

工具使用合集

工具使用合集

- Windows接口使用

 - C++接口使用说明

 - Python接口使用说明

- Linux接口使用

 - C++使用说明

 - Python使用说明

- Visual Studio使用说明

 - 生成模式的设置

 - 命令行参数的设置

 - cmd脚本的参数修改

- C++接口必看

 - 计时相关

 - 线程睡眠

 - 异步接口的使用

 - `auto` 类型推导

 - STL相关

 - `std::vector`

 - `std::array`

- Python接口必看

- C++相关小知识

 - lambda表达式

 - lambda表达式概述

 - lambda 表达式的捕获

 - 捕获的概念

 - 捕获 `this` 与 `*this`

 - 附注

 - lambda 表达式的修饰符 `mutable`

 - lambda 表达式的本质

 - lambda 表达式的应用

 - 关于 lambda 表达式的其他说明

 - `std::thread`

 - 智能指针

 - 总述

 - `std::shared_ptr`

 - 概览

 - 自定义释放函数

 - 常见的错误用法

 - `std::weak_ptr`

 - `std::unique_ptr`

Windows接口使用

C++接口使用说明

- 首先把 lib 文件夹复制到 CAPI\cpp\ 文件夹下
- 然后在 CAPI\cpp\API\src\AI.cpp 中编写代码
- 然后用 Visual Studio 打开 CAPI\cpp\CAPI.sln 编译，注意 Debug 模式和 Release 模式的区别
- 最后使用 RunCppDebug.cmd 或 RunCppRelease.cmd，需要和上面的编译保持一致。

Python接口使用说明

- 首先在 Python 环境下运行 GeneratePythonProto.cmd，以安装必要的包、并生成对应的 grpc python 文件
- 然后在 CAPI\python\PyAPI\AI.py 中编写代码
- 最后通过运行 RunPython.cmd 执行比赛代码

Linux接口使用

C++使用说明

- 首先自行安装 gRPC，具体方法可以参考官方教程<https://grpc.io/docs/languages/cpp/quickstart/>。
- 然后在 CAPI\cpp\API\src\AI.cpp 中编写代码
- 接下来用 cmake，对 CAPI\cpp\CMakeLists.txt 进行编译
- 最后使用 RunCpp.sh 执行比赛代码

Python使用说明

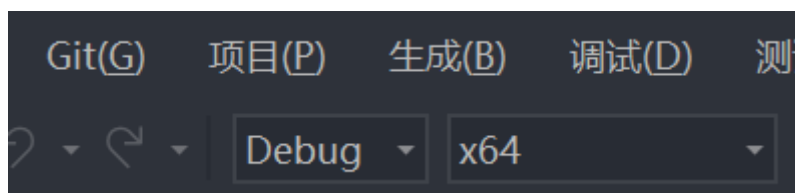
- 首先在 Python 环境下运行 GeneratePythonProto.cmd，以安装必要的包、并生成对应的 grpc python 文件
- 然后在 CAPI\python\PyAPI\AI.py 中编写代码
- 最后通过运行 RunPython.cmd 执行比赛代码

Visual Studio使用说明

比赛只保证!!! 支持VS2022，选手使用其他版本后果自负。

生成模式的设置

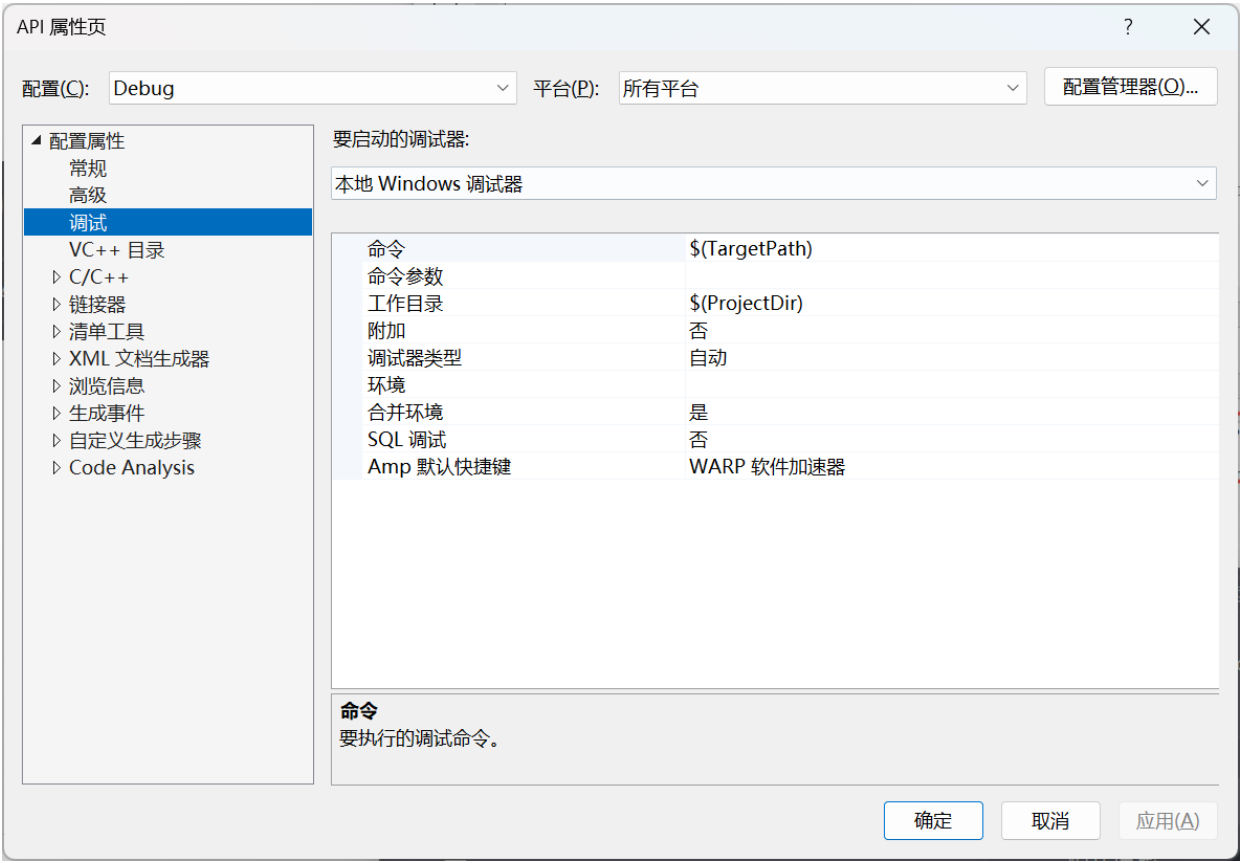
菜单栏下方一行



可以更改生成模式为 Debug 或 Release

命令行参数的设置

左上方菜单栏 调试->调试属性



在命令参数一栏中加入命令行参数进行调试

cmd脚本的参数修改

右键点击 .cmd 或 .bat 文件之后，选择编辑就可以开始修改文件。通过在一行的开头加上 ::，可以注释掉该行。

C++接口必看

在此鸣谢\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。

由于我们的比赛最终会运行在Linux平台上，因此程设课上学到的一些只适用于Windows的C++操作很可能并不能正确执行。此外，代码中使用了大量Modern C++中的新特性，可能会使选手在编程过程中遇到较大困难。因此，此处介绍一些比赛中使用C++接口必须了解的知识。

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` !!! 应当使用 `std::chrono`

头文件: `#include <chrono>`

可以用于获取时间戳，从而用于计时、例如计算某个操作花费的时间，或者协调队友间的合作。

```
1  #include <iostream>
2  #include <chrono>
3  int main()
4  {
5      auto sec = std::chrono::duration_cast<std::chrono::seconds>
6      (std::chrono::system_clock::now().time_since_epoch()).count();
7      auto msec = std::chrono::duration_cast<std::chrono::milliseconds>
8      (std::chrono::system_clock::now().time_since_epoch()).count();
9      std::cout << "从 1970 年元旦到现在的: 秒数" << sec << "; 毫秒数: " << msec <<
10     std::endl;
11     return 0;
12 }
```

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:

```
1  std::this_thread::sleep_for(std::chrono::milliseconds(20));    // 休眠 20 毫秒
2  std::this_thread::sleep_for(std::chrono::seconds(2));          // 休眠 2 秒
3
4  // 下面这个也能休眠 200 毫秒
5  std::this_thread::sleep_until(std::chrono::system_clock::now() +=
6  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。

异步接口的使用

本届比赛中，我们可能会看到类似 `std::future<bool>` 这样类型的接口返回值，这实际上是一个异步接口。在调用同步接口后，在接口内的函数未执行完之前，线程通常会阻塞住；但是异步接口的调用通常不会阻塞当前线程，而是会另外开启一个线程进行操作，当前线程则继续向下执行。当调用 `get()` 方法时，将返回异步接口的值，若此时异步接口内的函数依然未执行完，则会阻塞当前线程。

如果不需要返回值或没有返回值，但是希望接口内的函数执行完之后再进行下一步，即将接口当做常规的同步接口来调用，也可以调用 `wait()` 方法。

```
1  #include <iostream>
2  #include <thread>
3  #include <future>
4  #include <chrono>
5
6  int f_sync()
7  {
8      std::this_thread::sleep_for(std::chrono::seconds(1));
9      return 8;
10 }
11
12 std::future<int> f_async()
13 {
14     return std::async(std::launch::async, []()
15                     { std::this_thread::sleep_for(std::chrono::seconds(1));
16                       return 8; });
17 }
18
19 int main()
20 {
21     auto start = std::chrono::system_clock::now();
22     std::cout << std::chrono::duration_cast<std::chrono::duration<double,
std::::milli>>(std::chrono::system_clock::now() - start).count() << std::endl;
23     auto x = f_async();
24     std::cout << std::chrono::duration_cast<std::chrono::duration<double,
std::::milli>>(std::chrono::system_clock::now() - start).count() << std::endl;
25     std::cout << x.get() << std::endl;
26     std::cout << std::chrono::duration_cast<std::chrono::duration<double,
std::::milli>>(std::chrono::system_clock::now() - start).count() << std::endl;
27     auto y = f_sync();
28     std::cout << std::chrono::duration_cast<std::chrono::duration<double,
std::::milli>>(std::chrono::system_clock::now() - start).count() << std::endl;
29     std::cout << y << std::endl;
30     std::cout << std::chrono::duration_cast<std::chrono::duration<double,
std::::milli>>(std::chrono::system_clock::now() - start).count() << std::endl;
31 }
```

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：

```

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

```

STL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:

```

1  std::vector<int> v { 9, 1, 2, 3, 4 }; // 初始化 vector 有五个元素, v[0] = 9, ...
2  v.emplace_back(10); // 向 v 尾部添加一个元素, 该元素构造函数参数为 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
3  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:

```

1  // std::vector<int> v;
2  for (int i = 0; i < (int)v.size(); ++i)
3  {
4      /*可以通过 v[i] 对其进行访问*/
5  }
6
7  for (auto itr = v.begin(); itr != v.end(); ++itr)
8  {
9      /*
10     * 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。
11     * 可以通过 *itr 得到元素; 以及 itr-> 的用法也是支持的
12     * 实际上它内部就是封装了指向 vector 中元素的指针
13     * 此外还有 v.cbegin()、v.rbegin()、v.crbegin() 等
14     * v.begin()、v.end() 也可写为 begin(v)、end(v)
15     */
16 }
17
18 for (auto&& elem : v)
19 {
20     /*
21     * elem 即是 v 中每个元素的引用, 也可写成 auto& elem : v
22     * 它完全等价于:
23     * {

```

```

24     *   auto&& __range = v;
25     *   auto&& __begin = begin(v);
26     *   auto&& __end = end(v);
27     *   for (; __begin != __end; ++__begin)
28     *   {
29     *       auto&& elem = *__begin;
30     *       // Some code
31     *   }
32     * }
33     */
34 }

```

例如：

```

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

```

作为 STL 的容器之一，它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：

```

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

```

同时也支持各种容器操作：

```

1  double sum = 0.0;
2  for (auto itr = begin(arr); itr != end(arr); ++itr)
3  {
4      sum += *itr;
5  }
6  // sum 结果是 35

```

更多用法参见（点击进入）：[cppreference array](#)。

Python接口必看

比赛中的Python接口大多使用异步接口，即返回一个类似于 `Future[bool]` 的值。为了获取实际的值，需要调用 `result()` 方法。

```
1  from concurrent.futures import Future, ThreadPoolExecutor
2  import time
3
4
5  class Cls:
6      def __init__(self):
7          self.__pool: ThreadPoolExecutor = ThreadPoolExecutor(10)
8
9      def Test(self, a: int, b: int) -> Future[int]:
10         def test():
11             time.sleep(0.5)
12             return a + b
13
14         return self.__pool.submit(test)
15
16
17  if __name__ == '__main__':
18      f1 = Cls().Test(1, 2)
19      print(time.time())
20      print(f1.result())
21      print(time.time())
22
```

C++相关小知识

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：


```
1 [捕获列表] + lambda 声明（可选） + 复合语句
2
3 lambda 声明指的是：
4 （参数列表） + 一堆修饰符（可选）
```

下面是一个简单的例子：

```
1 #include <iostream>
2 using namespace std;
3 int main(void)
4 {
5     auto GetOne = []{ return 1; };           // GetOne 是一个 lambda 表达式
6     cout << GetOne() << endl;              // 使用起来就像一个函数，输出 1
7     return 0;
8 }
```

它还可以有参数：

```
1 #include <iostream>
2 using namespace std;
3 int main(void)
4 {
5     auto GetSum = [](int x, int y){ return x + y; };
6     cout << GetSum(2, 3) << endl;          // 5
7     return 0;
8 }
```

或者临时调用：

```
1 #include <iostream>
2 using namespace std;
3 int main(void)
4 {
5     cout << [](int x, int y){ return x + y; }(2, 3) << endl;    // 5
6     return 0;
7 }
```

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：

```

1  #include <iostream>
2  using namespace std;
3  int main(void)
4  {
5      int x, y, z;
6      cin >> x >> y;
7      [x, y, &z]() { z = x + y; }();
8      cout << z << endl; // z = x + y
9      return 0;
10 }
```

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：

```

1  #include <iostream>
2  using namespace std;
3
4  struct Foo
5  {
6      int m_bar;
7      void Func()
8      {
9          [this]()
10         {
11             cout << ++m_bar << endl;
12         }();
13     }
14 }
```

```

13     }
14 };
15
16 int main()
17 {
18     Foo foo;
19     foo.m_bar = 999;
20     foo.Func();    // 输出 1000
21 }

```

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：

```

1 int a = 4;
2 auto f = [a]() { cout << a << endl; }; // 此时捕获 a，值是 4
3 a = 9;
4 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：

```

1 int Func1(const int& x) { return x; }
2 void Func2()
3 {
4     const int x = 4;
5     []()
6     {
7         int y = x;    // OK, constant folding
8         int z = Func1(x); // Compile error! odr-used! x is not captured!
9     }();
10 }

```

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：

```
1 int a = 4;
2 auto f = [a]()
3 {
4     ++a;    // Compile error: a cannot be modified!
5 };
```

但是我们可以通过加 `mutable` 关键字让它达到这个目的：

```
1 int a = 4;
2 auto f = [a]() mutable
3 {
4     ++a;    // OK
5     cout << a << endl;
6 };
7 f();        //输出 5
8 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则，但是 lambda 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 lambda 表达式与一个类是大致上相同的。也就是说，lambda 表达式：

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

和下面的代码大致相同：

```
1  int a = 0, b = 0;
2  class __lambda__
3  {
4  private:
5      int a;
6      int& b;
7  public:
8      __lambda__(int& a, int& b) : a(a), b(b) {}
9      auto operator(int x) const { return a + b + x; }
10 };
11 __lambda__ f(a, b);
12 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 lambda 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）：

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 1, b = 1, c = 1;
7      auto m1 = [a, &b, &c]() mutable
8      {
9          auto m2 = [a, b, &c]() mutable
10         {
```

```

11     cout << a << b << c;
12     a = 4; b = 4; c = 4;
13 };
14     a = 3; b = 3; c = 3;
15     m2();
16 };
17     a = 2; b = 2; c = 2;
18     m1();
19     cout << a << b << c << endl;
20     return 0;
21 }

```

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。

std::thread

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

```

1  #include <iostream>
2  #include <thread>
3  #include <functional>
4
5  void Func(int x, int& cnt)
6  {
7      for (int i = 0; i < 110; ++i)
8      {
9          std::cout << "In Func: " << x << std::endl;
10         ++cnt;
11         std::this_thread::sleep_for(std::chrono::milliseconds(20));
12     }
13 }
14
15 int main()
16 {
17     int cnt = 0;
18
19     // 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
20     std::thread thr(Func, 2021, std::ref(cnt));
21
22     for (int i = 0; i < 50; ++i)
23     {

```

```

24     std::cout << "In main: " << 110 << std::endl;
25     ++cnt;
26     std::this_thread::sleep_for(std::chrono::milliseconds(20));
27 }
28
29 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
30 std::cout << "Count: " << cnt << std::endl;
31 return 0;
32 }

```

或者使用 lambda 表达式达到同样效果:

```

1  #include <iostream>
2  #include <thread>
3  #include <functional>
4
5  int main()
6  {
7      int cnt = 0, x = 2021;
8      std::thread thr
9      (
10         [x, &cnt]()
11         {
12             for (int i = 0; i < 110; ++i)
13             {
14                 std::cout << "In Func: " << x << std::endl;
15                 ++cnt;
16                 std::this_thread::sleep_for(std::chrono::milliseconds(20));
17             }
18         }
19     );
20
21     for (int i = 0; i < 50; ++i)
22     {
23         std::cout << "In main: " << 110 << std::endl;
24         ++cnt;
25         std::this_thread::sleep_for(std::chrono::milliseconds(20));
26     }
27
28     thr.join();
29     std::cout << "Count: " << cnt << std::endl;
30     return 0;
31 }

```

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:

```

1  #include <iostream>
2  #include <thread>
3  #include <functional>
4
5  int main()
6  {
7      int cnt = 0, x = 2021;
8      std::thread
9      (
10         [x, &cnt]()
11         {
12             for (int i = 0; i < 110; ++i)
13             {
14                 std::cout << "In Func: " << x << std::endl;
15                 ++cnt;
16                 std::this_thread::sleep_for(std::chrono::milliseconds(20));
17             }
18         }
19     ).detach();
20
21     for (int i = 0; i < 50; ++i)
22     {
23         std::cout << "In main: " << 110 << std::endl;
24         ++cnt;
25         std::this_thread::sleep_for(std::chrono::milliseconds(20));
26     }
27
28     std::cout << "Count: " << cnt << std::endl;
29     return 0;
30 }

```

更多内容请参看 (点击进入) : [cppreference thread](#)

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：

```
1  #include <memory>
2
3  void Func()
4  {
5      int* p = new int(110);           // 在自由存储区 new 一个 int 对象，初值
    为 110
6      auto sp = std::make_shared<int>(110); // 在自由存储区 new 一个 int 对象，初值
    为 110
7                                           // sp 被自动推导为
    std::shared_ptr<int> 类型
8      delete p;                         // 释放内存
9
10     // 编译器调用 sp 的析构函数，并将其指向的 int 释放掉
11 }
```

关于引用计数：

```
1  #include <memory>
2
3  void Func()
4  {
5      int x = 110;
6      {
7          auto sp1 = std::make_shared<int>(x);    // 得到一个 int，初值为 110。
8
9          // 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
10
11         {
12             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
13
14             // 故此处引用计数为2
15
16             std::cout << *sp2 << std::endl;      // 输出 110
17
18             // 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
19         }
20
21         // 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
22         // 也就是不再有智能指针指向它了，调用 delete 释放内存
```

```
23     }
24 }
```

将普通指针交给智能指针托管：

```
1  int* p = new int(110);
2  int* q = new int(110);
3  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
4  std::shared_ptr sq;    // sq 什么也不托管
5  sq.reset(q);          // 让 sq 托管 q
6
7  //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

```
1  #include <memory>
2
3  void IntArrayDeleter(int* p) { delete[] p; }
4
5  int main()
6  {
7      std::shared_ptr<int> sp(new int[10], IntArrayDeleter); // 让
    IntArrayDeleter 作为释放资源的函数
8      // sp 析构时自动调用 IntArrayDeleter 释放该 int 数组
9      return 0;
10 }
11
12 // 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統资源，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：

```

1 HDC hdc;           // DC: Device context, 一个指向 DC 的句柄 (HANDLE)
2 hdc = GetDC(hwnd); // 获取设备上下文
3 /*执行绘图操作*/
4 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:

```

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

```

```

1 // 不使用 lambda 表达式的写法:
2 struct Releaser
3 {
4     HWND hwnd;
5     Releaser(HWND hwnd) : hwnd(hwnd) {}
6     void operator()(void* hdc)
7     {
8         ReleaseDC(hwnd, (HDC)hdc);
9     }
10 };
11
12 void PaintFunc()
13 {
14     /*...*/
15     std::shared_ptr<void> sp(GetDC(hwnd), Releaser(hwnd));
16     /*...*/
17 }

```

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:

```

1 #include <memory>
2
3 void Func()
4 {
5     int* p = new int(110);
6     std::shared_ptr<int> sp(p); // 让 sp 托管 p
7     std::shared_ptr<int> sq(p); // 让 sq 托管 p
8
9     // Runtime Error! 程序至此崩溃
10 }

```

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：

```
1  #include <iostream>
2  #include <memory>
3
4  class B;
5
6  class A
7  {
8  public:
9      void SetB(const std::shared_ptr<B>& ipB)
10     {
11         pB = ipB;
12     }
13
14 private:
15     std::shared_ptr<B> pB;
16 };
17
18 class B
19 {
20 public:
21     void SetA(const std::shared_ptr<A>& ipA)
22     {
23         pA = ipA;
24     }
25
26 private:
27     std::shared_ptr<A> pA;
28 };
29
30 void Func()
31 {
32     auto pA = std::make_shared<A>();
33     auto pB = std::make_shared<B>();
34     pA->SetB(pB);
35     pB->SetA(pA);
36     // 内存泄露!!!
37 }
38
39 /*...*/
```

太糟糕了！上面的 `pA` 指向的对象和 `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：

```
1 auto sp = std::make_shared<int>(5);
2 std::weak_ptr<int> wp = sp; // 正确，让 wp 指向 sp 指向的资源
3 // std::shared_ptr<int> sp1 = wp; // 错误，weak_ptr 不能直接赋值给 shared_ptr
4
5 /* Do something */
6
7 if (wp.expired())
8 {
9     std::cout << "The resource has been released!" << std::endl;
10 }
11 else
12 {
13     // std::cout << *wp << std::endl; // Compile error! weak_ptr 不能直接使用！
14     auto sp1 = wp.lock(); // 从 weak_ptr 中恢复出 shared_ptr, sp1 的类型为
15     std::shared_ptr<int>
16     std::cout << *sp1 << std::endl;
17 }
```

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）：

- [c++reference shared_ptr](#)
- [c++reference weak_ptr](#)
- [c++reference unique_ptr](#)