

# 第一章、从C语言过渡到C++

## 1.命名空间的引入

首先看一段代码

```
#include <stdio.h>

int rand = 10;
int main()
{
    printf("%d\n", rand);
    return 0;
}
```

编译并不会报错，但如果加上头文件 `stdlib` 呢？

✘ C2365 "rand": 重定义; 以前的定义是“函数”

编译器告诉我们，`rand` 重定义了，以前的定义是函数，因为在 `stdlib` 这个库中，`rand` 是一个函数，将头文件展开变量 `rand` 会和库里面的 `rand` 冲突

我们在编码时，会创建很多的变量、函数，我们并不能保证每个变量和函数都不能同名，或者是不和库中的函数重名，所以我们引入关键字 `namespace`，基本格式如下：

```
namespace name
{ }
```

`namespace` 的本质是定义一个命名空间域，避免头文件展开后会和库函数冲突，这个域跟全局域各自独立，不同的域可以定义同名变量

```
#include <stdio.h>
#include <stdlib.h>
// 引入namespace进行命名隔离
namespace Vect
{
    int rand = 0;
}
```

```
int main()
{
    printf("%d", Vect::rand);
    return 0;
}
```

编译通过：

```
显示输出来源(S): 生成
1>test.cpp
1>Entery.vcxproj -> D:\CODE\CPP\new\Entery\x64\Debug\Entery.exe
===== 生成: 1 成功, 0 失败, 0 最新, 0 已跳过 =====
===== 生成 于 18:50 完成, 耗时 02.582 秒 =====
```

下面介绍 `namespace` 的一些性质

## 1.1. `namespace` 的性质

---

- 不影响变量/函数的生命周期，仅作名字的隔离作用
- `namesp` 只能定义在全局，可以嵌套定义，多文件下同名的命名空间会进行合并

```
#include <stdio.h>
#include "head1.h"

namespace Vect
{
    // 1. 命名空间可以定义变量/函数/类型
    int num = 10;
    int* ptr = &num;

    void Print()
    {
        printf("HELLO,C++!\n");
    }

    struct node
    {
        int val;
        struct node* next;
    };
};
```

```
// 2. 命名空间可以嵌套定义
namespace coke
{
    // ...
}

int main()
{
    return 0;
}
```

## 1.2. namespace 的使用

---

编译器在编译时，会按照一定的顺序进行查找变量/函数/类型

1. 局部域，通俗理解是自己家
2. 全局域，通俗理解是公共区域；展开的命名空间（别人家声明你可访问），这两个域相同优先级

而查找命名空间也有三种方式：

1. `using` 将命名空间中某个成员展开，项目中推荐这种方式
2. 使用域作用限定符 `::` 指定查找
3. `using` 展开全部的命名空间，项目中非常不推荐，日常练习无所谓

```
#include <stdio.h>
// 使用方法

// 1. 域作用限定符指定访问
int num = 10;
namespace Vect
{
    int num = 0;
    char ch = '?';
}

int main()
{
```

```

    printf("%d\n", Vect::num);
    printf("%c\n", Vect::ch);
    // 若域作用限定符左边无命名空间，默认查找全局域
    printf("%d\n", ::num);
    printf("%d\n", num);
}

```

```

int num = 10;
namespace Vect
{
    int num = 0;
    char ch = '?';
}

int main()
{
    printf("%d\n", Vect::num);
    printf("%c\n", Vect::ch);
    // 若域作用限定符左边无命名空间，默认查找全局域
    printf("%d\n", ::num);
    printf("%d\n", num);
}

```

Microsoft Visual Studio 调试

```

0
?
10
10
D:\CODE\CPP\new\Entery\x64
代码为 0 (0x0)

```

先找局部域中的，无，接着找全局域

```

// 2. using展开局部对象
namespace Vect
{
    int num = 0;
    char ch = '?';
}
using Vect::num;
int main()
{
    printf("%d\n", num);
    printf("%c\n", Vect::ch);
    // 若域作用限定符左边无命名空间，默认查找全局域
}

```

// 2. using展开局部对象

```
namespace Vect
```

```
{
```

```
    int num = 0;
```

```
    char ch = '?';
```

```
}
```

```
using Vect::num;
```

```
int main()
```

```
{
```

```
    printf("%d\n", num);
```

```
    printf("%c\n", Vect::ch);
```

```
    // 若域作用限定符左边无命名空间，默认查找全局域
```

```
}
```

Microsoft Visual S

0

?

D:\CODE

// 3. using展开全体对象

```
namespace Vect
```

```
{
```

```
    int num = 0;
```

```
    char ch = '?';
```

```
}
```

```
using namespace Vect;
```

```
int main()
```

```
{
```

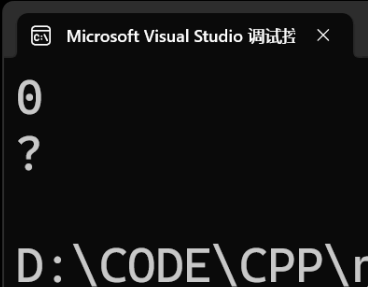
```
    printf("%d\n", num);
```

```
    printf("%c\n", ch);
```

```
    // 若域作用限定符左边无命名空间，默认查找全局域
```

```
}
```

```
// 3. using展开全体对象
namespace Vect
{
    int num = 0;
    char ch = '?';
}
using namespace Vect;
int main()
{
    printf("%d\n", num);
    printf("%c\n", ch);
    // 若域作用限定符左边无命名空间，默认查找全局域
}
```



The screenshot shows the Microsoft Visual Studio IDE. The main editor window displays the C++ code. To the right, there is a 'Microsoft Visual Studio 调试器' (Microsoft Visual Studio Debugger) window showing the output of the program, which is '0' followed by a newline and then '?'. Below the output, the file path 'D:\CODE\CPP\r' is visible.

思考：局部域、全局域、命名空间域有什么联系和区别

局部域、全局域和命名空间域是 C++ 中三种作用域：

- **局部域**：在函数或代码块内部，只在当前块内有效。
- **全局域**：定义在所有函数外部，整个程序都能访问，可能需要注意 `extern`。
- **命名空间域**：用来组织代码，防止命名冲突，通过 `namespace::变量` 展开局部变量 或者 全部展开 三种方式访问。

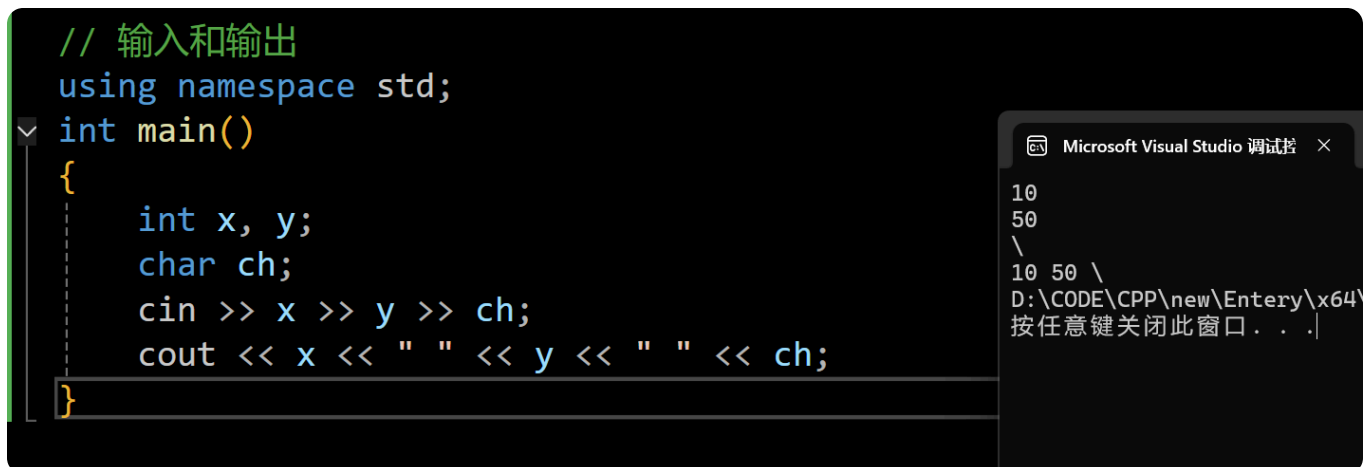
三者的区别在于作用范围和访问方式，它们之间可以嵌套使用，但同名时，遵循局部优先原则，命名空间通过域作用限定符来区分。实际开发中，我倾向使用局部变量和命名空间来增强代码的可读性和安全性，尽量少使用全局变量。



## 2. C++的输入与输出

- `iostream` 表示 `input output stream` 是标准输入输出流的库
- `std::cin` 是 `stream` 类的对象，用于输入窄字符
- `std::cout`stream` 类的对象，用于输出窄字符
- `<<` `>>` 分别是流插入和流提取符
- C++ 的输入输出可以自动识别变量类型(本质是通过函数重载实现)
- `cout/cin/endl` 等都属于 C++ 标准库，C++ 标准库都放在一个叫 `std(standard)` 的命名空间中，所以要通过命名空间的使用方式去用他们

```
// 输入和输出
using namespace std;
int main()
{
    int x, y;
    char ch;
    cin >> x >> y >> ch;
    cout << x << " " << y << " " << ch;
}
```



```
// 输入和输出
using namespace std;
int main()
{
    int x, y;
    char ch;
    cin >> x >> y >> ch;
    cout << x << " " << y << " " << ch;
}
```

Microsoft Visual Studio 调试控制台

```
10
50
\
10 50 \
D:\CODE\CPP\new\Entery\x64\
按任意键关闭此窗口 . . .
```



### 3.缺省参数（默认参数）

- **缺省参数**是函数声明或定义时，给函数形参指定的一个默认值，如果调用该函数未传参，则使用这个缺省参数，若调用时传参了，则优先使用传递的实参
- 缺省参数分为全缺省和半缺省，全缺省是将所有形参都给一个默认值，而半缺省是给部分形参一个默认值，C++ 规定半缺省参数必须**从右往左依次连续缺省**，不能间隔跳跃给缺省值
- 带缺省值函数调用时，C++ 规定必须**从左往右依次传递实参**，不能间隔跳跃给实参
- 函数**声明和定义分离**时，缺省参数不能在函数声明和定义中同时出现，规定**必须函数声明给缺省值**

```
// 全缺省
int Add(int a = 10, int b = 20)
{
    return a + b;
}
```

```

}

// 半缺省 必须从左往右依次传递实参，不能间隔跳跃给实参
void Print(int a, int b = 10, int c = 20)
{
    cout << a << "    " << b << "    " << c << endl;
}

int main()
{
    // 如果调用该函数未传参，则使用这个缺省参数，若调用时传参了，则优先使用传递的实参
    int ret1 = Add();
    int ret2 = Add(1,2);
    cout << ret1 << "    " << ret2 << endl;
    // 半缺省调用 必须从左往右依次传递实参，不能间隔跳跃给实参
    Print(11);
    Print(11, 12);
    Print(11, 12, 13);

    return 0;
}

```

```

// 全缺省
int Add(int a = 10, int b = 20)
{
    return a + b;
}

// 半缺省 必须从左往右依次传递实参，不能间隔跳跃给实参
void Print(int a, int b = 10, int c = 20)
{
    cout << a << "    " << b << "    " << c << endl;
}

int main()
{
    // 如果调用该函数未传参，则使用这个缺省参数，若调用时传参了，则优先使用传递的实参
    int ret1 = Add();
    int ret2 = Add(1,2);
    cout << ret1 << "    " << ret2 << endl;
    // 半缺省调用 必须从左往右依次传递实参，不能间隔跳跃给实参
    Print(11);
    Print(11, 12);
    Print(11, 12, 13);

    return 0;
}

```

Microsoft Visual Studio 调试控制台

```

30    3
11    10    20
11    12    20
11    12    13

```

D:\CODE\NewCPP\...  
按任意键关闭此窗口



```
// 函数声明定义分离情况,给声明缺省值,定义不要给
int Mul(int a = 1, int b = 6);

int main()
{
    return 0;
}

// 给缺省值了 C2572重定义默认参数
int Mul(int a = 1, int b = 6)
{
    return a * b;
}
```

```
1>D:\CODE\NewCPP\Entery\Entery\test.cpp(148,28): error C2572: "Mul": 重定义默认参数 : 参数 2
1> D:\CODE\NewCPP\Entery\Entery\test.cpp(141,5):
1> 参见 "Mul" 的声明
1>已完成生成项目 "Entery.vcxproj" 的操作 - 失败。
===== 生成: 0 成功, 1 失败, 0 最新, 0 已跳过 =====
```

将定义的缺省值去掉即可成功编译

```
1>Entery.vcxproj -> D:\CODE\NewCPP\Entery\x64\Debug\Entery.exe
1>已完成生成项目 "Entery.vcxproj" 的操作。
===== 生成: 1 成功, 0 失败, 0 最新, 0 已跳过 =====
===== 生成 于 8:55 完成, 耗时 03.115 秒 =====
```



## 4.缺省参数

C++ 中允许有多个同名函数存在

在同一作用域中,函数名相同,形参不同(类型、数量、顺序)与函数名无关

```
// 函数重载: 一词多义 C++中允许有多个同名函数存在
// 在同一作用域中 函数名相同 参数不同(类型、数量、顺序)与函数名无关

// 参数类型不同
void Swap(int* a, int* b)
{
}
```

```

    int tmp = *a;
    *a = *b;
    *b = tmp;
}
void Swap(double* x, double* y)
{
    double tmp = *x;
    *x = *y;
    *y = tmp;
}

// 参数数量不同
void Print(int a)
{
    cout << a << endl;
}
void Print(int x, int y)
{
    cout << x << " " << y << endl;
}
// 参数顺序不同
void f(int a, double b)
{
    cout<<"f(int a, double b)" << endl;
}

void f(double b, int a)
{
    cout << "f(double b, int a)" << endl;
}

///// 返回值不能作为判断条件
///// C2556 只是在返回类型上不同
//void fun(){}
//int fun(){}

///// 这两个函数会产生调用歧义 编译器也不知道调用哪个函数
///// 将缺省值去掉后, 给出具体实参即可顺利编译
//void f1()
//{

```

```

// cout << "f()" << endl;
//}
//void f1(int a = 10)
//{
// cout << "f(int a)" << endl;
//}

int main()
{
    int a = 10, b = 20;
    double x = 1.2, y = 2.4;
    cout << a << " " << b << endl;
    cout << x << " " << y << endl;
    Swap(&a, &b);
    Swap(&x, &y);
    cout << a << " " << b << endl;
    cout << x << " " << y << endl;

    Print(2);
    Print(1,2);
    f(1,1.2);
    f(2.1, 1);

    //// C2668 对重载函数的调用不明确
    //f1();
    //f1(20);
    return 0;
}

```

**重点：**为什么 C语言 不支持函数重载，而 C++ 支持

C语言不支持函数重载，主要是因为它的编译器在链接阶段使用**函数名本身**来唯一标识一个函数，也就是说函数名在C语言中必须是唯一的；而C++支持函数重载，是因为它采用了**名称修饰 (Name Mangling)**的机制（把参数类型带到函数名字中去），会在编译时将函数名、参数类型等信息编码到最终的符号名中，生成一个唯一的函数标识，这样就能区分参数不同但函数名相同的多个函数

```

// 演示C++函数名修饰规则
void print(int);
void print(double);
int main()

```

```

{
    // error LNK2019: 无法解析的外部符号 "void __cdecl print(int)" (?
print@@YAXH@Z), 函数 main 中引用了该符号
    print(42);
    // error LNK2019: 无法解析的外部符号 "void __cdecl print(double)" (?
print@@YAXN@Z), 函数 main 中引用了该符号
    print(2.1);
    return 0;
}

```

```

230 // 演示C++函数名修饰规则
231 void print(int);
232 void print(double);
233 int main()
234 {
235     // error LNK2019: 无法解析的外部符号 "void __cdecl print(int)" (?print@@YAXH@Z), 函数 main 中引用了该符号
236     print(42);
237     // error LNK2019: 无法解析的外部符号 "void __cdecl print(double)" (?print@@YAXN@Z), 函数 main 中引用了该符号
238     print(2.1);
239     return 0;
240 }
241

```

输出

显示输出来源(S): 生成

1>test.obj : error LNK2019 无法解析的外部符号 "void \_\_cdecl print(int)" (?print@@YAXH@Z), 函数 main 中引用了该符号  
1>test.obj : error LNK2019 无法解析的外部符号 "void \_\_cdecl print(double)" (?print@@YAXN@Z), 函数 main 中引用了该符号  
1>D:\CODE\NewCPP\Entry\x64\Debug\Entry.exe : fatal error LNK1120: 2 个无法解析的外部符号  
已生成项目 "Entry - x64Debug" 的操作 = 失败

注意观察：

`int` 类型： `?print@@YAXH@Z`

`double`： `?print@@YAXN@Z`

二者只有 `H` 和 `N` 的区别



## 5. 引用&

引用不是新定义一个变量，而是给已存在变量取了一个别名，编译器不会为引用变量开辟内存空间，它和它引用的变量共用同一块内存空间

比如说一个人，有自己的大名和小名，用哪个称呼都指代这个人，所以不会开辟新的内存空间

类型& 引用名 = 引用对象

使用引用时有如下规则

- 引用在定义时必须初始化

- 一旦确认了一个引用实体，就不能再引用其他实体了
- 一个变量可以有多个引用

```
// 引用&
using namespace std;
int main()
{
    int a = 1;
    // 引用在定义时必须初始化  error C2530: “nq”: 必须初始化引用
    //int& nq;
    // 一个变量可以有多个引用
    int& na = a;
    int& nb = a;
    int& nc = na;

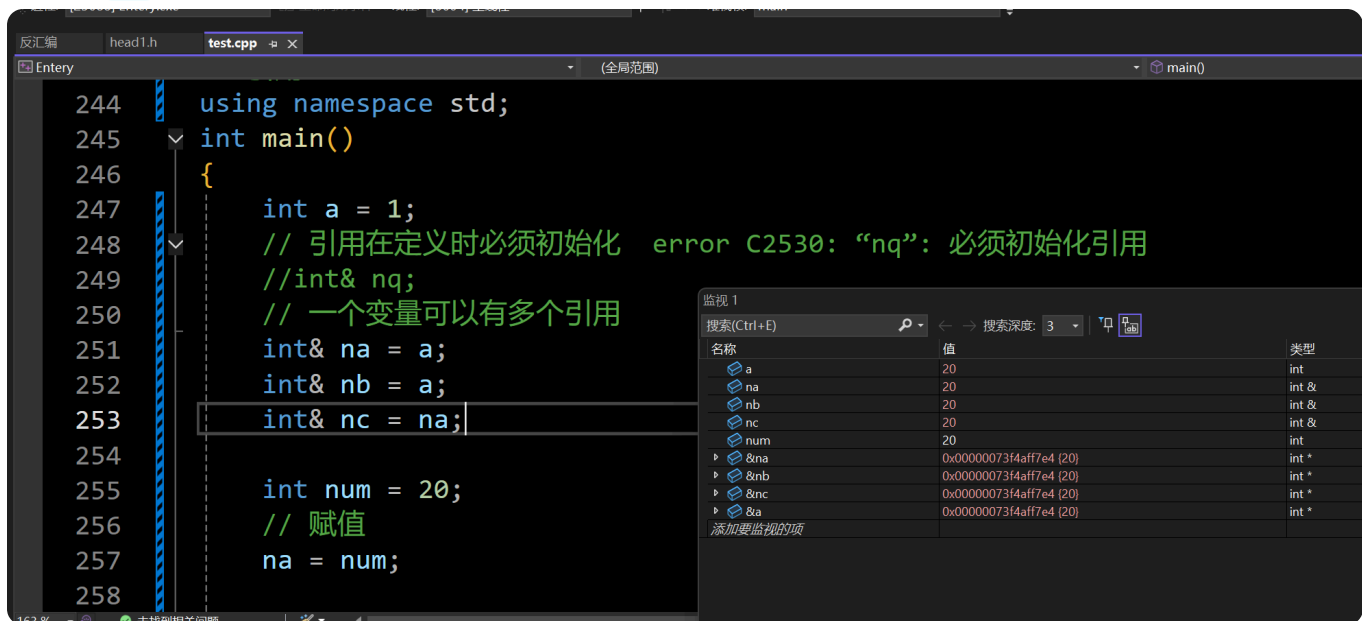
    int num = 20;
    // 赋值
    na = num;

    // 一旦确认了一个引用实体后，就不能再引用其他实体
    int b = 2;
    int& np = b;
    np = na; // 这里是赋值，而不是改变引用指向

    return 0;
}
```



这里 `a` `na` `nb` `nc` 是同一个值  
而改变 `na` 的值会发生什么？



全都变成20

我们再看看地址，所有值的地址都是一样的，也证明了在语法上引用不开辟新的空间，而是变量的别名

## 5.1. 引用的使用

- 引用在事件中主要用于引用传参和引用做返回值中减少拷贝提高效率 and 改变引用对象时同时改变被引用对象。
- 引用传参和指针传参功能类似，而引用传参相对更方便一些

```
// const修饰引用
using namespace std;
```

```

int main()
{
    const int a = 10;
    // 不可以 a只读 ra可读可写 权限放大
    // int& ra = a;

    // 可以 a和pa都是只读 权限平移
    const int& pa = a;

    // 可以 b可读可写 rb只读 权限缩小
    int b = 10;
    const int& rb = b;

}

```

而**表达式运算**（必须要有一个值来存结果）和**类型转换**会产生临时变量，临时变量具有常性

```

// 类型转换和表达式运算会产生临时变量 临时变量具有常性
using namespace std;
int main()
{
    int a = 20;
    int c = 10;
    double b = a;
    // E0434 非常量限定
    // int& rb = b;
    const int& rb = b;

    // E0461 非常量引用的初始值必须为左值
    // int& plus = a + c;
    const int& plus = a + c; // a c固定，而a+c必须有一个临时变量来存储结果

    return 0;
}

```

```
int a = 20;  
double b = a;  
// E0434 非常量限定  
// int& rb = b;  
const int& rb = b;
```



```
// E0461 非常量引用的初始值必须为左值
```

```
// int& plus = a + c;  
const int& plus = a + c;
```

```
}
```

临时变量

## 5.2. 指针和引用的区别

- 在语法上，引用不开辟新的内存空间，而指针开辟新的内存空间，但是在底层，二者没有区别



```

    int numa = 10;
00007FF73B81528E  mov             dword ptr [numa],0Ah
    int* ptr = &numa;
00007FF73B815295  lea             rax,[numa]
00007FF73B815299  mov             qword ptr [ptr],rax

    int& rnuma = numa;
00007FF73B81529D  lea             rax,[numa]
00007FF73B8152A1  mov             qword ptr [rnuma],rax
    return 0;
00007FF73B8152A5  xor             eax,eax

```

转到汇编，我们发现二者都是开辟了新的空间，将值存在 `rax` 中


- 引用在定义时必须初始化，而指针没有强制要求
- 引用在引用了一个确定对象之后，就不能改变指向了，而指针可以
- 引用可以直接访问对象，指针必须解引用之后才能访问
- `sizeof(引用类型)` 大小就是引用类型大小，`sizeof(指针类型)`，大小取决于环境，`64位` 不论类型，指针大小均为 `8`，而 `32位` 不论类型，指针大小均为 `4`；引用对象`++`，是该对象值`+1`，指针对象`++`，是该对象值`+`（类型大小）

```

using namespace std;
int main()
{
    int numa = 10;
    int* ptr = &numa;
    cout << sizeof(int*) << endl;
    cout << *ptr << endl;
    ptr++;
    cout << *ptr << endl;

    int& rnuma = numa;
    cout << sizeof(rnuma) << endl;
    cout << rnuma << endl;
    rnuma++;
    cout << rnuma << endl;
    return 0;
}

```



Microsoft Visual Studio 调试器

```

4
10
-858993460
4
10
11
D:\CODE\NewCPP\Entery\Debug
已退出，代码为 0 (0x0)。
按任意键关闭此窗口

```

因为不在数组中，往后挪动四位，指向了随机值

- 一般来说没有空引用的概念，而有空指针

```
321  int* ptr = nullptr;
322  int& rptr = *ptr;
323  /*cout << rptr;*/
324
325  return 0;
326 }
```

322行在底层是一个指针存一个空指针的地址，不报错

输出

显示输出来源(S): 生成

```
l>test.cpp
l>Entry.vcxproj -> D:\CODE\NewCPP\Entry\Debug\Entry.exe
===== 生成: 1 成功, 0 失败, 0 最新, 0 已跳过 =====
===== 生成 于 11:20 完成, 耗时 02.508 秒 =====
```

```
int* ptr = nullptr;
int& rptr = *ptr;
cout << rptr << endl;

return 0;
```

解引用时才会崩溃

Microsoft Visual Studio 调试器

D:\CODE\NewCPP\Entry\Debug\Entry.exe (进程 3344)已退出, 代码为 -1073741819 (0xc0000005)。  
按任意键关闭此窗口...

```
int* ptr = nullptr;
int& rptr = *ptr;
/*cout << rptr << endl;*/

return 0;
```

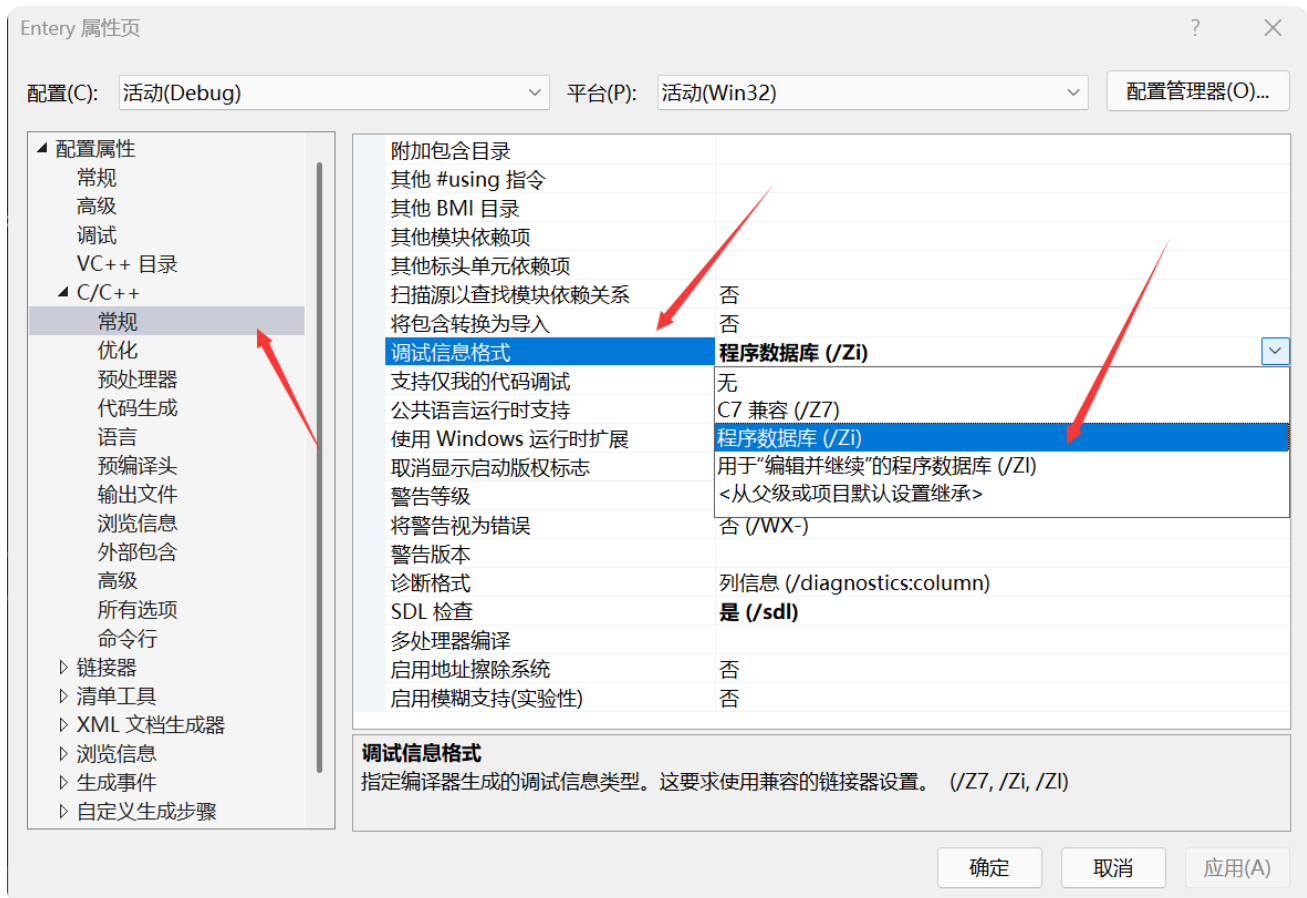
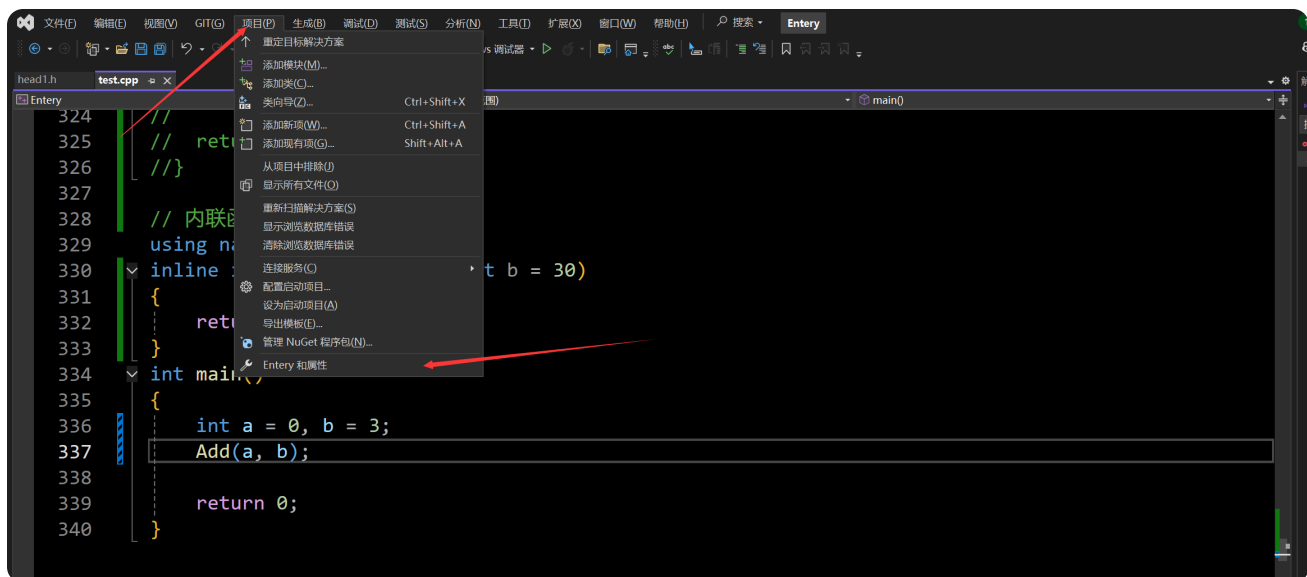
程序正常运行

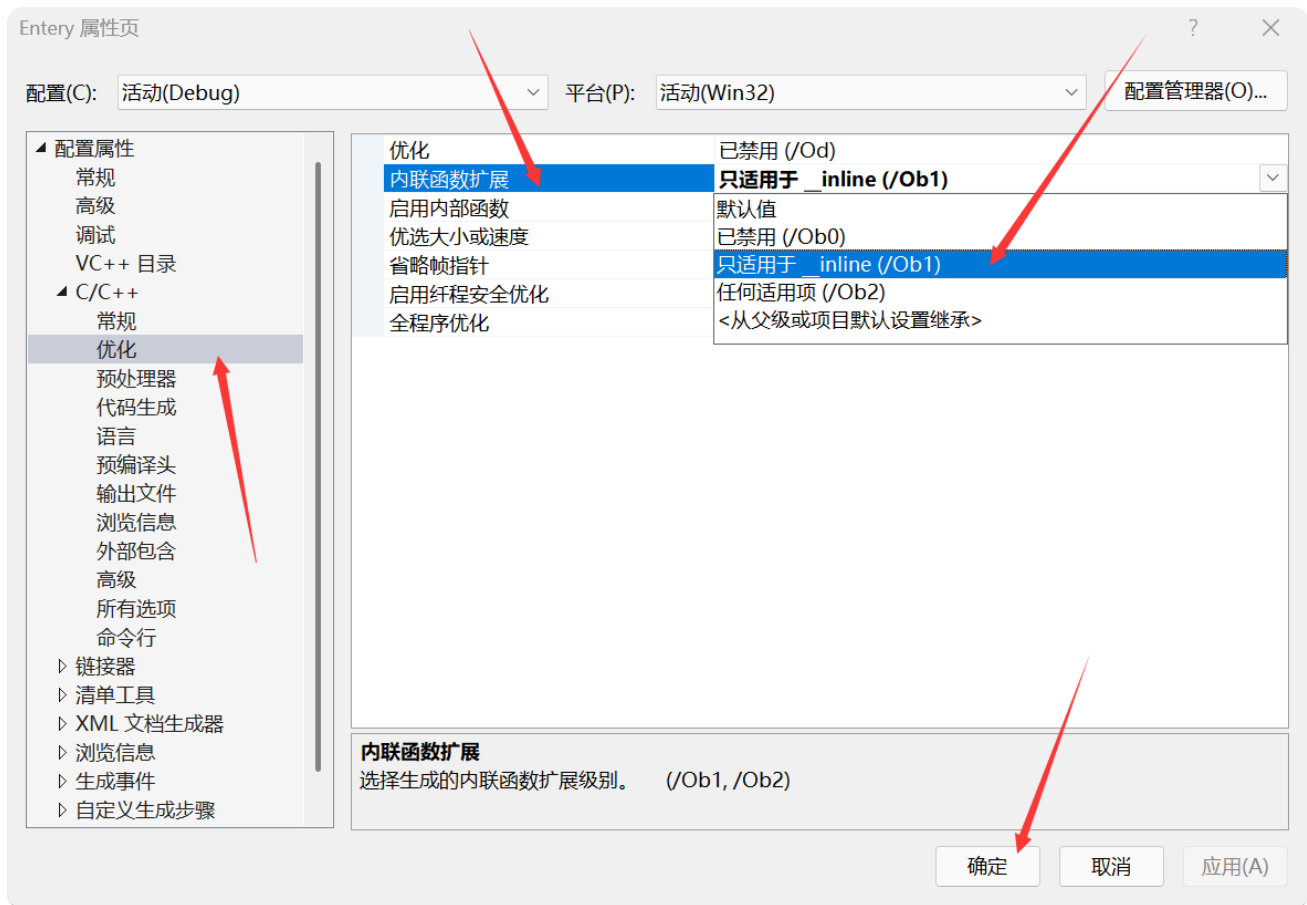
Microsoft Visual Studio 调试器

D:\CODE\NewCPP\Entry\Debug\Entry.exe (进程 13060)已退出, 代码为 0 (0x0)。  
按任意键关闭此窗口...

## 6. 内联函数

- 用 `inline` 修饰的函数叫内联函数，编译时会在调用的地方直接展开内联函数，并不会像传统函数一样建立栈帧，这样可以提高效率
- `inline` 对于编译器只是一个建议，取决于不同编译器。`inline` 适合频繁调用的代码量小的函数，对于递归函数和代码量大的函数，加上 `inline` 也会被编译器忽略
- `inline` 不建议声明和定义分离到两个文件中，分离会导致链接错误。`inline` 被展开，就没有函数地址了，链接时找函数会出现报错  
想要实现内联函数，需要调整





```
// 内联函数
using namespace std;
inline int Add(int a = 10, int b = 30)
{
    return a + b;
}
int main()
{
    // 可以通过汇编观察程序是否展开
    // 有call Add语句就是没有展开，没有就是展开了
    int a = 0, b = 3;
    Add(a, b);

    return 0;
}
```

```
int a = 0, b = 3;
003314FE  mov     dword ptr [a],0
00331505  mov     dword ptr [b],3
Add(a, b);
```

没有call 函数调用

## nullptr

`NULL` 实际是一个宏，在传统的 C 头文件 (`stddef.h`) 中，可以看到如下代码：

```
#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

在 C++ 中，`nullptr` 是空指针，可以理解为 `(void*)0` 而 `NULL` 是个宏，代表 `0`