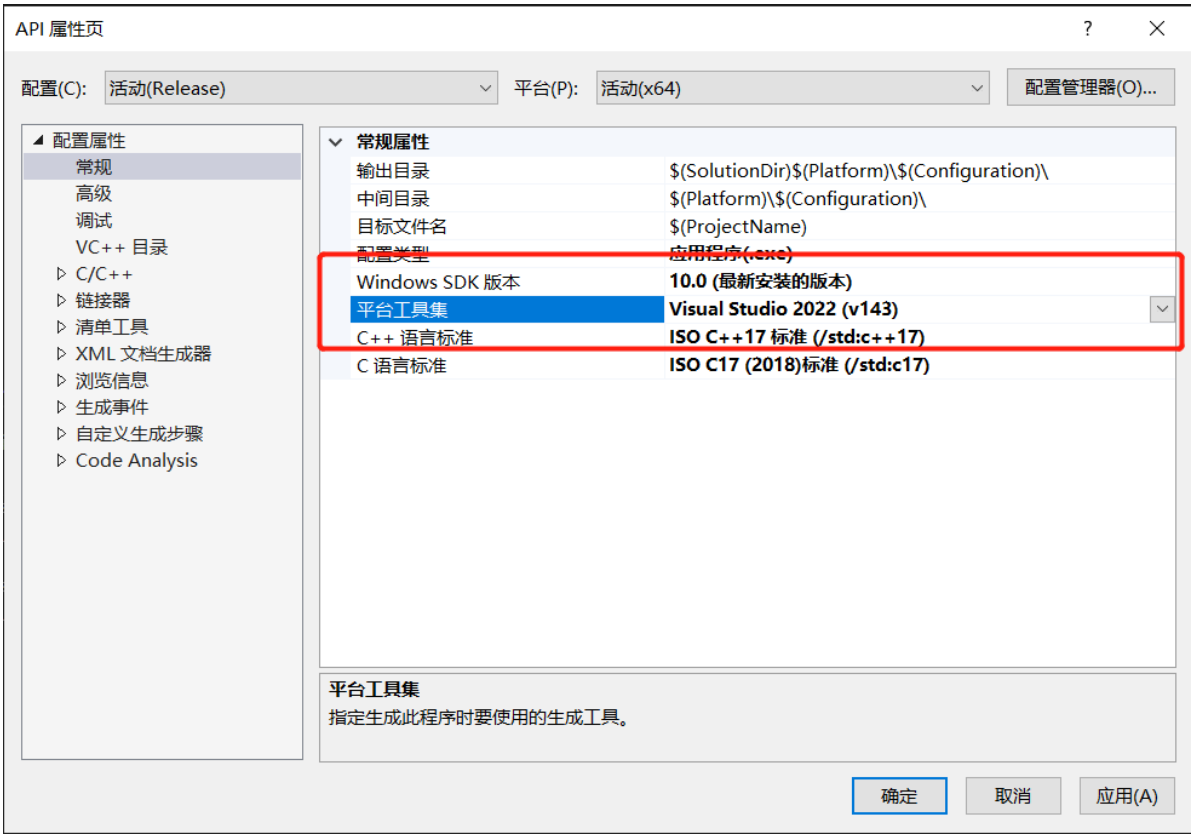
工具使用合集

Visual Studio 2019/2022使用

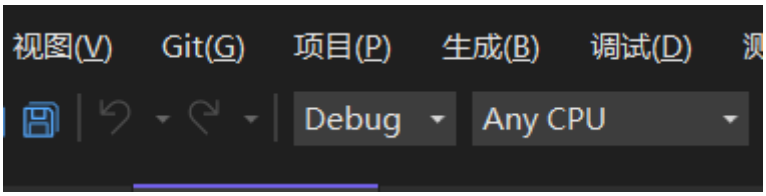
修改工具集

使用vs2022的选手，打开时弹出界面点**取消**，请注意更改默认工具集为vs2022（如图）



生成模式的设置

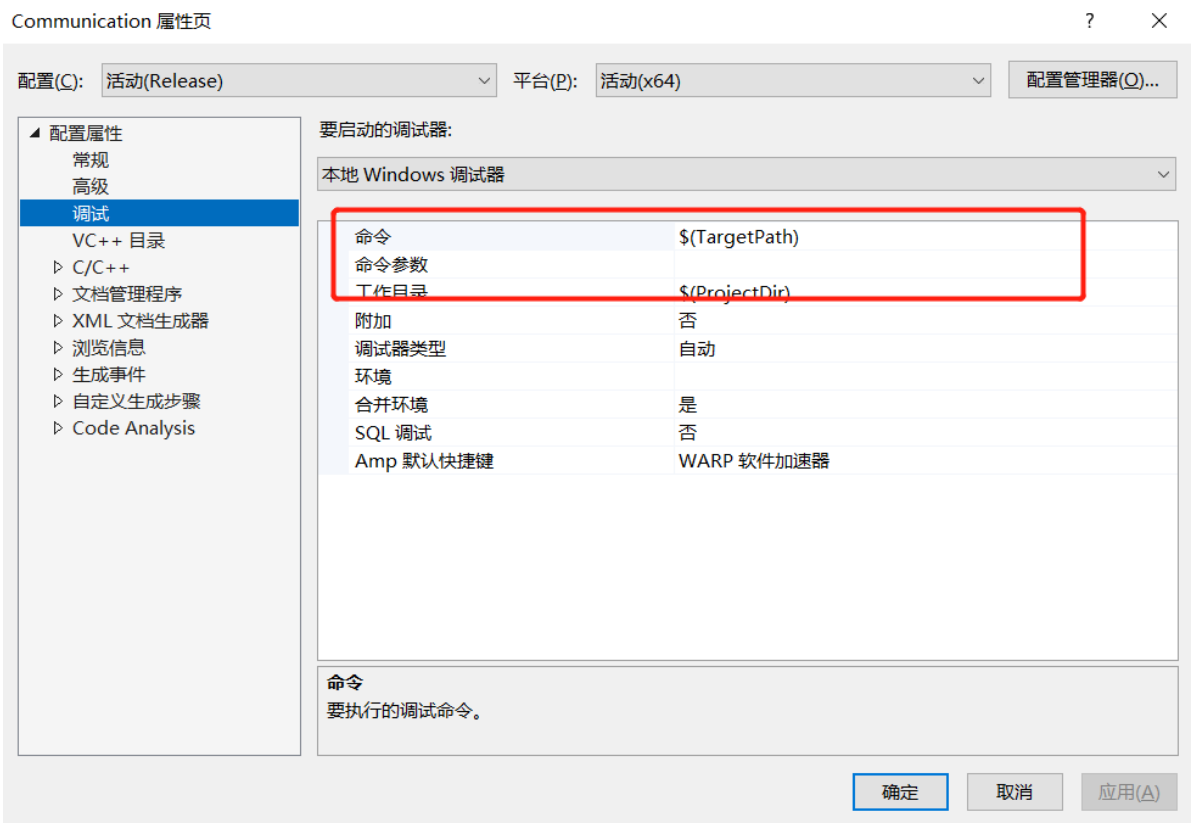
菜单栏下方一行



可以更改生成模式为`Debug`或者为`Release`

命令行参数的设置

左上方菜单栏`调试->调试属性`



在空格中加入命令行参数进行调试

cmd脚本的参数修改

以client.cmd文件为例，选中文件，右键单击，弹出菜单

可以编辑其中的文本进行参数的修改

可能用到的一些C++小知识

在此鸣谢\xfgg/\xfgg/\xfgg/\xfgg/\xfgg/\xfgg/\xfgg/，看到这里的选手可以到选手群膜一膜！！！

除非特殊指明，以下代码均在 MSVC 19.28.29913 `/std:c++17` 与 g++ 10.2 for linux `-std=c++17` 两个平台下通过。

一、计时相关

编写代码过程中，我们可能需要获取系统时间等一系列操作，C++ 标准库提供了这样的行为。尤其注意**不要**使用 Windows 平台上的 `GetTickCount` 或者 `GetTickCount64`！！！

std::chrono

头文件：`#include <chrono>`

获取时间戳（从公元 1970 年元旦到当前时刻的时间）

```cpp
#include <iostream>
#include <chrono>

int main()
{
    // 这一大串是我完全背着写下来的，还没看代码补全，厉害吧？
    // 不要问我这个乱七八糟的东西为什么这么长，去问 C++ 标准委员会，这是他们搞得神仙用的标准库（逃
    // 为了简短，强烈建议根据自己的喜好来用 using 命个名，例如 using msecs = std::chrono::milliseconds;
    auto sec = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now().time_since_epoch()).count();
    auto msec = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).count();
    std::cout << "从 1970 年元旦到现在的：秒数" << sec << "；毫秒数：" << msec << std::endl;
    return 0;
}
```

通过时间戳，我们可以既可以用来计时，也可以获取某个操作所花费的时间，还可以用来协调队友间的合作。

二、`auto` 类型推导

C++11 开始支持使用 `auto` 自动推导变量类型，废除了原有的作为 storage-class-specifier 的作用：

```cpp
int i = 4;
auto x = i;     // auto 被推导为 int，x 是 int 类型
auto& y = i;    // auto 仍被推导为 int，y 是 int& 类型
auto&& z = i;   // auto 被推导为 int&，z 是 int&&&，被折叠为 int&，即 z 与 y 同类型
auto&& w = 4;   // auto 被推导为 int，w 是 int&& 类型
```

三、lambda 表达式

lambda 表达式概述

lambda 表达式是 C++ 发展史上的一个重大事件，也是 C++ 支持函数式编程的重要一环。可以说，lambda 表达式不仅给 C++ 程序员带来了极大的便利，也开创了 C++ 的一个崭新的编程范式。但是同时 lambda 表达式也带来了诸多的语法难题，使用容易，但精通极难。

lambda 表达式确实是一个非常有用的语法特性。至少个人在学了 lambda 表达式之后，编写 C++ 代码就再也没有离开过。因为，它真的是非常的方便与易用。

lambda 表达式首先可以看做是一个临时使用的函数。它的一般格式如下：

```
[捕获列表] + lambda 声明（可选） + 复合语句

lambda 声明指的是：
(参数列表) + 一堆修饰符（可选）
```

下面是一个简单的例子：

```
#include <iostream>
using namespace std;
int main(void)
{
    auto GetOne = []{ return 1; };      // GetOne 是一个 lambda 表达式
    cout << GetOne() << endl;           // 使用起来就像一个函数，输出 1
    return 0;
}
```

它还可以有参数：

```
#include <iostream>
using namespace std;
int main(void)
{
    auto GetSum = [](int x, int y){ return x + y; };
    cout << GetSum(2, 3) << endl;   // 5
    return 0;
}
```

或者临时调用：

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << [](int x, int y){ return x + y; }(2, 3) << endl;    // 5
    return 0;
}
```

lambda 表达式的捕获

捕获的概念

lambda 表达式是不能够直接使用函数内的局部变量的（之后你将会看到这是为什么）。如果需要使用函数内的局部变量，需要手动进行捕获。捕获的方式有两种：按值捕获与按引用捕获。按值捕获，只会获得该值，而按引用捕获，则会获得函数内局部变量的引用。声明要捕获的变量就在 lambda 表达式的 `[]` 内：

- `[]`：不捕获任何局部变量
- `[x]`：按值捕获变量 `x`
- `[&y]`：按引用捕获变量 `y`
- `[=]`：按值捕获全部局部变量
- `[&]`：按引用捕获全部局部变量
- `[&, x]`：除了 `x` 按值捕获之外，其他变量均按引用捕获
- `[=, &y]`：什么意思不用我都说了吧
- `[r = x]`：声明一个变量 `r`，捕获 `x` 的值
- `[&r = y]`：声明一个引用 `r`，捕获 `y` 的引用
- `[x, y, &z, w = p, &r = q]`：作为练习
- `[&, x, y, p = z]`：这个也作为练习

这样我们就可以写出下面的代码了：

```
#include <iostream>
using namespace std;
int main(void)
{
    int x, y, z;
    cin >> x >> y;
    [x, y, &z](){ z = x + y; }();
    cout << z << endl;  // z = x + y
    return 0;
}
```

捕获 `this` 与 `*this`

当 lambda 表达式位于类的成员函数内时，该如何使用该类的成员变量呢？我们知道，在类的成员函数体内使用成员变量，都是通过 `this` 指针访问的，此处 `this` 作为成员函数的一个参数，因此只需要捕获 `this` 指针，就可以在 lambda 体内访问其成员变量了！

捕获时，我们可以选择捕获 `[this]`，也可以捕获 `[*this]`。区别是，前者捕获的是 `this` 指针本身，而后者是按值捕获 `this` 指针所指向的对象，也就是以 `*this` 为参数复制构造了一个新的对象。看下面的代码：

```
#include <iostream>
using namespace std;

struct Foo
{
    int m_bar;
    void Func()
    {
        [this]()
```

```
        {
            cout << ++m_bar << endl;
        }();
    }
};

int main()
{
    Foo foo;
    foo.m_bar = 999;
    foo.Func();     // 输出 1000
}
```

附注

需要注意的是，lambda 表达式的捕获发生在 **lambda 表达式定义处**，而不是 lambda 表达式调用处，比如：

```
int a = 4;
auto f = [a]() { cout << a << endl; };  // 此时捕获 a，值是 4
a = 9;
f();        // 输出 4，而非 9
```

> **C++ 真奇妙：不需要捕获的情况**
>
> 看这特殊的引用块就知道，本段内容仅作介绍，感觉较难者请跳过本块。
>
> 有时，即使是局部变量，不需要捕获也可以编译通过。这是 C++ 标准对编译器实现做出的妥协。这种现象叫做“常量折叠（constant folding）”；与之相对的是不能直接使用，必须进行捕获的情况，通常称作“odr-used”。这两个概念比较复杂，在此不做过多展开。看下面的例子：
>
> ```
> int Func1(const int& x) { return x; }
> void Func2()
> {
>  const int x = 4;
>  []()
>  {
>      int y = x;          // OK, constant folding
>      int z = Func1(x);  // Compile error! odr-used! x is not captured!
>  }();
> }
> ```
>
> 但是个别较老的编译器即使是 odr-used 也可能会编译通过

lambda 表达式的修饰符 `mutable`

lambda 表达式可以有一些修饰符，例如 `noexcept`、`mutable` 等，这里仅介绍 `mutable`。

lambda 表达式按值捕获变量时，捕获的变量默认是不可修改：

```
int a = 4;
auto f = [a]()
{
    ++a;    // Compile error: a cannot be modified!
};
```

但是我们可以通过加 `mutable` 关键字让它达到这个目的：

```
int a = 4;
auto f = [a]() mutable
{
    ++a;    // OK
    cout << a << endl;
};
f();        //输出 5
cout << a << endl;  //输出 4
```

需要注意的是，按值捕获变量是生成了一个新的变量副本，而非原来的变量，所以在 lambda 外的 `a` 的值仍然是 `4`

lambda 表达式的本质

本段内容仅是粗略地讲述，不做深入讨论。读者也可以跳过本块。

上面说了这么多语法规定，但是 lamdba 表达式究竟是什么？知道了这个可以帮助我们理解 lambda 表达式的这些规定。

C++17 标准中如此定义 lambda 的类型：

> The type of a *lambda-expression* (which is also the type of the closure object) is a unique, unnamed non-union class type, called the closure type....

lambda 表达式类型是一个独一无二的、没有名字的、并且不是联合体的类类型。我们把它叫做“**closure type**”。

后面还有一堆关于它性质的约束，这里就不展开了，大致上就是编译器可以自由决定它的很多性质，有兴趣的可以去翻阅《ISO/IEC 14882: 2017》第 8.1.5.1 款。

大体来看，一个 lamdba 表达式与一个类是大致上相同的。也就是说，lambda 表达式：

```
int a = 0, b = 0;
auto f = [a, &b](int x) { return a + b + x; }
f(5);
```

和下面的代码大致相同：

```
int a = 0, b = 0;
class __lambda__
{
private:
    int a;
    int& b;
public:
    __lambda__(int& a, int& b) : a(a), b(b) {}
    auto operator(int x) const { return a + b + x; }
};
__lambda__ f(a, b);
f.operator()(5);
```

不过它们两个**并不完全相同**。首先，不同编译器的实现本身就有不同；另外，它们在语法上的规定也有一些差别。篇幅所限，在此不做过多展开。

lambda 表达式的应用

看了上面这么多介绍，你可能要问：这东西能用什么用处？为什么不直接写个函数，或者是干脆不用 lambda 表达式而直接写在函数体里呢？有这个疑问是正常的。因为我上面给的例子都是可以不用 lambda 表达式就能轻松解决的。但是，lambda 表达式在很多应用场景具有不可替代的优势。最简单的例子，比如在局部，你要重复某些操作，但是另写一个函数又不是很方便，就可以用 lambda 表达式完成。此外，它最大的作用就是在函数式编程中，或者是其他需要回调函数的情况，以 lambda 表达式作为函数的参数以作为回调函数。在下面的教程中，例如多线程、智能指针，我们将会多次用到 lambda 表达式。届时你将会看到使用 lambda 表达式是多么的方便。

关于 lambda 表达式的其他说明

lambda 表达式还有很多有趣之处，例如泛型 lambda、返回 lambda 表达式的 lamdba 表达式，此外 `decltype` 在 lambda 表达式中的使用也是光怪陆离……总之，lambda 表达式非常有趣。

到了这里，相信你对 lambda 表达式已经有了相当的理解，就让我们来做一道简单的练习吧（狗头）

> 请给出下面程序的输出（该程序选自《ISO/IEC 14882: 2017 Programming Language --- C++》第 107 页）：
>
> ```
> #include <iostream>
> using namespace std;
>
> int main()
> {
> int a = 1, b = 1, c = 1;
> auto m1 = [a, &b, &c]() mutable
> {
>    auto m2 = [a, b, &c]() mutable
>    {
>        cout << a << b << c;
>        a = 4; b = 4; c = 4;
>    };
>    a = 3; b = 3; c = 3;
>    m2();
> ```

```
};
a = 2; b = 2; c = 2;
m1();
cout << a << b << c << endl;
return 0;
}
```

相信聪明的你一下就看出了答案。没错，答案就是我们小学二年级学习的数字：**123234**！怎么样，你答对了吗？

如果阅读本文之后你觉得 lambda 表达式很有趣，欢迎阅读《ISO/IEC 14882: 2017 Programming Language --- C++》110~120 页，或点击进入网址：[cppreference_lambda](#) 获取更多信息。

四、多线程相关

我们在游戏过程中，经常会遇到线程的相关操作，这个时候，使用 Windows 平台上的 `Sleep`、`CreateThread` 等都是行不通的。

线程睡眠

由于移动过程中会阻塞人物角色，因此玩家可能要在移动后让线程休眠一段时间，直到移动结束。C++ 标准库中使线程休眠需要包含头文件：`#include <thread>`。示例用法：

```
std::this_thread::sleep_for(std::chrono::milliseconds(20));      // 休眠 20 毫秒
std::this_thread::sleep_for(std::chrono::seconds(2));            // 休眠 2 秒

// 下面这个也能休眠 200 毫秒
std::this_thread::sleep_until(std::chrono::system_clock::now() +=
std::chrono::milliseconds(200));
```

休眠过程中，线程将被阻塞，而不继续进行，直到休眠时间结束方继续向下执行。

std::thread

头文件：`#include <thread>`。用于开启新的线程。示例代码：

```
#include <iostream>
#include <thread>
#include <functional>

void Func(int x, int& cnt)
{
    for (int i = 0; i < 110; ++i)
    {
        std::cout << "In Func: " << x << std::endl;
        ++cnt;
```

```
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }
}

int main()
{
    int cnt = 0;

    // 由于这种情况下函数的调用与传参不是同时的，提供参数在函数调用之前，因此以引用方式传递参数时需要用 std::ref
    std::thread thr(Func, 2021, std::ref(cnt));

    for (int i = 0; i < 50; ++i)
    {
        std::cout << "In main: " << 110 << std::endl;
        ++cnt;
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }

    thr.join();     // 等待子线程结束，在 thr 析构前若未 detach 则必须调用此函数，等待过程中主线程 main 被阻塞
    std::cout << "Count: " << cnt << std::endl;
    return 0;
}
```

或者使用 lambda 表达式达到同样效果：

```
#include <iostream>
#include <thread>
#include <functional>

int main()
{
    int cnt = 0, x = 2021;
    std::thread thr
    (
        [x, &cnt]()
        {
            for (int i = 0; i < 110; ++i)
            {
                std::cout << "In Func: " << x << std::endl;
                ++cnt;
                std::this_thread::sleep_for(std::chrono::milliseconds(20));
            }
        }
    );

    for (int i = 0; i < 50; ++i)
    {
        std::cout << "In main: " << 110 << std::endl;
        ++cnt;
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }

    thr.join();
```

```
    std::cout << "Count: " << cnt << std::endl;
    return 0;
}
```

如果不希望等待子线程结束，`main` 结束则程序结束，则可以构造临时对象调用 `detach` 函数：

```
#include <iostream>
#include <thread>
#include <functional>

int main()
{
    int cnt = 0, x = 2021;
    std::thread
    (
        [x, &cnt]()
        {
            for (int i = 0; i < 110; ++i)
            {
                std::cout << "In Func: " << x << std::endl;
                ++cnt;
                std::this_thread::sleep_for(std::chrono::milliseconds(20));
            }
        }
    ).detach();

    for (int i = 0; i < 50; ++i)
    {
        std::cout << "In main: " << 110 << std::endl;
        ++cnt;
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }

    std::cout << "Count: " << cnt << std::endl;
    return 0;
}
```

更多内容请参看（点击进入）：[cppreference thread]

std::future

头文件：`#include <future>`。用于创建异步任务执行，功能与开启 `thread` 相似，但更为强大与可靠。示例代码：

```
#include <iostream>
#include <future>

unsigned long long Fac(unsigned n)
{
    unsigned long long result = n == 0 ? 1 : n * Fac(n - 1);

```

```
    // 假设这个机器非常渣，算一次乘法需要半秒钟
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    return result;
}

int main()
{
    unsigned n;
    std::cin >> n;
    auto calculateFac = std::async(Fac, n);     // 开启一个异步任务，执行 Fac(n)
    for (unsigned i = 2; i < n; ++i)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        std::cout << "Half second passed!" << std::endl;
    }

    // 获取返回值，如果未执行完成则阻塞；如果不需要返回值，可以调用 calculateFac.wait();
    std::cout << "Waiting..." << std::endl;
    auto result = calculateFac.get();
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

五、智能指针

总述

头文件：`include <memory>`

智能指针是 C++ 标准库中对指针的封装，它的好处是可以不需要 `delete`，而自动对其指向的资源进行释放，这在一定程度上降低了 C++ 程序员管理内存的难度，但同时智能指针的使用也具有一定的技巧。

智能指针主要有三种：`shared_ptr`、`weak_ptr`、`unique_ptr`。

`std::shared_ptr`

概览

`shared_ptr` 可以说是最常用的智能指针了。它的用法最为灵活，内部实现方式是**引用计数**。即，它会记录有多少个 `shared_ptr` 正在指向某个资源，并当指向该资源的智能指针数为零时，调用相应的释放函数（默认为 `delete` 操作符）释放该资源。

像 `new` 会在自由存储区动态获取一块内存并返回其一样，如果要动态分配一块内存并得到其智能指针，可以使用 `std::make_shared` 模板，例如：

```cpp
#include <memory>

void Func()
{
    int* p = new int(110);                  // 在自由存储区 new 一个 int 对象，初值为 110
    auto sp = std::make_shared<int>(110);   // 在自由存储区 new 一个 int 对象，初值为 110
                                            // sp 被自动推导为 std::shared_ptr<int> 类型
    delete p;                               // 释放内存

    // 编译器调用 sp 的析构函数，并将其指向的 int 释放掉
}
```

关于引用计数：

```cpp
#include <memory>

void Func()
{
    int x = 110;
    {
        auto sp1 = std::make_shared<int>(x);    // 得到一个 int，初值为 110。

        // 上述此语句执行过后，只有一个智能指针 sp1 指向这个 int，引用计数为 1

        {
            auto sp2 = sp1;                     // 构造一个智能指针 sp2，指向 sp1 指向的内存，并将引用计数+1

            // 故此处引用计数为2

            std::cout << *sp2 << std::endl;     // 输出 110

            // 此处 sp2 生存期已到，调用 sp2 的析构函数，使引用计数-1，因此此时引用计数为1
        }

        // 此处 sp1 生命期也已经到了，调用 sp1 析构函数，引用计数再-1，故引用计数降为0
        // 也就是不再有智能指针指向它了，调用 delete 释放内存
    }
}
```

将普通指针交给智能指针托管：

```cpp
int* p = new int(110);
int* q = new int(110);
std::shared_ptr sp(p);  // 把 p 指向的内存交给 sp 托管，此后 p 便不需要 delete，sp 析构时会自动释放
std::shared_ptr sq;     // sq 什么也不托管
sq.reset(q);            // 让 sq 托管 q

//此后 p 与 q 便不需要再 delete
```

需要注意的是，这种写法是非常危险的，既可能导致 `p` 与 `q` 变为野指针，也可能造成重复 `delete`，我们应该更多使用 make_shared。

自定义释放函数

之前说过，默认情况下是释放内存的函数是 `delete` 运算符，但有时我们并不希望这样。比如下面的几个情况：

- 使用智能指针托管动态数组

```
#include <memory>

void IntArrayDeleter(int* p) { delete[] p; }

int main()
{
    std::shared_ptr<int> sp(new int[10], IntArrayDeleter);  // 让 IntArrayDeleter 作为释放资源的函数
    // sp 析构时自动调用 IntArrayDeleter 释放该 int 数组
    return 0;
}

// 或者利用 lambda 表达式：std::shared_ptr<int> sp(new int[10], [](int* p) { delete[] p; });
```

- 释放系统资源

  在编程过程中，难免与操作系统打交道，这时我们可能需要获取一系列的系统资源，并还给操作系统（实际上 `new` 和 `delete` 也就是一个例子）。一个比较有特色的例子就是 Windows API。在传统的 Win32 程序中，如果我们要在屏幕上进行绘制图形，我们首先需要获取设备的上下文信息，才能在设备上进行绘图。设想这样一个情景：我们有一个窗口，已经获得了指向这个窗口的句柄（即指针）`hWnd`，我们要在窗口上绘图，就要通过这个窗口句柄获取设备上下文信息。代码如下：

```
HDC hdc;                // DC: Device context，一个指向 DC 的句柄（HANDLE）
hdc = GetDC(hWnd);      // 获取设备上下文
/*执行绘图操作*/
ReleaseDC(hWnd, hdc);   // 绘图完毕，将设备上下文资源释放，归还给 Windows 系统
```

  使用智能指针对其进行托管，代码如下：

```
// 使用 lambda 表达式写法（推荐）
std::shared_ptr<void> sp(GetDC(hWnd), [hWnd](void* hdc) { ReleaseDC(hWnd, (HDC)hdc); });
```

```
// 不使用 lambda 表达式的写法：
struct Releaser
{
    HWND hWnd;
    Releaser(HWND hWnd) : hWnd(hWnd) {}
    void operator()(void* hdc)
    {
        ReleaseDC(hWnd, (HDC)hdc);
    }
```

```
};

void PaintFunc()
{
    /*...*/
    std::shared_ptr<void> sp(GetDC(hWnd), Releaser(hWnd));
    /*...*/
}
```

常见的错误用法

`std::shared_ptr` 虽然方便，但是也有一些错误用法，这个是常见的：

```
#include <memory>

void Func()
{
    int* p = new int(110);
    std::shared_ptr<int> sp(p);     // 让 sp 托管 p
    std::shared_ptr<int> sq(p);     // 让 sq 托管 p

    // Runtime Error! 程序至此崩溃
}
```

这是因为，只有复制构造函数里面才有使引用计数加1的操作。即当我们写 `std::shared_ptr<int> sq = sp` 的时候，确实引用计数变成了2，但是我们都用一个外部的裸指针 `p` 去初始化 `sp` 和 `sq`，智能指针并不能感知到它们托管的内存相同。所以 `sp` 和 `sq` 所托管的内存被看做是独立的。这样，当它们析构的时候，均会释放它们所指的内存，因此同一块内存被释放了两次，导致程序出错。所以个人还是推荐使用 `make_shared` ，而不是用裸指针去获取内存。

另一个著名的错误用法，请继续阅读 `std::weak_ptr`。

`std::weak_ptr`

看完了上面的 `shared_ptr` 的讲述，相信你已经对使用智能指针胸有成竹了。一切都用 `shared_ptr`、`make_shared` 就万事大吉了嘛！但事情可能没那么简单。看下面的例子：

```
#include <iostream>
#include <memory>

class B;

class A
{
public:
    void SetB(const std::shared_ptr<B>& ipB)
    {
        pB = ipB;
    }

private:
    std::shared_ptr<B> pB;
};

```

```cpp
class B
{
public:
    void SetA(const std::shared_ptr<A>& ipA)
    {
        pA = ipA;
    }

private:
    std::shared_ptr<A> pA;
};

void Func()
{
    auto pA = std::make_shared<A>();
    auto pB = std::make_shared<B>();
    pA->SetB(pB);
    pB->SetA(pA);
    // 内存泄露！！！
}

/*...*/
```

太糟糕了！上面的 `pA` 指向的的对象和 `pB` 指向的对象一直到程序结束之前永远不会被释放！如果不相信，可以在它们的析构函数里输出些什么试一试。相信学习了引用计数的你，一定能想出来原因。我们就把它当作一道思考题作为练习：为什么这两个对象不会被释放呢？（提示：注意只有引用计数降为0的时候才会释放）

实际上，`std::shared_ptr` 并不是乱用的。它除了作为一个指针之外，还表明了一种逻辑上的归属关系。从逻辑上看，类的成员代表一种归属权的关系，类的成员属于这个类。拥有 `shared_ptr` 作为**成员**的对象，是对 `shared_ptr` 所指向的对象具有所有权的，`shared_ptr` 也是基于这个理念设计的。但是，有时候我们并不希望这是个所有权的关系，例如我们有双亲和孩子的指针作为“人”的成员，但是人与人之间是平等相待和谐共处的，我们不能说一个人是另一个人的附属品。这时候，`std::weak_ptr` 便应运而生了！

`std::weak_ptr` 与 `shared_ptr` 的区别是，它指向一个资源，并不会增加引用计数。当指向一个资源的 `shared_ptr` 的数量为 0 的时候，即使还有 `weak_ptr` 在指，资源也会被释放掉。也是因此，`weak_ptr` 也是存在悬垂指针的可能的，即它指向的资源已经被释放掉。 也是因此，`weak_ptr` 不允许直接地被解引用，必须先转换为相应的 `shared_ptr` 才能解引用，获取其所指的资源。它的用法如下：

```cpp
auto sp = std::make_shared<int>(5);
std::weak_ptr<int> wp = sp; // 正确，让 wp 指向 sp 指向的资源
// std::shared_ptr<int> sp1 = wp;    // 错误，weak_ptr 不能直接赋值给 shared_ptr

/* Do something */

if (wp.expired())
{
    std::cout << "The resource has been released!" << std::endl;
}
else
{
```

```
    // std::cout << *wp << std::endl;   // Compile error! weak_ptr 不能直接使用!
    auto sp1 = wp.lock();       // 从 weak_ptr 中恢复出 shared_ptr，sp1 的类型为 std::shared_ptr<int>
    std::cout << *sp1 << std::endl;
}
```

从类的设计本身来看，`weak_ptr` 不会增加引用计数；从逻辑上看，`weak_ptr` 描述了一种联系，即 `weak_ptr` 的拥有者与其指向的对象之间不是一种归属关系，而是一种较弱的联系。一个类的对象只需知道另一个类的对象是谁，而不对其拥有占有权，这时候用 `weak_ptr` 是合适的。

上面的 `A` 类和 `B` 类的问题，将 `A` 和 `B` 成员从 `shared_ptr` 换成 `weak_ptr` 就会解决内存泄露的问题了！

`std::unique_ptr`

`std::unique_ptr` 顾名思义，独有的指针，即资源只能同时为一个 `unique_ptr` 所占有。它部分涉及到 `xvalue` 、右值引用与移动语义的问题，在此不做过多展开。

更多关于智能指针的知识，可以参考（点击进入）：

- cppreference shared_ptr
- cppreference weak_ptr
- cppreference unique_ptr

六、STL 容器相关

std::vector

头文件：`#include <vector>`，类似于可变长的数组，支持下标运算符 `[]` 访问其元素，此时与 C 风格数组用法相似。支持 `size` 成员函数获取其中的元素数量。

创建一个 `int` 型的 `vector` 对象：

```
std::vector<int> v { 9, 1, 2, 3, 4 };   // 初始化 vector 有五个元素，v[0] = 9, ...
v.emplace_back(10);          // 向 v 尾部添加一个元素，该元素饿构造函数的参数为 10（对于 int，只有一个语法意义上的构造函数，无真正的构造函数），即现在 v 有六个元素，v[5] 的值是 10
v.pop_back();                // 把最后一个元素删除，现在 v 还是 { 9, 1, 2, 3, 4 }
```

遍历其中所有元素的方式：

```
// std::vector<int> v;
for (int i = 0; i < (int)v.size(); ++i)
{
    /*可以通过 v[i] 对其进行访问*/
}
```

```cpp

for (auto itr = v.begin(); itr != v.end(); ++itr)
{
    /*
    * itr 作为迭代器，可以通过其访问 vector 中的元素。其用法与指针几乎完全相同。
    * 可以通过 *itr 得到元素；以及 itr-> 的用法也是支持的
    * 实际上它内部就是封装了指向 vector 中元素的指针
    * 此外还有 v.cbegin()、v.rbegin()、v.crbegin() 等
    * v.begin()、v.end() 也可写为 begin(v)、end(v)
    */
}

for (auto&& elem : v)
{
    /*
    * elem 即是 v 中每个元素的引用，也可写成 auto& elem : v
    * 它完全等价于：
    * {
    *   auto&& __range = v;
    *   auto&& __begin = begin(v);
    *   auto&& __end = end(v);
    *   for (; __begin != __end; ++__begin)
    *   {
    *       auto&& elem = *__begin;
    *       // Some code
    *   }
    * }
    */
}
```

例如：

```cpp
for (auto elem&& : v) { std::cout << elem << ' '; }
std::cout << std::endl;
```

作为 STL 的容器之一，其具有容器的通用接口。但是由于这比较复杂，在此难以一一展开。有兴趣的同学可以在下方提供的链接里进行查阅。

注：请千万不要试图使用 `std::vector<bool>`，若需使用，请用 `std::vector<char>` 代替！

更多用法参见（点击进入）：[cppreference vector]

std::array

头文件：`#include <array>`，C 风格数组的类封装版本。

用法与 C 风格的数组是基本相似的，例如：

```cpp
std::array<double, 5> arr { 9.0, 8.0, 7.0, 6.0, 5.0 };
std::cout << arr[2] << std::endl;   // 输出 7.0
```

同时也支持各种容器操作：

```cpp
double sum = 0.0;
for (auto itr = begin(arr); itr != end(arr); ++itr)
{
    sum += *itr;
}
// sum 结果是 35
```

更多用法参见（点击进入）：cppreference_array。