Practice of Programming 5
dynamic structure

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

- 固定尺寸结构
- 全局与局部变量
- 动态尺寸结构
- 新建和删除
- 堆
- 函数重载
- 析构器

固定尺寸结构

- 目前为止我们构建的数据结构都能够存放“至多N个”元素
- 例如：
  - 大富翁棋盘至多有40个格子
  - 各种 IntSet 实现至多能存放100个不同的整数
  - 扑克牌游戏中至多存放52张牌

- 因此，我们希望能够扩展它们的尺寸
  - 但目前我们只了解如何创建静态的、固定尺寸的结构

- 我们必须声明这些类型的变量能够存放多大的数据
  - 并且需要编写代码处理溢出的情况（例如IntSet）

固定尺寸结构

- 有时，我们建模的处理存在物理极限
  - 这使得静态的、固定尺寸的结构是合理的选择
- 例如，扑克牌的牌面有52种情况
  - 因此，它就有一个明确的上限

固定尺寸结构

- 但是，上限有时是没有意义的
  - 对于一个“整数集”来说，限制其上限没有任何意义
- 因此，无论我们将这个集的容量设置成多大
  - 某个应用最终可能还是需要更大的容量
- 这个问题对于分配内存空间是至关重要的
  - 例如，对变量分配空间
- 有两类这样的变量
  - 全局变量和局部变量

全局变量

- 在函数定义之外定义的变量
- 在程序开始执行之前，会为这些变量预留内存空间
  - 并将一直保留这些空间，直至程序执行完毕
- 这部分空间是在编译时刻保留的

局部变量

- 在块(block)中定义的所有变量
- 包括函数的参数
  - 它们就像是在函数最外层的块中定义的变量一样

- 当执行到相关的块时，会为这些变量预留内存空间
  - 并将一直保留这些空间，直至块执行完毕

- 这部分空间是在运行时刻保留的，但是编译器知道其尺寸

静态信息

- 全局变量和局部变量都是静态的
  - 因为编译器必须知道它们的确切尺寸
- 静态信息必须由程序员声明

动态尺寸结构

- 我们可以创建第三种类型的对象
  - 动态(dynamic)类型

- 它们的动态性是相对于编译器而言的：
  - 编译器不知道它们有多大
  - 编译器不知道它们会存活多久

动态尺寸结构

- 例如：

- IntSet的尺寸应该能够根据用户的需求变大
  - 仅受制于物理机器的上限
- IntSet的生命周期应该和用户的需求一样长
  - 用户负责销毁

- 这是通过语言机制中动态存储设施(Dynamic Storage Facilities)实现的

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

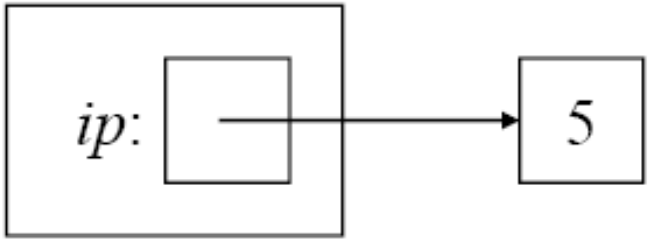
- 下面是一个示例：
  - int *ip = new int;
- 这将为一个整数创建新的内存空间
  - 并返回指向该空间的地址，将其赋给ip
- 注意，我们还未对该整数做任何初始化
  - 任何随机的适合int表示的整数都是合法的值

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

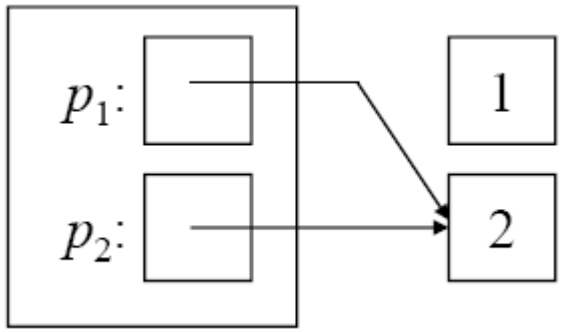
- 我们也可以销毁用new创建的ADT类型
  - delete isp;
- 在此例中(删除一个IntSet)
  - IntSet只包含普通类型的成员变量(int和int数组)
  - 因此也不需要做任何额外的操作去销毁它
- 但是，并非所有的ADT实例销毁事件都是如此
- 因此，我们会发现
  - 就像使用构造器创建对象一样
  - 我们有时需要使用析构器(destructor)来恰当地销毁对象

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

尺寸未知的对象

- 目前为止，我们动态创建的对象都具有确定的尺寸
  - 编译器知道该尺寸
- 例如：
  - 持有一个整数所需的内存数量很容易计算
  - 只要给出恰当的类型定义即可
- 但是，我们也可以创建尺寸未知的对象
  - 编译器不了解其尺寸
  - 可以通过创建动态数组来实现

动态数组

- 创建动态数组的语法与创建单个元素的语法很相似

- 内存分配工作正如我们所期望的那样

  int *ia = new int[5]

  – 这会在堆中创建一个包含5个整数的数组
  – 并在ia中存储一个指向该数组第一个元素的指针
- 与单个整数一样
  – 整数数组不必初始化

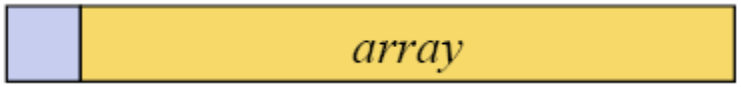
动态数组

- 释放一个数组的工作与释放单个对象略有差异
- 释放数组：

  delete[] ia;
- 如果分配了一个 T 类型的数组，那么
  - 必须使用delete[]操作符来释放它
  - 一般的delete操作符无法实现此目的
- 概念上讲，数组分配器与单例分配器
  - 完全不同
  - 混用会导致未定义的行为

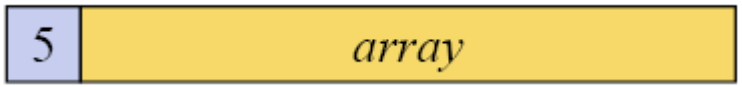
动态数组

- 这是因为
  - 语言的运行时系统必须跟踪数组的尺寸
    - 因为(通常)编译器无法知晓
  - 这个信息是动态的
- 当new操作符发现需要分配数组时
  - 它会将数组的尺寸和数组一起存储
- 它会从数组的空间中辟出一块

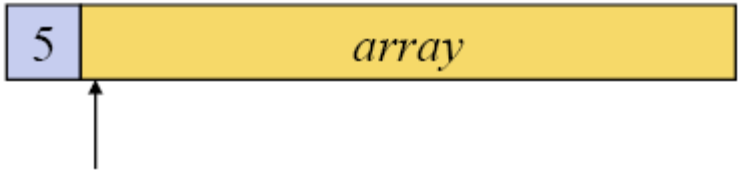


动态数组

- 在数组之前的空间中记录了数组中元素的位置



- 返回的指针指向数组的首元素



- 现在，如果只是delete ia
  - delete操作符就会认为它只需要释放足够在堆上存放一个整数的空间即可
- delete[]操作符则可以看到紧挨指针之前的信息
  - 即看到需要将多少元素的内存返回给堆

使用动态数组的IntSet

- 我们可以构建一个新的IntSet版本
  - 它可以让用户指定集的容量应该有多大
- 数据成员将会进行一些修改：

```
class IntSet {
    int *elts;      // pointer to dynamic array
    int sizeElts;  // capacity of array
    int numElts; // current occupancy
  public:
    ...
};
```

使用动态数组的IntSet

- 现在，不是明确地持有一个数组
  - 而是持有一个指针，它指向动态创建的数组
- sizeElts表示分配的数组尺寸
  - 它现在不必是MAXELTS了
- numElts仍旧表示实际上包含多少元素
- 所有这些改动都基于对集中元素不排序的假设

使用动态数组的IntSet

- 所有的方法几乎都不需做修改
- 缺省构造器需要修改一番

```
IntSet::IntSet()
{
    // Allocate the default-size array
    elts = new int[MAXELTS];
    sizeElts = MAXELTS;
    numElts = 0;
}
```

使用动态数组的IntSet

- 除了缺省构造器
  - 我们还可以编写一个可替代构造器

- 它与缺省构造器具有相同的名字
  - 但是具有不同的类型签名

```
IntSet::IntSet(int size); // constructor w/ explicit capacity
 // REQUIRES: size > 0
 // EFFECTS: create a set with size capacity
...
```

IntSet类


```
class IntSet {
   int *elts; // pointer to dynamic array
   int sizeElts  // capacity of array
   int numElts; // current occupancy
  public:
   IntSet::IntSet(); // default constructor
    // EFFECTS: create a set with MAXELTS capacity
   IntSet::IntSet(int size); // constructor w/ explicit capacity
    // REQUIRES: size > 0
    // EFFECTS: create a set with size capacity
   ...
```

构造器重载

- 重载的构造器将创建一个具有指定尺寸的数组：

```
IntSet::IntSet(int size)
{
    elts = new int[size];
    sizeElts = size;
    numElts = 0;
}
```

- 这属于函数重载

构造器重载

- 由于编译器知道引元的具体类型
  - 在创建新对象时，它会选择正确的构造器

- 例如：

```
IntSet is1; // No arguments---calls the default  constructor
IntSet is2(200);  // An integer argument---calls alternate
```

- 注意，这两个构造器几乎一样
  - 唯一的差别是我们是否使用缺省尺寸MAXELTS

构造器重载

- 这种代码显得比较浪费！
- 因为我们在重复编码
  - 此时应该使用参数泛化机制(parametric generalization)
- 解决此问题的一种方法被称为
  - 缺省引元(default argument)
- 要实现此目的，我们可以只定义一个构造器
  - 但是它的引元是可选的

缺省引元

- 重新声明IntSet构造器：

```
class IntSet {
    int *elts; // pointer to dynamic array
    int sizeElts; // capacity of array
    int numElts  // current occupancy
  public:
    IntSet::IntSet(int size = MAXELTS);
    // EFFECTS: create a set with the specified capacity
    // capacity defaults to MAXELTS if not supplied
```

析构器(Destructor)

- 我们仍然有一个问题没解决

- 如果定义了一个局部IntSet，当函数返回时，会发生什么？
- 为什么这个问题在使用静态数组的IntSet版本中不存在呢？

```
void foo()
{
    IntSet is2(300)  // Work with is2 in some way
```

析构器(Destructor)

- 要解决由此引发的内存泄露
  - 必须执行整数数组的资源回收操作
  - 当拥有它的IntSet对象被销毁时
- 可以通过析构器来实现此目的

- 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

析构器(Destructor)

- 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

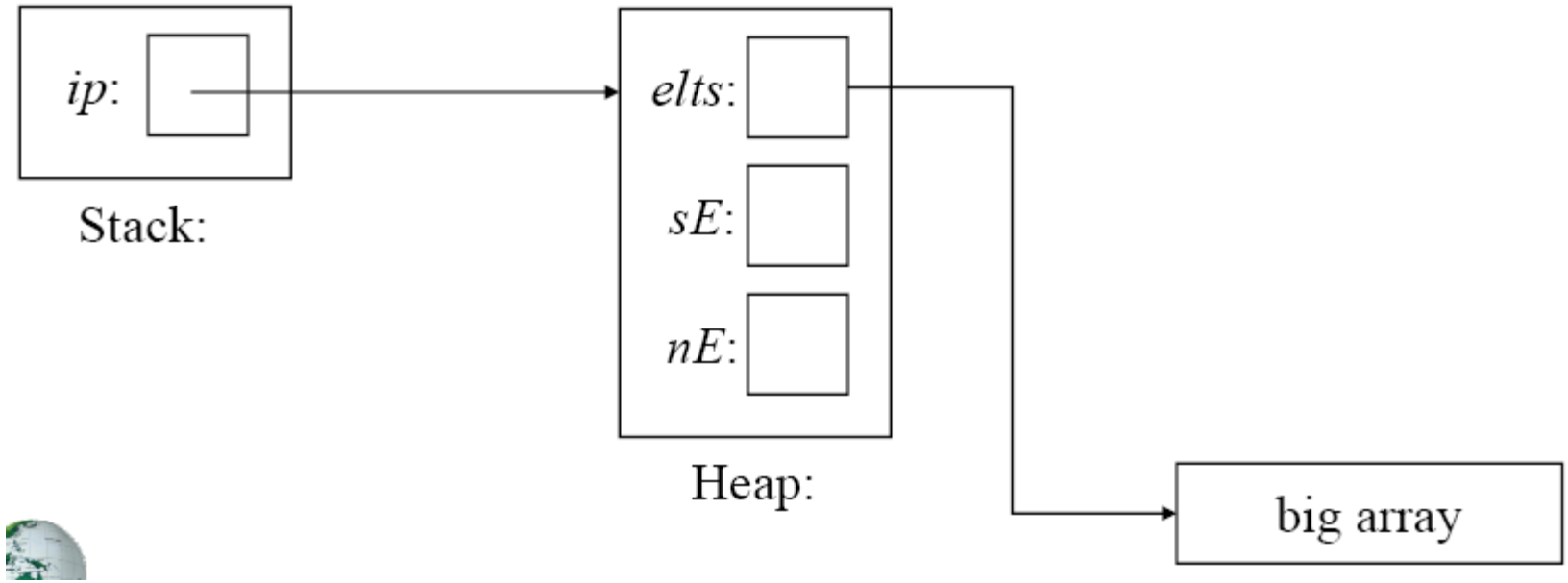
- 新的IntSet定义可以动态地创建和销毁
  - 就像其他类型一样

- 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



带有析构器的IntSet

- 当在带有析构器的类的对象上调用delete时
  - 或者在这种实例的数组上调用delete[]时
  - 析构器会首先被调用
  - 然后这个对象自身被删除

- 回收IntSet(一个指针和两个整数)
  - 调用析构器会回收由50个数组构成的数组

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
```

void*

- void*指的是“指向那些编译器不知道是什么类型内容的指针”
- 我们使用void*，可以在两端相互不知道对方类型的代码片段间传递内存地址——当然程序员自己得知道具体的类型
  - 例: 一个回调函数的参数
- 没有void类型的对象

```
void v;        // 错误
void f();      // f() 什么都不返回– f() 不会返回void类型的对象
```

- 指向任何类型的指针都可以被赋值给void*

```
int* pi = new int;
double* pd = new double[10];
void* pv1 = pi;
void* pv2 = pd;
```

void*

- 使用void*时，我们必须自己告诉编译器它到底指向的是什么

```
void f(void* pv)
{
    void* pv2 = pv;      // 拷贝没有问题 (拷贝就是void*s用途)
    double* pd = pv;     // 错误：不能将void*转化成double*
    *pv = 7;             // 错误：不能对void*解引用
                         // int 7 不能像double那样用7.0表示
    pv[2] = 9;           // 错误：不能对void*使用下标
    pv++;                // 错误：不能自增
    int* pi = static_cast<int*>(pv);        // ok: 显示转换
    // ...
}
```

- static_cast能够被用于显式将指针转化到某个对象类型
  - “static_cast” 是故意为一个危险的操作取的名字——只有当万不得已时才使用它

void*

- void* 是C++中最接近纯机器地址的
  - 一些系统设备需要使用到void*
  - 还记得FLTK回调函数吗?
    - Address 就是void*:

```
typedef void* Address;
  void Lines_window::cb_next(Address,Address)
```

指针和引用

- 可以把引用认为是一个会自动解引用的指针
  - 或者认为是“一个对象的别名”
  - 引用必须被初始化
  - 引用的值在初始化以后就无法改变

```
int x = 7;
int y = 8;
int* p = &x;
*p = 9;
p = &y;        // ok
int& r = x;
x = 10;
r = &y;        // 错误 (其他试图改变r的指向的操作也会出错)
```

Thank You!