第12章 动态内存

1. **静态内存** 用来保存局部 `static` 对象、类 `static` 数局成员以及定义在任何函数之外的变量。
2. **栈内存** 用来保存定义在函数内的非 `static` 对象。
3. 分配在 **静态** 或 **栈** 内存中的对象由编译器自动创建和销毁。
4. 对于 **栈对象**，仅在其定义的程序块运行时才存在。
5. `static` 对象在使用之前分配，在程序结束时销毁。
6. 程序用 **堆（heap）** 来存储 **动态分配（dynamically allocate）** 的对象（程序运行时分配的对象）。动态对象的生存期由程序来控制，当动态对象不再使用时，我们的代码必须显式地销毁它们。
7. 虽然使用动态内存有时是必要的，但是正确地管理动态内存是非常棘手的。

12.1 动态内存与智能指针

1. `new` 在动态内存中为对象分配空间并返回一个指向该对象的指针，我们可以选择对对象进行初始化。
2. `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。
3. 忘记释放内存，会产生内存泄漏。
4. 在尚有指针引用内存的情况下释放了内存，会产生引用非法内存的指针。
5. `shared_ptr` 允许多个指针指向同一个对象。
6. `unique_ptr` 则 “独占” 所指向的对象。
7. `weak_ptr` 是一个伴随类，一种弱引用，指向 `shared_ptr` 所管理的对象。

12.1.1 `shared_ptr` 类

1. 默认初始化的智能指针中保存着一个空指针。
2. `p = q`： p 和 q 都是 `shared_ptr`，此操作会递减 p 指向对象的引用计数，递增 q 原来指向对象的引用计数。若 p 的引用计数变为 0，则将其管理的原内存释放。
3. `p.use_count()` 返回与 p 共享对象的智能指针数量，可能很慢，主要用于调试。
4. 到底是用一个计数器还是其他数据结构来记录有多少指针共享对象，完全由标准库的具体实现来决定。关键是智能指针能记录有多少个 `shared_ptr` 指向相同的对象，并能在恰当的时候自动释放对象。
5. `shared_ptr` 在无用之后仍然保留的一种可能情况是：将 `shared_ptr` 存放在一个容器中，而后不再需要全部元素，而只使用其中一部分。（在此情况下，要记得用 `erase` 删除不再需要的那些元

素。）

6. 程序使用动态内存出于以下三种原因之一：
   （1）程序不知道自己需要使用多少对象
   （2）程序不知道所需对象的准确类型
   （3）程序需要在多个对象间共享数据
7. 下面通过一个例子来详细说明一下程序需要在多个对象间共享数据的情况：

```
// 数据不共享的情况：
vector<string> v1;    // 空 vector

{   // 新作用域
    vector<string> v2 = { "a", "an", "the" };
    v1 = v2;    // 从 v2 拷贝元素到 v1 中
}   // v2 被销毁，其中的元素也被销毁
    // v1 有三个元素，是原来 v2 中元素的拷贝


// 数据共享的情况：
// 假定我们希望定义一个名为 StrBlob 的类，保存一组元素
// 与容器不同，我们希望 StrBlob 对象的不同拷贝之间共享相同的元素
StrBlob b1;    // 空 StrBlob

{   // 新作用域
    StrBlob b2 = { "a", "an", "the" };
    b1 = b2;    // b2 和 b1 共享相同的元素
}   // b2 被销毁了，但是 b2 中的元素不能销毁
    // b1 指向最初的由 b2 创建的元素
```

8. 接下来我们定义 StrBlob 类（使用 vector 来保存元素，并将 vector 保存在动态内存中）：

```
// StrBlob.h
class StrBlob{
public:
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);

private:
    std::shared_ptr<str::vector<string>> data;
}

// StrBlob.cpp
StrBlob::StrBlob() : data(make_shared<vector<string>>()) {}
StrBlob::StrBlob(initializer_list<string> il) :
                data(make_shared<vector<string>>(il)) {}
```

9. 当我们拷贝、赋值或销毁一个 StrBlob 对象时，它的 Shared_ptr 成员也会被拷贝、赋值或销毁。

10. 拷贝一个 `shared_ptr` 会递增其引用计数；将一个 `shared_ptr` 赋予另一个 `shared_ptr` 会递增赋值号右侧的 `shared_ptr` 的引用计数，而递减左侧 `shared_ptr` 的引用计数。
11. 如果一个 `shared_ptr` 的引用计数变为 0 ，它所指向的对象会被自动销毁。因此，对于由 `StrBlob` 构造函数分配的 `vector` ，当最后一个指向它的 `StrBlob` 对象被销毁时，它会随之被自动销毁。

12.1.2 直接管理内存

1. C++ 定义了两个运算符来分配和释放动态内存：运算符 `new` 分配内存， `delete` 释放 `new` 分配的内存。
2. 相对于智能指针，使用这两个运算符管理内存非常容易出错，在学习第13章前，除非使用智能指针来管理内存，否则不要分配动态内存。
3. 在自由空间（堆）分配的内存是 **无名** 的，因此 `new` 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int* pi = new int;          // pi指向一个动态分配的、未初始化的无名对象
```

4. 默认情况下，动态分配的对象是 **默认初始化** 的，这意味着 **内置类型** 或 **组合类型的对象** 的值将是 **未定义** 的，而 **类类型对象** 将用 **默认构造函数** 进行初始化。
5. 也可以使用 **直接初始化**（**传统的构造方式（使用圆括号）**、**列表初始化**） 和 **值初始化(使用空括号)** 来动态分配对象。

```
// 默认初始化
string* ps1 = new string;           // 默认初始化为空string
int* pi1 = new int;                 // 默认初始化，*pi1的值未定义

// 传统构造方式
int *pi2 = new int(1024);           // pi2指向的对象值为1024
string* ps2 = new string(10, '9');  // *ps2为“9999999999”
vector<int>* pv = new vector<int>{0, 1, 2, 3};  // vector有4个元素，值依次从0到3

// 值初始化
string* ps3 = new string();         // 值初始化为空string
int* pi3 = new int();               // 值初始化为0
```

6. 出于于变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。
7. 对于一个定义了默认构造函数的类类型，其 `const` 动态对象可以隐式初始化，而其他类型的对象就必须显示初始化。

```
const string* pcs = new const string;   // 隐式初始化
const int* pci = new const int(1024);   // 显示初始化
```

8. 可以使用 **定位 new (placement new)** 来阻止动态对象分配时因内存耗尽而抛出的异常。

```
int* p1 = new int;              // 如果分配失败，会抛出 std::bad_alloc

// 向new传递一个标准库定义的nothrow对象
int* p2 = new (nothrow) int;    // 如果分配失败，返回一个空指针
```

9. `delete` 表达式执行两个动作：销毁给定的指针指向的对象；释放对应的内存。
10. 传递给 `delete` 的指针必须指向 **动态分配的内存**，或者 **空指针**。释放一块并非 `new` 分配的内存，或者将相同的指针值释放多次，其行为是未定义的。
11. 通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。编译器也不能分辨一个指针所指向的内存是否以及被释放了。
12. 由内置指针（而不是智能指针）管理的动态内存在被显示释放前一直都会存在。
13. 使用 `new` 和 `delete` 管理动态内存有三个常见的问题：
    （1）忘记 `delete` 内存；
    （2）使用已经释放掉的对象；
    （3）同一块内存被释放两次。
14. 坚持只使用 **智能指针**，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。
15. 在 `delete` 之后，指针就变成了 **空悬指针（dangling pointer）**（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），在 `delete` 之后将 `nullptr` 赋予指针，可以清楚的指出指针不指向任何对象。
16. 当有多个指针指向相同的内存时，在 `delete` 内存之后重置指针的办法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有用的。

```
int* p(new int(1024));  // p指向动态内存
auto q = p;             // p和q指向同一块内存
delete p;               // p和q均变为无效
p = nullptr;            // 此时p不再绑定到任何对象
// 此时重置p对q没有任何作用，q仍然指向原来那块（已经被释放的）内存地址
// 在实际系统中，查找指向相同内存的所有指针是异常困难的！
```

12.1.3 `shared_ptr` 和 `new` 结合使用

1. 接受指针参数的智能指针构造函数是 `explicit` 的。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用 **直接初始化形式** 来初始化一个智能指针。：

```
shared_ptr<int> p1 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024));  // 正确：使用了直接初始化形式
```

2. 一个返回 `shared_ptr` 的函数不能在其返回语句中隐式转换一个普通指针：

```
shared_ptr<int> clone(int p){
    return new int(p);                  // 错误
    return shared_ptr<int>(new int(p)); // 正确
}
```

3. 一般情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因此智能指针默认使用 `delete` 释放它所关联的对象。 但是也可以将智能指针绑定到一个指向其他类型的资源的指针上，只是此时需要提供自己的释放操作来替代 `delete` 。

```
// 如果p是唯一指向其对象的shared_ptr，reset会释放此对象。
p.reset();

// 若传递了可选的参数内置指针q，会令p指向q，否则会将p置为空。
p.reset(q);

//若还传递了参数d，将会调用d而不是delete来释放q
p.reset(q, d);
```

4. 不要混合使用普通指针和智能指针，使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁：

```
// ptr是传值方式传递，因此拷贝时会递增其引用次数。
void process(shared_ptr<int> ptr){
    // 使用ptr
}   // ptr离开作用域，ptr会被销毁。由于ptr销毁了，引用次数会递减

// process正确的使用方法：
shared_ptr<int> p(new int(42)); // 此时引用计数为1
process(p);     // 拷贝p会递增其引用次数，此时在process中引用计数值为2
int i = *p;     // 离开了process作用域，引用计数值为1

// process错误的使用方法：
int *x(new int(1024));  // 危险：x是普通指针，不是智能指针
process(x);             // 错误：不能将int* 转换为shared_ptr<int>
process(shared_ptr<int>(x)); // 合法，但是由于传参时使用的是一个临时变量，
                             // 临时变量在传递完之后会被销毁，所以传完参
                             // 之后的process里的ptr引用计数为1
int j = *x; // 未定义：离开了process作用域，引用计数值为0，此时x为一个悬空指针
```

5. `get` 用来将指针的访问权限传递给代码，只有在确定代码不会 `delete` 指针的情况下，才能使用 `get` 。特别是，永远不要用 `get` 初始化另一个智能指针或者为另一个智能指针赋值。

```
shared_ptr<int> p(new int(42)); // 引用计数为1
// 新程序块
{
    // 两个独立的shared_ptr指向相同的内存
    shared_ptr<int> q(p.get());    // 错误：将get用来初始化指针指针
}   // 程序块结束，q被销毁，它指向的内存被释放

int foo = *p;    // 未定义：p指向的内存已经被释放了
```

12.1.4 智能指针和异常

1. 那些分配了资源，而又 **没有定义析构函数** 来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员非常容易忘记释放资源。与管理动态内存类似，我们通常可以使用智能指针来管理不具有良好定义的析构函数的类：

```
class destination;
class connection;    // 假定connection没有析构函数

connection connect(destination* d) {}
void disconnect(connection c) {}

void f(destination& d)
{
    connection c = connect(&d);
    // ...
    // 如果我们在f退出前忘记调用disconnect，就无法关闭c了
}

// 使用智能指针管理对象的生命周期
// 定义删除器
void end_connection(connection* p) { disconnect(*p); }
void f(destination& d)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // ...
    // 当f退出时（即便是由于异常而退出），connection会被正确关闭
}
```

2. 智能指针陷阱：
   （1）不使用相同的内置指针值初始化（或 `reset` ）多个智能指针。
   （2）不 `delete` `get()` 返回的指针。
   （3）不使用 `get()` 初始化或 `reset` 另一个智能指针。
   （4）如果你使用 `get()` 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无

效了。

（5）如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器。

12.1.5 unique_ptr

1. 一个 unique_ptr “拥有”它所指向的对象。某个时刻只能有一个 unique_ptr 指向一个给定对象，当 unique_ptr 被销毁时，它所指向的对象也被销毁。
2. 定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。初始化 unique_ptr 必须采用直接初始化形式。
3. unique_ptr 不支持普通的拷贝或赋值操作。
4. 不能拷贝 unique_ptr 的规则有一个例外：我们可以拷贝或赋值一个将要被销毁的 unique_ptr 。常见的例子是从函数返回一个 unique_ptr ：

```
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;
}
```

5. 向 unique_ptr 传递删除器：

```
// 将之前的例子中的shared_ptr换成unique_ptr
void f(destination& d)
{
    connection c = connect(&d);

    // 使用decltype来指明函数指针类型
    // 必须加一个*来指出我们正在使用该类型的指针
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
}
```

12.1.6 weak_ptr

1. weak_ptr 是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象。
2. 将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。即使有 weak_ptr 指向对象，对象也还是会被释放。因此， weak_ptr 的名字抓住了这种智能指针“弱”共享对象的特点。

3. 由于对象可能不存在，我们不能使用 `weak_ptr` 直接访问对象，而必须调用 `lock` 。检查对象是否存在，如果存在， `lock` 返回一个指向共享对象的 `shared_ptr` ，否则返回一个空 `shared_ptr` ：

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);     // wp弱共享p；p的引用计数未改变

// 如果np不为空则条件成立
if(shared_ptr<int> np = wp.lock()) {
    // 在if中，np与p共享对象
}
```

12.2 动态数组

1. 大多数应用应该使用 **标准库容器** 而不是 **动态分配的数组**。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。
2. 使用容器的类可以使用默认版本的拷贝、赋值和析构操作。分配动态数组的类则必须定义自己版本的操作，在拷贝、赋值以及销毁对象时管理所关联的内存。

12.2.1 `new` 和数组

1. 为了让 `new` 分配一个对象数组，我们要在类型名之后跟一对方括号，指明要分配的对象的数目。

```
int* p1 = new int[42];  // p1指向第一个int

//也可以使用一个表达数组类型的类型别名来分配一个数组
typedef int arrT[42];   // arrT表示42个int的数组类型
int* p = new arrT;      // 分配一个42个int的数组，p指向第一个int
```

2. 通常称 `new T[]` 分配的内存为 “动态数组”，但是这种叫法有些误导。实际上我们得到的是一个指向第一个元素的数组元素类型的指针。
3. 不能对动态数组调用 `begin` 、 `end` 和范围 `for` 循环操作。
4. 可以对数组中的元素进行值初始化，方法是加（）或{}列表。

```
int* pia = new int[3];              // 3个未初始化的int
int* pia2 = new int[3]();           // 3个值初始化为0的int
int* pia3 = new int[3]{0, 1, 2};    // 跟列表相同的int
int* pia4 = new int[3]{0, 1, 2, 3}; // 列表数目超出，抛出bad_array_new_length异常

// 虽然可以用空括号对数组中元素的进行值初始化
// 但是不能在括号中给出初始化器
// 这意味着不能用auto分配数组
int obj = 1;
auto p = new auto[3](obj);
```

5. `int* p = new int[0];`：当用 `new` 分配一个大小为0的数组时， `new` 返回一个合法的非空指针。此指针保证与 `new` 返回的其他任何指针都不相同。我们可以像使用尾后迭代器一样使此指针。但是此指针 **不能解引用**。
6. `delete [] p;` 释放一个指向数组的指针时，方括号是必须的。
7. 在 `delete` 一个数组时忘记了方括号，或者在 `delete` 一个单一对象的指针时使用了方括号，编译器很可能 **不会给出警告**。我们的程序可能在执行过程中在没有任何警告的情况下行为异常（行为是未定义的）。
8. 标准库提供一个可以管理 `new` 分配到数组的 `unique_ptr` 版本，但是不能使用点和箭头成员运算符操纵。

```
unique_ptr<int[]> up(new int[10]);

for(size_t i = 0; i != 10; ++i)
    up[i] = i;  // 可以使用下标运算符来访问数组中的元素。

up.release();   // 自动用delete[]销毁其指针
```

9. `shared_ptr` 不直接支持管理动态数组。如果希望使用，必须提供自己定义的删除器。否则将会使用默认的 `delete` 而不是 `delete[]` 销毁对象。

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

// shared_ptr 未定义下标运算符，并且不支持指针的算数运算
for(size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i;    // 必须用get获取内置指针才能访问元素

sp.reset(); // shared_ptr没有release()
```

12.2.2 `allocator` 类

1. 当分配单个对象时，通常希望将内存分配和对象初始化组合在一起（对应操作是 `new` 和 `delete`）。当分配一大块内存时，我们通常计划在一块内存上按需构造对象，此时我们希望内存

分配和对象构造分离

2. 标准库 `allocator` 类帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方式，它分配的内存是原始的、未构造的。

```
int n = 3;
allocator<string> alloc;       // 可以分配string的allocator对象
auto const p = alloc.allocate(n);// 分配n个未初始化的string

auto q = p;                    // q指向最后构造的元素之后的位置
alloc.construct(q++);          // *q为空字符串
alloc.construct(q++, 10, 'c'); // *q为ccccccccccc
alloc.construct(q++, "hi");    // *q为hi!

// 在还未构造对象的情况下使用原视内存是错误的:
cout << *p << endl;            // 正确：使用string的输出运算符
cout << *q << endl;            // 错误：q指向未构造的内存!

// 用完对象后，必须销毁每个构造的元素
while(q != p)
    alloc.destroy(--q);        // 释放我们真正构造的string

// 释放内存
alloc.deallocate(p, n);
```

3. 标准库还为 `allocator` 类定义了两个伴随算法，可以在未初始化内存中创建对象：

```
vector<int> vi{0, 1, 2, 3};

// 分配一块比vector中元素所占用空间大一倍的动态内存，
allocator<string> alloc;
auto p = alloc.allocate(vi.size() * 2);

// 然后将原vector中的元素拷贝到前一半空间，
// copy有两种写法:
// q 指向最后一个构造的元素之后的位置。
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
auto q = uninitialized_copy_n(vi.begin(), vi.size(), p);

// 对后一半的空间用一个给定值进行填充，
// fill也有两种写法:
uninitialized_fill(q, q + n, 42);
uninitialized_fill_n(q, vi.size(), 42);
```