**内存**

// 静态内存：保存 static数据 和 全局变量 等

// 栈内存：保存局部变量

// 堆内存（动态内存）：程序运行时分配的对象

// c++ 动态内存管理使用 new 分配，delete 删除

// c++ 提供了两种智能指针（行为像指针的类），可以安全的管理动态分配的内存，即使出现异常智能指针也能安全的释放内存

**智能指针**

// shared\_ptr，允许其他shared\_ptr指针贡献一个内存，当所有的shared\_ptr被销毁，内存也会被销毁

// shared\_ptr 维护一个引用内存的 计数器 ，为了在多个shared\_ptr贡献这个计数器，同样将这个计数器放在动态内存中

// 当一个shared\_ptr被销毁或被赋予其他内存的引用（即不在引用该内存），则计数器减1，当计数器为0是，会使用delete销毁内存

shared\_ptr<vector<string>> sp1;

if(sp1){

cout << "has object" << endl; // 有指向对象

}

sp1.get(); // 返回 sp1 保存的指向动态内存（保存着vector<string>）的指针

// sp2 从 sp1拷贝

shared\_ptr<vector<string>> sp2(sp1);

shared\_ptr<vector<string>> sp3;

sp1.swap(sp3); // 交换 sp1 和 sp3 的指针

sp2 = sp3; // sp2 指向 sp3 的内存

sp1.use\_count(); // 返回当前共享指针的数量

sp1.unique(); // 当use\_count为1时，返回true

// 更安全的指针生成方式，传入参数为泛型参数指定的类型

auto sp4 = make\_shared<vector<string>>(vector<string>{ "a", "b" });

for(auto s : \*sp4){

cout << s << endl;

}

{

// 离开此作用域，sp5会被 delete，sp5的析构函数被调用，由于共享内存只有sp5使用，所以sp5在析构函数中 delete共享内存

auto sp5 = make\_shared<vector<string>>(vector<string>{ "a", "b" });

}

// 通过new生成指针

auto sp6 = make\_shared<string>(string("aaa"));

auto sp7 = make\_shared<string>();

sp6.reset(new string("aa")); // 将sp6指向一个新对象

// unique\_ptr指针，只允许有一个指针指向当前内存，unique\_ptr不支持拷贝操作（函数返回值的拷贝是一个特殊的拷贝）

unique\_ptr<vector<string>> up1;

auto p1 = up1.release(); // 释放对内存的管理，返回指向该内存的指针

unique\_ptr<vector<string>> up2(p1); // up1释放了对内存的管理，那我们必须自己管理这个内存，这里使用up2接管内存

// 指针转换

std::shared\_ptr<DFFile> file;

std::shared\_ptr<Directories> dir = std::static\_pointer\_cast<Directories>(file);

**自定义释放操作**

void customizeOpt(string \*sp){

delete sp;

}

void customizeOptTest(){

// 默认释放操作时 delete 对象指针，但我们可以自定义释放操作

shared\_ptr<string> sp1(new string("aaa"), customizeOpt);

unique\_ptr<string, decltype(customizeOpt)\*> up1(new string("aaa"), customizeOpt);

}

**自己管理内存**

// 直接管理内存，一般我们不希望直接管理

int \*np1 = new int; // 分配一个int的内存，但未初始化

delete np1; // 不在使用np1后要删除

int \*np2 = new int(0); // 分配一个int的内存，初始化未0

delete np2;

auto \*np3 = new auto(string("aaa")); // np3的类型是 string\*

delete np3;

// 如果内存耗尽，返回空指针

auto \*np4 = new string("aaa");

if(np4 == nullptr){

}

delete np4;

const string \*np5 = new const string("a"); // 常量指针

delete np5;

np5 = nullptr; // 推荐在delete后将指针置为空

int \*arrp1 = new int[10]; // 分配数组

delete [] arrp1; // 删除数组

**弱智能指针**

// weak\_ptr，弱智能指针，保存share\_ptr指向内存的指针和计数器，但不会增加share\_ptr的计数器

// 当weak\_ptr检测到计数器为0时，将失效

auto sp1 = make\_shared<string>(string("aaaaaa"));

weak\_ptr<string> wp1(sp1);

wp1.use\_count(); // share\_ptr的数量

wp1.expired(); // 指向的内存是否失效

sp1.reset(new string("aaaaaa")); // 将共享指针指向新的内存，此时旧内存的计数器将变为0

auto p = wp1.lock(); // 如果未失效，返回 share\_ptr ，否则返回空的 share\_ptr

if(!p){

cout << wp1.expired() << endl; // 返回true

cout << wp1.use\_count() << endl; // 返回0

}

wp1.reset(); // 置空物品wp1

**allocator内存分配与构造分离**

// 当我们希望将分配和对象构造分离时，可以使用allocator类

allocator<string> alloc1; // 空的内存

auto start = alloc1.allocate(10); // 分配10个string大小的内存，返回p指向内存的指针

auto end = start;

alloc1.construct(end, "a"); // 在第0个分配a

end++;

alloc1.construct(end, "ab"); // 在第1个分配b;

end++;

auto start1 = alloc1.allocate(20); // 分配新的内存

uninitialized\_copy(start, end, start1); // 将旧内存的对象拷贝到新内存，注：不要拷贝未构造的内存

uninitialized\_copy\_n(start, 2, start1); // 将旧内存的对象拷贝到新内存

uninitialized\_fill(start1, start1 + 20, "a"); // 填充对象

while (end != start)

{

end--;

alloc1.destroy(end); // 销毁end指向的对象，一旦元素被销毁，就可以重新使用这块内存

}

alloc1.deallocate(start, 10); // 释放分配的内存

**使用头文件**

#include <memory>