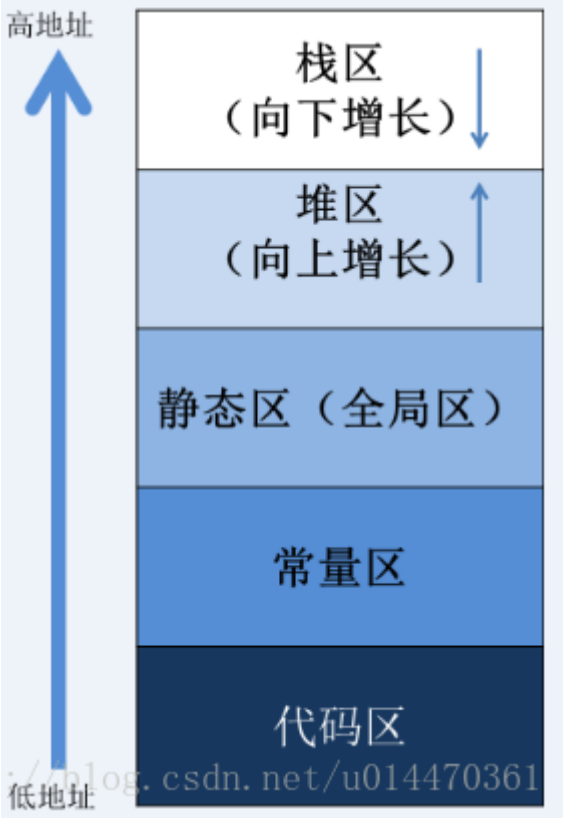
第12章 动态内存

程序内存分类



- 代码区：存放程序的代码，即CPU执行的机器指令，并且是只读的、
- 常量区：存放常量(程序在运行的期间不能够被改变的量，例如: 10，字符串常量”abcde”， 数组的名字等)
- 静态区（全局区）：用来保存局部static对象、类static数据成员以及**任何定义在任何函数之外的变量**
  - 使用前分配，直到程序结束时销毁
- 堆区：由程序员动态分配，必须显式的销毁

> 所谓的动态内存

- 栈区：存放函数内的局部变量，形参和函数返回值
  - 仅在其定义的程序块运行时才存在

> 分配在栈区和静态区的对象由编译器负责创建和销毁回收

动态内存与智能指针

- C++中通过一对运算符完成动态内存的管理
  - `new`：为对象分配空间并返回一个指向该对象的指针
  - `delete`：接受一个动态对象的指针，**销毁该对象，并释放与之关联的内存**

```
int*a = new(int);       // new 接受一个类型，根据该类型决定所需内存大小
*a = 1;
cout<<a;
delete(a);
```

> 在正确的时间释放内存是极其困难的 ---> 引入智能指针(`C++11`)

- 两种智能指针：负责自动释放所指向的对象
  - `shared_ptr`：允许多个指针指向同一个对象；
  - `unique_ptr`："独占"所指向的对象；

> 智能指针也是模版

- 标准库还定义了一个名为`weak_ptr`的伴随类，它是一种弱引用，指向shared_ptr所管理的对象

> - 伴随类 `? ?`
> - 在外部符号引用在目标文件被最终链接成可执行文件时，如果没有找到该符号的定义，链接器就会报符号未定义错误，这种被称为强引用；对于弱引用，如果该符号有定义，则链接器将该符号的引用决议，如果未定义，则链接器也不会报错，编译器不认为它是一个错误。一般对于未定义的弱引用，链接器会默认为0，或者是一个特殊值，以便程序代码被识别。弱引用和弱符号主要用于库的链接

> 上述三种类型都定义在`<memory>`中

share_ptr类

```
shared_ptr<string> p1;       // 定义可以指向string的智能指针
shared_ptr<list<int>> p2;
```

- 默认初始化的智能指针中保存一个空指针

表 12.1：shared_ptr 和 unique_ptr 都支持的操作	
shared_ptr<T> sp unique_ptr<T> up	空智能指针，可以指向类型为 T 的对象
p	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
*p	解引用 p，获得它指向的对象
p->mem	等价于 (*p).mem
p.get()	返回 p 中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
swap(p, q) p.swap(q)	交换 p 和 q 中的指针

```
auto p = make_shared<int>(2);
cout<<p<<endl;              // 0xdc1780
cout<<p.get()<<endl;        // 0xdc1780     p.get()返回一个int*
cout<<*p.get();             // 2
return 0;
```

- p.get()返回一个内置指针，指向智能指针管理的内容
  - 使用get()返回的指针不能使用`delete`此指针
  - 永远不要用get初始化另一个智能指针

> 如果delete该指针，那么智能指针的引用计数为0时，会再次delete，造成二次delete错误

- **shared_ptr独有操作**

```
make_shared<T>(args);           // 返回一个share_ptr<T> 类型，使用 args 构造 给定类型的对象
shared_ptr<T> p(q);             // p是shared_ptr q的拷贝；q中的计数器会递增；q中的指针必须能转换为 T*
p = q;          // p、q所保存的指针必须能互相转换；q中的计数器递增，p中的计数器递减（如果减到0,则将其管理的内存释放）
p.unique();     // 若p.ues_count()=1,返回true
p.use_count();  // 返回与p共享对象的智能指针数量
```

  - **引用计数**：每个shared_ptr都有一个**关联的**计数器

> 个人理解：所谓的计数器，实际上是对其指向的对象的智能指针数的记录

```
r = q;          // 递增q指向的对象的引用计数
                // 递减r原来指向的对象的引用计数。如果该操作导致r原来指向的对象已没有引用者，该对象会自动释放
```

程序使用动态内存的原因：

- 程序不知道自己需要用多少对象 ----> 容器
- 程序不知道所需对象的准确类型（15章）
- 程序需要在多个对象间共享数据 -----> 自定义类`Blog`，它使用动态内存是为了让许多对象能**共享相同的底层数据**

```
Blog<string> b1;
{
    Blog<string> b2 ={"a","b"};
    b1 = b2;        // b1内的元素并不是b2内元素的拷贝，而是指向是同一块内存
}   // b2被销毁了，当b2的元素不给销毁
    // b1 指向最初由b2创建的元素
```

> Blog最终应该是一个模版，但是我们这些先实现一个管理string的Blog

```
class StrBlob{
    public:
        StrBlog(std::initializer list<std::string> il);         // 如何为类设置列表初始化的构造方法
    ......
    private:
        std::shared_ptr<std::vector<std::string> data;
        void checke(syze_type i, const std::string &msg) const;
}
```

  - 用动态内存作成员对象，保证类被销毁后，其中的数据成员不被销毁

直接管理内存： new | delete

- 在自由空间分配的内存是无名的，因此new无法为其分配的对象命名，**只能返回一个指向该对象的指针；**
- 默认情况下，动态分配的对象时默认初始化的
  - 内置类型的对象的值将是未定义的；
  - 类类型对象是所有默认构造函数初始化
- 可以使用圆括号或花括号来初始化；

  可以使用值初始化

```
int *p = new int;           // 未定义
int *p = new int(43);
int *p = new int{1};
int *p = new int();         // 值初始化
-----------------------
auto p1 = new auot(obj);    // c++11
```

- 动态分配的const对象必须要初始化

```
const int *pci = new const int(12);
```

- 分配内存时，如果内存耗尽，会抛出`bad_alloc`异常

> ```
> int *p = new (nothrow) int;
> ```
>
> - 这种形式称为 **定义new**，允许我们向new传递额外的参数
> - `(nothrow)`告诉new不抛出异常
>
> 以上两个，都包含在`<new>`中

- delete释放内存时，释放一块非new分配的内存，或将相同的指针释放多次，都是未定义的行为

> 编译器无法识别这些错误

- `delete p`之后，p变为 **空悬指针**，即指向一块曾经保存数据对象但现在已经无效的内存的指针

> p的值不变，但是*p已经无意义
>
> 可以令`p = nullptr`，来处理

---

shared_ptr和new结合使用

```
shared_ptr<int> p1(new int(4));
```

- 接受指针参数的指针指针构造函数是explicit的，只支持直接初始化[^1]

> 因此，一个返回share_ptr的函数不能在其返回语句中隐式转换一个普通指针。

- 默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存

> 智能指针使用delete来释放所关联的对象

- 如果想要绑定到其他类型的资源的指针上，必须提供自己的操作来替代delete

其他操作

定义或改变shared_ptr的其他方法	含义
shared_ptr p(q)	p管理内置指针q所指向的对象；q必须指向new分配的内存，且能转换为T*类型
shared_ptr p(u)	p从unique_ptr u中那里 **接管** 了对象的所有权； **将u置空**
shared_ptr p(q, d)	p接管内置指针q所指向的对象的所有权；q必须能转化为T*类型 p将使用可调用对象d来代替delete
shared_ptr p(p2, d)	p是p2的拷贝，使用d来代替delete
p.reset() p.reset(q) p.reset(q, d)	若p是唯一指向其对象的share_ptr，reset会释放该对象 若有参数+q，会令p指向q; 若有参数d，将会调用d而不是delete来释放q

> 注意这里措辞：
>
> - 管理...对象：当将一个智能指针绑定到一个普通指针时，就将内存的管理责任交给了这个智能指针了
>
> ```
> void process(shared_ptr<int> ptr)
> {
>     cout<<p<<endl;
> }
> int main(){
>
>     int *x(new int(43));
>     cout<<x<<endl;
>     process(shared_ptr<int>(x));        // 这里创建一个管理x的智能指针，函数调用完后，智能指针自动销毁内存内容
>     cout<<*x;       // 未定义,x是一个空悬指针。
>     return 0;
> }
> ```
>
> **优先使用**make_shared而不是new来初始化share_ptr对象

替没有默认析构函数的类自动释放内存

```
// 假设 类connection 没有默认析构函数
void f(destinaiton &d){
    connection c = connect(&d);
    //
    //最后忘记调用disconnect,就无法关闭c了
}
// 使用智能指针
void f(destinaiton &d){
    connection c = connect(&d);
    shared_ptr<connection> c(&c, end_connection;
    //
    //最后忘记调用disconnect,但是因为引用计数=0，也会删除c
}
```

- 因为，默认情况下share_ptr假定指向的内存是动态内存，使用delete来维护；我们可以自定义一个删除器，代替delete

unique_ptr

某时刻，只能有一个unique_ptr指向一个给定对象

> 不能拷贝或赋值unique_ptr。
>
> - 特殊的拷贝：拷贝或赋值一个将要被销毁的unique_ptr(13.6)
>
> ```
> unique_ptr<int> clone(int p ) {
>     return unique_ptr<int>(new int(p));
> }
> ------------------------------------;
> unique_ptr<int> clone(int p ) {
>    unique_ptr<int> ret(new int(p));
>     ...
>     return ret;
> }
> ```

- unique_ptr独有操作

```
unique_ptr<T> u1;           // 空unique_ptr
unique_ptr<T, D> u2;        // 空uniequ_ptr, D是自定义删除器类型
unique<T, D> u(d) u(d);     // 空unique_ptr, 用类型为D的对象d来代替delete

u = nullptr;        // 释放u指向的对象，将u置空

u.release();        // u放弃对指针的控制权，返回u保存的指针，并将u置空
u.reset();          // 释放u所指向的对象
u.reset(q);         // 令u指向内置指针q的对象，否则将u置空
u.reset(nullptr);
```

- 调用release会切断unique_ptr和它原来管理的对象间的联系；

  其返回的指针通常用来初始化另一个智能指针或给另一个智能指针赋值；

  > 如果我们只是调用release，会丢失原来对象的指针，无法管理·

  ```
  p2.release();       // p2不会释放内存，但是原来的内存区域没有指向它的指针，也就没法管理（删除）
  ```

- unique_ptr管理删除器的方式与share_pte有所不同

  > 16.1节介绍原因

- 重载删除器

```
unique_ptr<connection, decltype(end_connection)*> p(&c, end_connection());
```

weak_ptr

一种不控制所指向对象生存周期的智能指针，它**指向一个share_ptr管理的对象**，但是**不会改变这个对象的引用计数（弱引用）**

> 只能用一个share_ptr来初始化weak_ptr

- weak_ptr操作

```
weak_ptr<T> w;           // 空weak_ptr
weak_ptr<T> w(sp);       // 与share_ptr指向相同对象的weak_ptr;

w = p;                // w与p共享对象

w.reset();            // 将w置空
w.use_count();        // 与w共享对象的shared_ptr的数量
w.expired();          // 如果w.use_conut()为0，返回true
w.lock();             // 如果expired为true，返回一个空share_ptr;否则返回一个指向w对象的share_ptr
```

- 由于对象可能不存在，不能使用weak_ptr直接访问对象，必须调用lock

```
shared_ptr<int> sp = make_shared<int>(2);
weak_ptr<int> wp = sp;
// cout<<*wp;        // 报错
cout<<*wp.lock();    // 不安全的操作，应该先判断是否存在
```

- 用途：
  - 当你想使用对象，但是并不管理对象，并且在需要时可以返回对象的shared_ptr时，则使用
  - 解决shared_ptr的**循环引用**问题和 **空悬指针问题**

> **循环引用没懂**

扩展阅读

- weak_ptr的使用场景
- C++弱引用智能指针weak_ptr的用处

动态数组

> **大多数应用应该使用标准库容器而不是动态分配的数组**
>
> 分配的动态数组必须定义自己版本的操作、在拷贝、复制以及销毁对象时管理所关联的对象

new和动态数组

```
int *p = new int[10];        // 用方括号表示分配10个连续的int内存，进行默认初始化
int *p = new int[10]{1, 2, 3,};      // 初始化列表如果超过内存大小，抛出bad_array_new_length错误
```

- 分配一个数组会得到一个元素类型的指针

  > 动态数组并不是一个数组类型的对象

  - 不能对动态数组调用`begin`或`end`
  - 不能使用范围for来处理动态数组内的元素

    > 因为数组维度其实是未知的，只是一块内存内置而已

- 因为不支持括号初始化动态数组，所以不允许使用auto分配数组

  > **花括号初始化原理？？**

- 允许动态分配一个空数组

```
size_t n = get_size();
int *p = new int[n];
```

  - 如果n=0，new会返回一个合法的非空指针，此指针不能解引用

    > 类似尾后指针

- 释放动态数组

```
delete [] p;        // 逆序销毁元素
```

  > 如果没加方括号，编译器也可能不会报错

智能指针和动态数组

标准库提供一个可以管理动态数组的`unique_ptr`版本。

```
unique_ptr<int[]> up(new int[10]);      // 必须在对象类型后面跟一对空方括号
up.release();       // 自动使用delete[]销毁其指针
```

- 指向数组的unique_ptr不支持成员访问运算符（点和箭头运算符）；

  额外支持下标运算访问

`shared_ptr`不直接支持管理动态数组。如果希望使用，则必须提供自己定义的删除器

> C++17已经支持了，并且支持下标操作。
>
> ```
> 1 shared_ptr<int[]> sp (new int[10]);
> ```

```
1 shared_ptr<int[]> sp (new int[10], [](int*p){delete []p;})
```

> 如果未提供删除器，这段代码是未定义的。
>
> 默认情况下，使用delete销毁指向的对象，对动态数组导致未定义的行为

- 不支持下标运算符

  ```
  1 *sp.get()   // == up[0];
  ```

allocator类

> 销毁对象和释放内存的关系：
>
> 销毁只意味着不可再访问该对象，这块内存中实际上还保存着内容

---

new将内存分配和对象构造组合在一起

- 缺点：
  - 可能分配的内存，某部分在之后没有被使用，那么就会创建一些永远用不到的值；
  - 每次使用到的元素都被赋值两次：一次在默认初始化时；一次在赋值时
  - 没有默认构造函数的类无法动态分配数组

```
class test{
public:
    test(int b):a(b){};
private:
    int a;
};
int main(int argc, char const *argv[])
{
    int *p = new test[10]{1,2,3};    // 错误，因为类没有支持列表初始化
    int *p = new test[10];           // 错误，因为类没有默认初始化函数
    return 0;
}
```

头文件：`<memory>`

特点：将内存分配和对象构造分离开

> allocate分配的内存是未构造的

相关操作

操作	含义
allocate a	定义一个名为a的allocate对象，它可以为类型为T的对象分配内存
a.allocate(n)	分配一段原始的、**未构造**的内存，保存n个类型为T的对象；返回第一个位置的指针。
a.deallocate(p, n)	释放从p开始的的n个对象的内存区域 **n必须对应之前分配的内存大小。** **调用该函数之前，必须调用destory**
a.construct(p, args)	arg被传递给类型为T的构造函数，用来在p指向的内存中构造**一个**对象
a.destroy(p)	对P指向的对象执行析构函数

- destory需要对内存内每个元素调用

```
while(q != q) {
    alloc.destory(--q);
}
```

表 12.8：allocator 算法	
这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。	
`uninitialized_copy(b,e,b2)`	从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 指向的内存必须足够大，能容纳输入序列中元素的拷贝
`uninitialized_copy_n(b,n,b2)`	从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
`uninitialized_fill(b,e,t)`	在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝
`uninitialized_fill_n(b,n,t)`	从迭代器 b 指向的内存地址开始创建 n 个对象。b 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

使用标准库：文本查询程序

```
1  using line_no = vector<string>::size_type;      //重命名，方便理解
2
3  class QuerryResutl{
4      // friend
5      friend ostream &print(ostream & os, const QuerryResutl & qr);
6  public:
7      QuerryResutl(string s, shared_ptr<set<line_no>> p,
   shared_ptr<vector<string>> f):sought(s),lines(p),file(f){}
8  private:
9      string sought;      // 保存查询的单词
10     shared_ptr<set<line_no>> lines;
11     shared_ptr<vector<string>> file;        // 输入文件
12 };
13
14 class TextQuerry{
15 public:
16     TextQuerry(ifstream&);
17
18     QuerryResutl query(const string&) const;
19
20 private:
21     shared_ptr<vector<string>> file;
22     map<string,shared_ptr<set<line_no>>> wm;
23 };
24 // 从文件中按行读取，并保存单词
```

```cpp
TextQuerry::TextQuerry(ifstream & in):file(make_shared<vector<string>>()) {                // 注意这里和初始化列表的联合使用
    string text;
    while(getline(in,text)) {
        file->push_back(text);
        int n = file->size() - 1;       // 当前行号
        // 获取每句话的单词
        istringstream line(text);
        string word;
        while (line>>word) {
            auto &line = wm[word];
            if (!line) {
                line.reset(new set<line_no>);
            }
            line->insert(n);        // line中保存了所有含该单词的行号
        }
    }
}
// 查询指定单词
QuerryResutl TextQuerry::query(const string &sought) const {
    static shared_ptr<set<line_no>> nodata(make_shared<set<line_no>>());
    auto loc = wm.find(sought);
    if (loc == wm.end()) {
        return QuerryResutl(sought, loc->second, file);
    } else {
        return QuerryResutl(sought, loc->second, file);
    }
}
string make_plural(size_t ctr, const string &word, const string& ending) {
    return (ctr > 1)?word+ending:word;
}
ostream &print(ostream & os, const QuerryResutl & qr)
{
    os<<qr.sought<<" occurs "<<qr.lines->size()<<" "<<make_plural(qr.lines->size(), "time", "s")<<endl;
    for (auto num : *qr.lines) {
        os <<"\t (line"<<num+1<<") "<<*(qr.file->begin()+num)<<endl;
    }
```

```
61      return os;
62  }
63  void runQuerise(ifstream & in) {
64      TextQuerry tp(in);
65      while (1) {
66          cout<<"输入要查询的单词";
67          string s;
68          if(!(cin>>s) || s == "q")    break;
69          print(cout, tp.query(s))<<endl;
70      }
71  }
72  int main() {
73      ifstream in("../text.txt");
74      if(in.is_open()) {
75          runQuerise(in);
76      }else
77          cout << "Hello, World!" << std::endl;
78      return 0;
79  }
80
```

- 使用面向对象的思想，多使用类
- 在类中共享数据
  - QueryResult类要表达查询的结果（包括给定单词的行号的set和对应的文件），而这些数据都保存在TextQuery中
  - 优先通过指针，传递该内容，避免复制操作
  - 但是考虑，T如果TextQuery对象在QueryResult之前销毁，那么QuerryResult中将引用一个不在存在的对象;

    即，两个类中的对象周期应该是同步的，所以使用智能指针，共享同一个内存，同时自动管理
  -