# 01\_走入智能指针

## 前言

在实际的C++开发中，我们经常会遇到诸如程序运行中突然崩溃、程序运行所用内存越来越多最终不得不重启等问题，这些问题往往都是内存资源管理不当造成的。比如：

1. 有些内存资源已经被释放，但指向它的指针并没有改变指向（成为了野指针），并且后续还在使用；
2. 有些内存资源已经被释放，后期又试图再释放一次（重复释放同一块内存会导致程序运行崩溃）；
3. 没有及时释放不再使用的内存资源，造成内存泄漏，程序占用的内存资源越来越多。

智能指针是用于存储动态分配对象指针的类，能自动释放对象，避免堆内存泄漏。它将动态分配资源交由类对象管理，类对象生命周期结束时，析构函数自动释放资源。所以智能指针管理内存的原理就是在函数结束时自动释放内存空间不需要手动释放内存空间，从而帮助彻底消除内存泄漏和悬空指针的问题。

C++里面的四个智能指针：

1. auto\_ptr（C++11已弃用）
2. unique\_ptr
3. shared\_ptr,
4. weak\_ptr

## 智能指针历史历程

C++ 98 中产生了第一个智能指针auto\_ptr。

C++boost给出了更加实用的scoped\_ptr（防止拷贝） 和 shared\_ptr（引进引用计数） 和 weak\_ptr。

C++ 11 引入了unquie\_ptr 和 shared\_ptr 和 weak\_ptr .需要注意的是，unique\_ptr对应的是boost中的scoped\_ptr。并且这些智能指针的实现是参照boost中的实现的。

## RALL机制

RAII（Resource Acquisition Is Initialization）是一种利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接、互斥量等等）的简单技术。

他的思想是资源分配即初始化，定义一个类来封装资源的的分配和释放，在构造函数完成资源的分配和初始化，在析构函数完成资源的清理，可以保证资源的正确释放和初始化

RAII要求，资源的有效期与持有资源的对象的⽣命期严格绑定，即由对象的构造函数完成资源的分配(获取)，同时由析构函数完成资源的释放。在这种要求下，只要对象能正确地析构，就不会出现资源泄露问题。

基于RALL思想提出了智能指针，智能指针就是基于RAII的要求使用模板类，自动化地管理动态资源的释放（不管理资源的创建），智能指针看上去是指针，实际上是赋予了定义的对象。智能指针(smart pointer)是存储指向动态分配（堆）对象指针的类，用于生存期控制，能够确保自动正确的销毁动态分配的资源，防止内存泄露。

## 智能指针的使用场景

在我们需要动态申请内存时，难免最后会有忘记[释放内存](https://so.csdn.net/so/search?q=%E9%87%8A%E6%94%BE%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/2303_79004636/article/details/_blank)的时候，这就导致了内存泄漏。在使用到异常时，某个函数抛出异常后，很可能前面申请的空间也未释放，因此也导致内存泄漏。  
例如：

场景1：普通情况下申请空间后忘记****释放****

void test()

{

int \*p = new int(10);

double \*pp = new double(1.1);

}

int main()

{

test();

return 0;

}

场景2：抛出异常后，申请的空间无法释放

下面程序中我们可以看到，new了以后，我们也delete了，但是因为抛异常导，后面的delete没有得到执行，所以就内存泄漏了，所以我们需要new以后捕获异常，捕获到异常后delete内存，再把异常抛出，但是因为new本身也可能抛异常，连续的两个new和下面的Divide都可能会抛异常，让我们处理起来很麻烦。智能指针放到这样的场景里面就让问题简单多了

int Div(int a, int b)

{

if (b == 0)

{

throw "除0错误";

}

else

{

return a / b;

}

}

void Func()

{

int \*p1 = new int[5];

int \*p2 = new int[5];

// 这里Div函数会抛异常，main函数会捕获异常，delete[]没有执行，引发内存泄漏

int ret = Div(5, 0);

delete[] p1;

delete[] p2;

}

int main()

{

try

{

Func();

}

catch (std::exception &e)

{

std::cout << e.what() << std::endl;

}

catch (...)

{

std::cout << "未知错误" << std::endl;

}

return 0;

}

## 总结

以上就是今天要讲的内容，后续会有更多内容。

## 参考资料

版权声明：本文参考了其他资料和CSDN博主的文章，遵循CC 4.0 BY-SA版权协议，现附上原文出处链接及本声明。

1. [https://blog.csdn.net/qq\_65285898/article/details/131665929](https://blog.csdn.net/qq_65285898/article/details/131665929" \t "https://blog.csdn.net/qq_65285898/article/details/_blank)
2. https://blog.csdn.net/weixin\_75172965/article/details/138239359
3. https://blog.csdn.net/2303\_79004636/article/details/143661852