会被调用，但看起来就像以普通方式创建一样

- 例如：
  - 如果要创建一个新的IntSet
  - IntSet *isp = new IntSet
  - 将会执行下面的操作

对象创建

- 从堆中分配足够的内容去持有一个IntSet对象
  - 这需要用到IntSet类的定义
    - 一个包含100个整数的数组(elts)
    - 一个持有集合尺寸的整数(numElts)

- 在这个“新生”的对象上调用构造器IntSet::IntSet()
  - 在IntSet的具体实现中，构造器将numElts设置为0
  - 表明elts数组中还没有任何元素

对象删除

- 普通的对象可以使用delete删除
  - 只要它们是通过new创建的
- 就像内建(built-in)类型的初始化不做任何操作一样
  - 销毁它们也不做任何操作
- 但是，这会释放所占用的内存空间
  - 使得它可以被其他调用new()创建的对象重用

对象删除

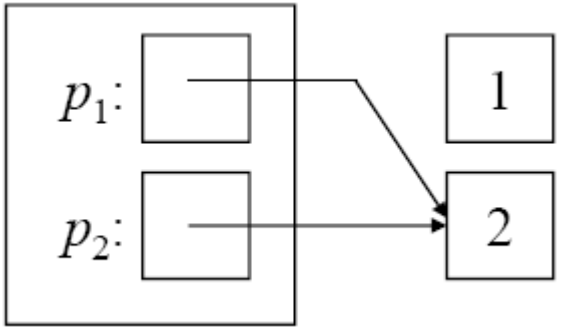
- 我们也可以销毁用new创建的ADT类型
  - delete isp;
- 在此例中(删除一个IntSet)
  - IntSet只包含普通类型的成员变量(int和int数组)
  - 因此也不需要做任何额外的操作去销毁它
- 但是，并非所有的ADT实例销毁事件都是如此
- 因此，我们会发现
  - 就像使用构造器创建对象一样
  - 我们有时需要使用析构器(destructor)来恰当地销毁对象

对象删除

- 注意：
  - 对象的生命周期完全在程序的控制之下
  - 它一直生存到明确地销毁它 (或者到程序结束)
- 即便你忘记了指向该对象的指针，也会如此
  - 例如

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
```

- 这会使内存像右图一样



对象删除

- 两个指针指向对象"2"
  - 并且没有任何指针指向对象 *"1"*

- 没有任何方式释放 “1”占用的内存了
- 并且，更糟糕的是:

```
delete p1;
delete p2;
```

- 将“释放”2占用的内存两次

- 这几乎可以肯定会带来不良的后果

对象删除

- 注意，这里有个概念需要区分
  - 指针变量的生命周期
  - 与它指向的对象的生命周期不同!
- 在前面的例子中
  - 退出定义p1的语句块会导致局部变量p1消失
  - 但是它所指向的动态对象仍旧存在
- 这会留下一个动态分配的对象
  - 没有任何方式可以回收它

内存泄露

- 这称为内存泄露(*memory leak*)
  - 如果它发生的频率足够频繁
  - 那么程序就可能会到达一种状态
  - 在该状态，程序将再也不能分配新的动态对象

堆(heap)

- 用于通过new()创建的对象的内存空间来自于
  - 内存中被称为堆的地方

- 首先探讨典型的C++进程所使用的内存模型
  - 每个运行的程序都有一个地址空间
    - 该程序可以访问的一组内存位置
  - 地址空间对于一个运行的程序是私有的
    - 其他任何运行的程序都不能访问/修改它

地址空间

- 在地址空间中，典型地有5个部分，被成为段“segments”：

地址空间

- 由编译过的程序构成的代码位于文本段 text segment
  - 它位于地址空间中非常低的地址
    - 但是通常不是从地址 0 开始
- 紧挨其上是编译器为全局变量分配的空间
  - 必要时会对它们进行初始化
- 对于通过new()分配的对象，位于堆中
  - 它可以向上扩展
- 当函数被调用时，会在栈上创建栈内存帧
  - 它可以向下扩展

析构器(Destructor)

- 析构器是构造器的对立面
- 构造器确保
  - 对象是合法的类的实例
- 析构器的工作于此相反

- 在许多类中，析构器不必做任何事
- 但是，在构造器(或其他方法)分配了动态存储空间的类中
  - 析构器就必须负责回收这些空间

- 析构器可以声明为：

```
~IntSet(); // Destroy this IntSet
```

带有析构器的IntSet


```
class IntSet {
   int *elts; // pointer to dynamic array
   int sizeElts  // capacity of array
   int numElts; // current occupancy
 public:
   IntSet(int size = MAXELTS);
   // EFFECTS: create a set with size capacity;
   // capacity is MAXELTS by default.
   ~IntSet(); // Destroy this IntSet
   ...
```

带有析构器的IntSet

- 析构器编码如下：

```
IntSet::~IntSet()
{
   delete[] elts;
}
```

- 注意，必须使用基于数组的delete操作符
  - 不能使用标准的delete操作符
- 因为用new[]创建的对象只能用delete[]删除

带有析构器的IntSet

- 现在，当IntSet被销毁时
  - 数组元素将首先被删除

- 注意：
  - 任何在代码块内局部声明的ADT对象，其析构器都
    - 会在块执行结束的时候自动被调用
    - 变量作用域结束

带有析构器的IntSet

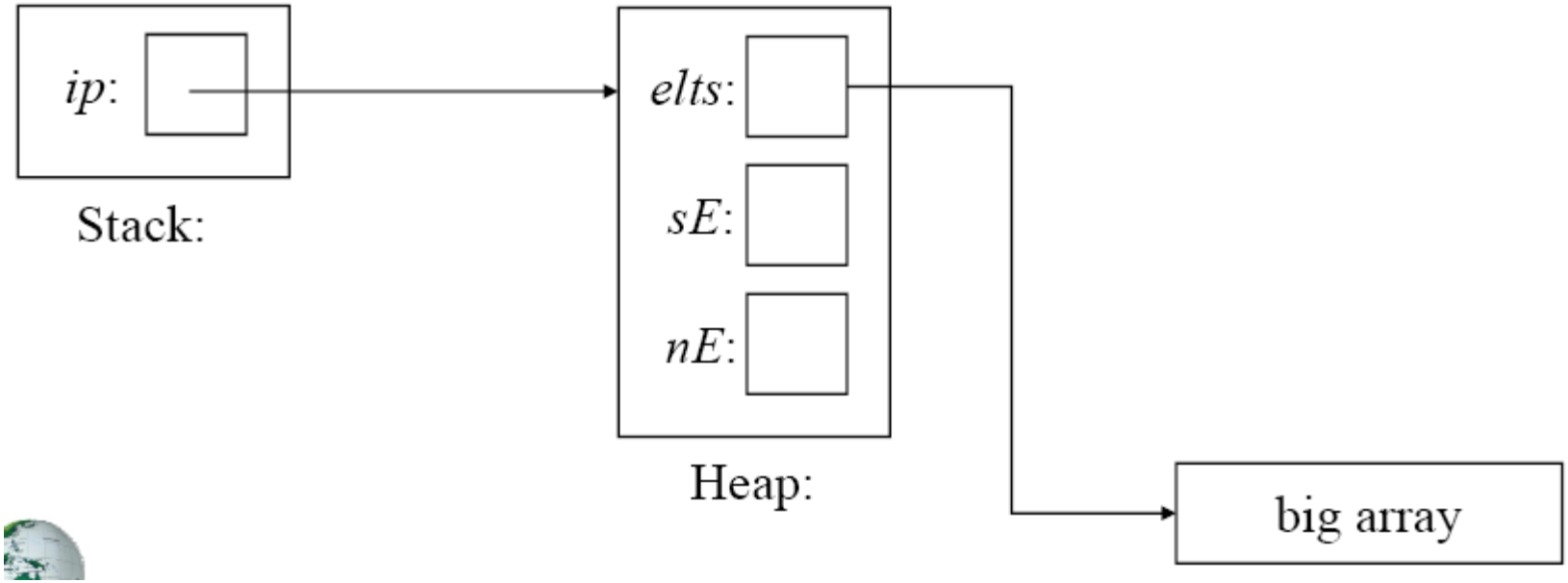
- 新的IntSet定义可以动态地创建和销毁
  - 就像其他类型一样

- 因此：

```
IntSet *ip = new IntSet(50); // a non-standard size
   // do stuff
delete ip; // Destroys the IntSet
```

带有析构器的IntSet

- 在集创建之后，我们可以
  - 分配空间以持有IntSet(一个指针和两个整数)
  - 调用对象上的构造器
    - 分配用于50个整数的数组空间



自由存储小结

- 使用new来分配空间
  - new会在自由存储上创建一个对象，有时候对象会被初始化，并且返回指向这个对象的指针。

```
int* pi = new int;                  // 默认初始化 (值未知)
char* pc = new char('a');           // 显式初始化
double* pd = new double[10];        // 分配未初始化数组
```

  - 如果分配工作无法完成，new将抛出一个bad_alloc异常

自由存储小结

- 使用delete和delete[ ]释放空间
  - delete和delete[ ]把使用new操作分配的内存空间还给自由存储，释放的空间可以用于新的分配

```
delete pi; // 释放单个对象
delete pc; // 释放单个对象
delete[ ] pd;          // 释放数组
```

  - 释放零值的指针（null指针）的操作不会做任何事情

```
char* p = 0;
delete p;  // 无害的
```

以传递常量引用方式替代传值方式


```
class Person {
public:
  Person();                    // parameters omitted for simplicity
  virtual ~Person();
  ...
private:
  std::string name;
  std::string address;
};

class Student: public Person {
public:
  Student();                   // parameters again omitted
  ~Student();
  ...
private:
  std::string schoolName;
  std::string schoolAddress;
};
```

以传递常量引用方式替代传值方式


```
bool validateStudent(Student s);        // function taking a Student   by value
Student plato;                          // Plato studied under Socrates
bool platoIsOK = validateStudent(plato);   // call the function
```

- 函数调用时，需要复plato给s
  - 一次Person构造器调用
  - 一次Studen构造器调用
  - 四次string构造器调用

```
bool validateStudent(const Student& s);
```

- 按常量引用方式调用就无需复制

以传递常量引用方式替代传值方式

- 按常量引用方式传递还可以解决切片问题

```
class Window {
public:
  ...
  std::string name() const;          // return name of window
  virtual void display() const;      // draw window and contents
};

class WindowWithScrollBars: public Window {
public:
  ...
  virtual void display() const;
};
```

以传递常量引用方式替代传值方式


```
void printNameAndDisplay(Window w)      // incorrect! parameter
{                                       // may be sliced!
  std::cout << w.name();
  w.display();
}

WindowWithScrollBars wwsb;
printNameAndDisplay(wwsb);
```

- wwsb在传递给printNameAndDisplay时，会当做Window复制给w
  - 切片！

```
void printNameAndDisplay(const Window& w)
```

- OK！

以传递常量引用方式替代传值方式

- 对于内置类型，例如int、float、bool等，并不适用
- 对于复制代价比较小的class类型，也并不一定需要用

返回引用时要谨慎


```
class Rational {
public:
  Rational(int numerator = 0,
           int denominator = 1);
  ...
private:
   int n, d;                          // numerator and denominator
 friend
   const Rational &   operator*(const Rational& lhs,  const Rational& rhs){
     Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
     return result;
  };
};

Rational a(1, 2);                     // a = 1/2
Rational b(3, 5);                     // b = 3/5
Rational c = a * b;                   // c should be 3/10
```

返回引用时要谨慎


```
inline const Rational operator*(const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

- inline函数从源代码层看，有函数的结构，而在编译后，却不具备函数的性质。编译时，类似宏替换，使用函数体替换调用处的函数名。
- 因此，这里会在被调用地方直接创建新的对象。

Thank You!