C++

笔记本： 我的第一个笔记本
创建时间： 2020/9/14 19:01 **更新时间：** 2020/9/29 21:20
作者： 1272209351@qq.com

C++

C语言之父-----里奇

语言（C\C++）、数据结构、操作系统（linux）、网络、数据库、项目
c++98、c++11

C++98 里边有63个关键字

asm	do	if	return	try	continue
auto	double	inline	short	typedef	for
bool	dynamic_cast	int	signed	typeid	public
break	else	long	sizeof	typename	throw
case	enum	mutable	static	union	wchar_t
catch	explicit	namespace	static_cast	unsigned	default
char	export	new	struct	using	friend
class	extern	operator	switch	virtual	register
const	false	private	template	void	true
const_cast	float	protected	this	volatile	while
delete	goto	reinterpret_cast			

命名空间----namespace

一、命名空间的三种方式：

1、（1）、常用方式；（2)、命名空间可以嵌套；
（3）同一个工程中可以定义多个名字相同的命名空间

```cpp
namespace N1
{
    int a = 10;
    int b = 20;

    int Add(int left, int right)
    {
        return left + right;
    }
}
```

```cpp
// 2. 命名空间可以嵌套
namespace N2
{
    int a = 10;
    int b = 20;

    int Sub(int left, int right)
    {
        return left - right;
    }

    namespace N3
    {
        int c = 10;
        int d = 20;

        int Mul(int left, int right)
        {
            return left*right;
        }
    }
```

```cpp
// 3. 在同一工程中，可以定义多个名字相同的命名空间
// 不会冲突
// 编译器会将多个相同名称的命名空间合并成一个
namespace N1
{
    int Div(int left, int right)
    {
        return left / right;
    }
}
```

相同作用域中不能出现相同的变量名字
相同名称的多个命名空间中：也不能出现相同的名字 因为编译器会将多个相同名称的命名空间最终合并成一个

```cpp
namespace N
{
    int a = 10;

    //嵌套的方式----该种方式不会冲突
    // 相当于外层N命名空间中又包含了一个N的命名空间
    namespace N
    {
        int b = 10;
        int a = 10;
    }
}


int main()
{

    return 0;
}
```

2、命名空间的访问方式：

：： 作用域运算符

```
namespace N
{
    int a = 10;
    int b = 20;

    int Add(int left, int right)
    {
        return left + right;
    }
}

int a = 20;

int main()
{
    int a = 30;

    printf("%d\n", a);
    return 0;
}
```

```
int main()
{
    int a = 30;

    // 就近原则
    printf("%d\n", a);

    // 如果访问全局作用域中的a
    // ::作用域运算符
    // ::a  明确说明要访问全局作用域中的a
    printf("%d\n", ::a);
    return 0;
}
```

```
// 访问N命名空间中的a
printf("%d\n", N::a);
```

```
// 该场景：对N命名命名空间中某些成员访问的非常频繁

using N::a;

int main()
{
    // 访问N命名空间中的a
    printf("%d\n", N::a);
    printf("%d\n", N::a);
    printf("%d\n", N::a);
    printf("%d\n", N::a);
    printf("%d\n", N::a);
    printf("%d\n", N::a);
    printf("%d\n", N::a);

    // 为了写代码简单，想要直接访问N命名空间中的a
    printf("%d\n", a);
    return 0;
}
```

using N : : a
该语句加上之后相当于将N 命名空间之内的 a 当成当前文件的一个全局变量来使用

```
// 该场景：对N命名命名空间中某些成员访问的非常频繁
// 优点：写代码简单了
// 缺点：如果该文件中有相同名称的变量或者函数，就会产生冲突
// 如果产生冲突，怎么办？----按照方式1使用即可

// 该条语句加上之后，相当于将N命名空间中的a当成当前文件的一个全局变量来使用
using N::a;


// 如果该文件中也有一个a，必然会产生冲突，只能按照方式1来进行使用
//int a = 10;

int main()
{
    int a = 10;
```

C++中的输入和输出：
C语言的输入输出方式在C++中依旧可以使用-----兼容

```
#pragma  warning (disable:4996)
#include<iostream>
using namespace std ;

cin >> endl;
cout << endl;
```

3、C语言中标准输入输出：
scanf/getchar/gets.....
printf/putchar/puts.....

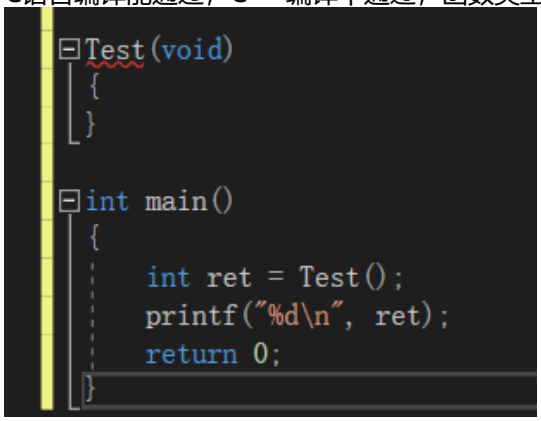
C++中：
cin>>endl
cout<<endl

注意：std 命名空间是C++语言提供的
cin 和 cout 包含在std 命令空间当中，但不是说std 命名空间中只包含了 cin 和 cout

4、C / C++ 的区别

```
// 没有返回值，也没有参数
// 在C语言中：以下代码可以通过编译
// 在C++语言中：以下代码不能通过编译
// C++编译器比C语言编译器语法检测更加严格
void TestFunc()
{

}

int main()
{
    TestFunc();
    TestFunc(10);
    TestFunc(10, 20);
    TestFunc(10,20,30);
    return 0;
```

C语言编译能通过，C++编译不通过，函数类型缺失



```
Test(void)
{
}

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

C和C++关于函数方面的区别:
（1）函数返回值类型
（2）函数参数返回值类型
结论：C++编译器比C语言对函数参数检测更加严格
（3）c++定义函数可以带参数值

```
void TestFunc(int a, int b)
{
    printf("%d %d", a, b);
}

int main()
{
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);
    TestFunc(10, 20);

    // 上述调用中，每次调用传递的都是同一组实参
    // 不想传递，但是让该函数执行起来后，还可以拿到10,20
    TestFunc();
    return 0;
}
```

可以在函数形参定义的同时进行赋值（C++能正常运行）

缺省参数：在定义函数同时给参数带上默认值

```
// 缺省参数：在定义函数时，可以给函数的参数待上一些默认值
// 在调用该函数时，如果没有指定实参则采用该默认值，否则使用指定的实参。
void TestFunc(int a=10, int b=20)
{
    printf("%d %d\n", a, b);
}
```

形参 a、b 都带有默认值，用户调用函数时，如果没有传递实参则a 和 b 使用默认值，如果用户传递了实参就使用传递的实参

缺省参数的分类：

（1）全缺省参数：每一个参数都带有默认值

```
void TestFunc(int a=1, int b=2, int c=3)
{
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
}

int main()
{
    TestFunc();             // 1 2 3
    TestFunc(10);           // 10 2 3?   1  2  10?
    TestFunc(10, 20);
    TestFunc(10, 20, 30);   // 10  20  30
    return 0;
}
```

```
a = 1
b = 2
c = 3
a = 10
b = 2
c = 3
a = 10
b = 20
c = 3
a = 10
b = 20
c = 30
```

（2）半缺省参数：部分参数带有默认值

```
void TestFunc(int a, int b = 2, int c = 3)
{
    cout << a << " " << b << " " << c << endl;
}
```

```cpp
// 2. 半缺省参数：部分参数带有默认值
// a      b      c
// 0      1      1      代码可以通过编译
// 1      0      1      代码编译失败
// 1      1      0      代码编译失败
// 0      0      1      可以通过编译
void TestFunc(int a, int b, int c=1)
{
    cout << a << " " << b << " " << c << endl;
}

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

结论：缺省参数只能从右往左依次给出，不能间隔着赋值

缺省参数可以在声明位置给出，也可以在定义位置给出，但是不能两个位置同时给出

5、函数重载

定义：在相同作用域中，函数名字相同 参数列表不同（个数、类型、类型次序）

C语言不支持函数重载

参数列表不同的几种形式：
参数个数不同；参数列表不同；参数类型次序不同

```cpp
// 函数重载

int Add(int left, int right)
{
    return left + right;
}

double Add(double left, double right)
{
    return left + right;
}


int main()
{
    Add(10, 20);
```

```cpp
// 1. 参数个数不同
// 2. 参数列表不同
void TestFunc()
{}

void TestFunc(int a)
{}

void TestFunc(char c)
{}

void TestFunc(int a, char b)
{}

void TestFunc(char a, int b)
{}
```

注意：如果两个函数仅仅只是返回值类型不同则不能构成函数重载

函数重载一定是在编译阶段具体来确定应该调用哪一个函数

函数重载的调用原理：

（1）编译器在编译阶段，会对函数实参类型进行推演，根据推演的结果找类型匹配的函数进行调用；

（2）如果有类型完全匹配的函数则直接进行调用；

（3）如果没有类型完全匹配的函数则会进行隐式类型转换，如果隐式类型转换后有对应函数则进行直接调用，如果没有对应的函数则会报错（或者说转换之后具有二义性------报错）

```cpp
int main()
{
    // 单纯从调用位置来看---只看到了一个Add
    Add(10, 20); // int, int--->

    Add(1.2, 3.4);// double, double--->Add(double, double)

    Add('1', '2'); // char, char--->Add(char,char)--->char和int之间可以进行隐
    return 0;
}
```

如果未定义char类型，会调用int 类型----隐式转换

```
// 编译阶段，对形参类型进行推演：int, double
// Add(int, double)-->该函数没有找到
// 发现：int和double之间可以进行隐式类型转换  int--->double  double--->int
// Add(int, int) || Add(double, double) 转换之后发现有两个函数都可以
// 编译器不知道到底应该调用哪一个方法了
// 编译报错
Add(1, 2.2);
```

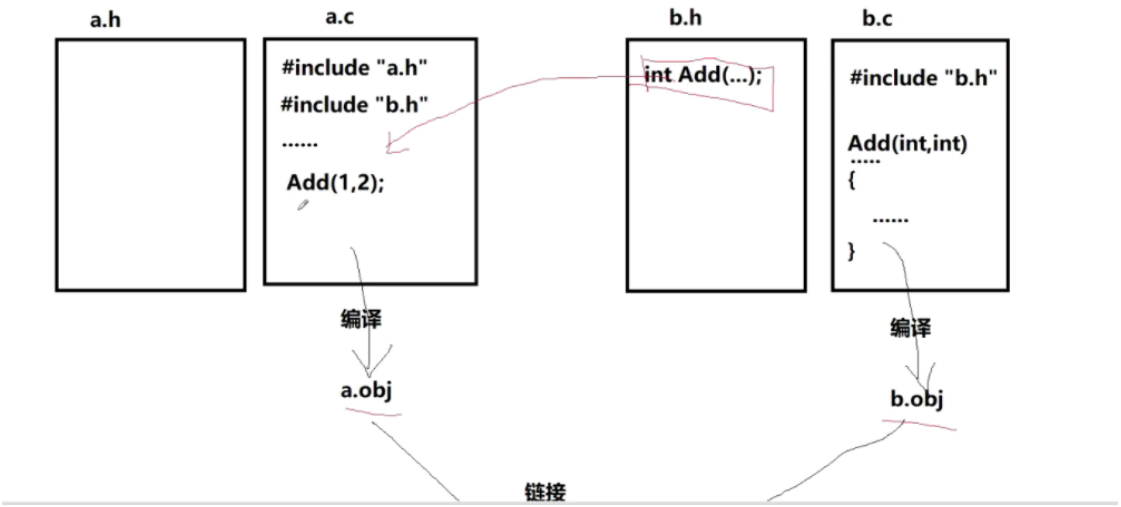
```
// vs是一个集成开发环境，当点击生成或者重新生成时：预处理--->编译-->汇编--->链接--->生成可执行程序
int Add(int left, int right);
//{
//  return left + right;
//}


//double Add(double left, double right)
//{
```

```
1>  20200917.c
1>20200917.obj : error LNK2019: 无法解析的外部符号 _Add，该符号在函数 _main 中被引用
1>D:\WorkStations\Gitee\cppLesson\实习冲刺班\ConsoleApplication1\Debug\ConsoleApplication1.exe : fatal error
========== 全部重新生成:  成功 0 个，失败 1 个，跳过 0 个 ==========
```

连接错误

（链接错误）

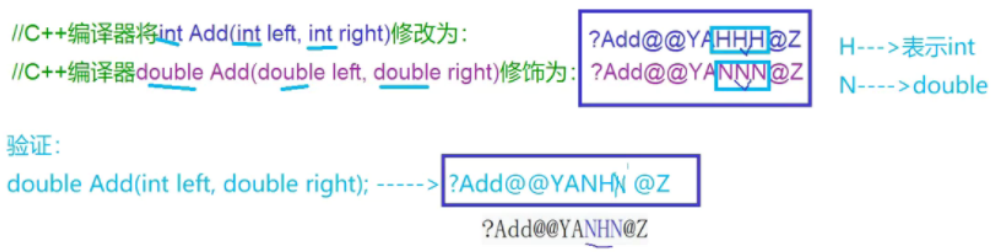
预处理-----编译--------汇编-------链接---------生成可执行程序



ctrl + f7 编译

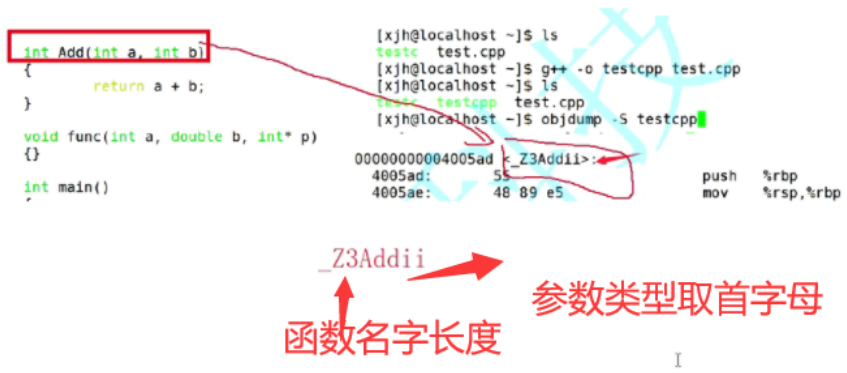
6、
在C语言中，C语言编译器在编译时，对函数名字修饰规则：在函数名前加了一个”__“
int Add (int left ,int right); ---------修改成 _Add

C++编译器在编译函数时，会对函数名字进行修改，将参数类型添加到最终的名字中；在代码层面函数名字一样但是在编译完成后在底层使用的名字不一样



HHH：第一个H 代表函数返回值类型为int
　　第二、三个H 代表函数参数类型int

linux 环境下g++ 编译器下函数命名规则:



在C++文件中可以用 extern "C" 来修饰一个函数，可以表示C 语言定义函数

7、引用类型的变量：
引用类型必须与实体类型是同种类型的；
引用类型变量在定义时候必须初始化-----否则编译器就不知道ra 到底是谁的别名；
一个变量可以有多个引用；
引用一旦引用一个实体后在不能引用其他的实体；

```
int a = 10;
int &ra = a;
```

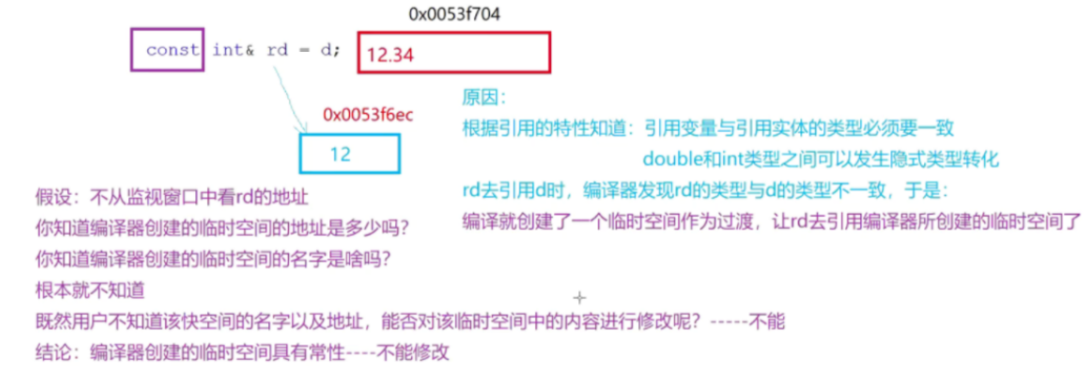
例题1:

```
int main()
{
    int a = 10;
    const int& ra = a;

    a = 100;


    // 思考下?
    double d = 12.34;
    const int& rd = d;  // rd已经是d的别名了--rd是int类型的，应该rd将d中整形部分拿到了

    d = 34.56;

    cout << rd << endl;
```

a=100
ra=100

d=34.56
rd=12

例题2:



引用的应用场景：
引用作用一：代码书写更加简便
引用作用二：引用类型作为函数形参---基本上可以取代C语言中的一级指针

引用作用三：作为函数的返回值类型

```
// 在C语言中，写一个函数，专门用来交换int*类型的指针，Swap函数该如何实现?
// 在C++语言中，写一个函数，专门用来交换int*类型的指针，Swap函数该如何实现?
```

在C++中传参：一级指针传参---->引用
二级指针---->一级指针的引用

```
int& Add(int left, int right)
{
    int ret = left + right;
    return ret;
}
int main()
{
    int& result = Add(1,2);
    Add(3, 4);
    Add(5, 6);
    return 0;
}
```

在main 函数中，后续代码执行时并没有对result 进行修改，但是在Add (3,4)调用结束后，result ---- > 7；Add(5,6) 调用结束后，result ---> 11
result 实际引用的是 Add 函数体中的 ret 局部变量，当 Add 函数调用结束之后，ret 局部变量被释放，result 实际引用的就是一块非法空间

函数体中的局部变量只在函数体内部有效，当函数运行结束之后函数体中的局部变量就被销毁了

以引用方式作为函数返回值时：
一定不能返回函数栈上空间------典型：局部空间；
如果在外部以引用方式来接收函数返回值，则引用的地址是非法空间

每一个函数在运行时都需要有自己独立的一块栈内存空间，该块栈内存空间称为该函数调用时刻所用到的栈帧
该块内存空间随函数调用而申请，随函数的结束而回收

函数栈帧中存储：函数在运行过程中的一些局部变量

esp ebp 两个寄存器，来标记栈帧的栈顶和栈底

_cdecl ： 是C / C++ 函数默认的调用约定（函数在调用期间所做的一些约定）

在C++中，函数传参有三种方式：
传值：形参是实参的一份拷贝，传参效率低
传址：形参保存的是实参的地址，传参效率高，可以通过改变形参来改变外部的实参
传引用：形参实际上是实参的别名，理论传参效率也比较高，可以通过形参改变外部实参---可以达到与指针类似的效果

lea汇编指令作用：取地址

```
int* pa = &a;
lea          eax,[a]
mov          dword ptr [pa],eax
*pa = 100;
mov          eax,dword ptr [pa]
mov          dword ptr [eax],64h
```

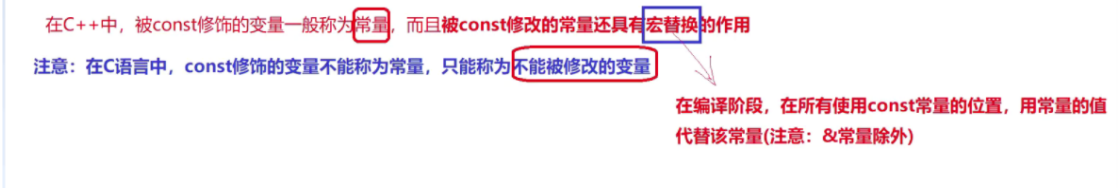
```
int& ra = a;
lea          eax,[a]
mov          dword ptr [ra],eax
ra = 100;
mov          eax,dword ptr [ra]
mov          dword ptr [eax],64h
```

（引用在底层是按照指针的方式来处理的）

int& <-----------> int*const
const int& <-------> const int*const
加 const 的原因是：引用一旦与一个实体结合之后在不能引用其他的实体了

从底层来看：引用和指针没有任何区别-----引用就是实体
在底层实现上：引用实际是有空间的----》因为引用就是指针，它里边存储的是引用实体的地址
从概念上：引用是一个别名，编译器不会给引用类型的变量开辟内存空间，引用与其引用的实体公用的是同一块内存空间

C语言中，const 修饰的一般称为变量；C++ 中，被const 修饰的称为常量，并且还有宏替换的作用



宏：不会进行类型检测，直接替换
const 修饰的常量来替换会进行类型检验

例题：

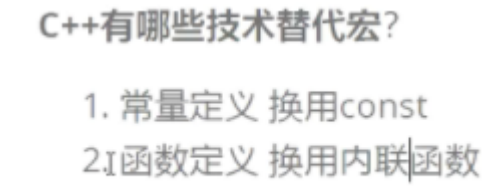
```
#define MAX(a,b) (a > b ? a : b)

int main()
{
    int a = 10;
    int b = 20;
    cout << MAX(a, ++b) << endl;
    return 0;
}
```

结果：22
a > ++b ? a : ++b ----------> b = 22

二、内联函数 inline

#pragma once 防止都文件重复定义

内联函数和宏函数的展开



int atoi(str) 字符串转整形
itoa() 整型转字符串

auto 修饰局部变量，表明该变量是一个自动变量------> 函数结束后，该变量会自动销毁（函数运行结束后，函数的栈帧就被收回了**）**

使用auto 定义变量时必须对其进行初始化，在编译阶段编译器需要根据初始化表达式来推导auto 的实际类型。因此auto 并非是一种“类型”的声明，而是一个类型声明时的“占位符”，编译器在编译器会将auto 替换为变量的实际类型。

auto 定义引用类型变量时，必须要加 &
auto 不能作为函数的参数；不能用来声明数组；

```
// 并不是所有的参数都有初始化表达式，因此编译器无法推演a的实际类型
//void TestAuto(auto a)
//{
//  printf("%d", a);
//}

int main()
{
    int array1[] = { 1, 2, 3 };


    auto array2[] = { 3, 4, 5 };

    return 0;
}
```

C++11中： nullptr --------- 用来表示空值指针

三、类和对象
C------面向过程
C++ -------- 基于面向对象（既有面向过程，也有面向对象）

面向过程和面向对象不是一门编程语言，而是一种解决问题的思路

C 语言中 struct 结构体类型，C 语言中 结构体内部不可以放函数；C++中可以

类定义的两种方式：
1. 声明和定义全部放在类中
2. 类在.h的头文件中进行声明，在.cpp的源文件中进行定义

面向对象：封装、继承、多态

类和模块----------封装特性

private 修饰的成员在类外不能直接被访问
protected 修饰的成员在类外不能直接被访问
public 修饰的成员在类外可以直接被访问

注：struct 定义的类在默认情况下的访问权限是 private

类：主要是对实体（对象）进行描述的，描述实体都有哪些属性（成员变量），实体中都有哪些方法（成员变量）
类是一个抽象概念

对象：实实在在存在的
对象模型：对象在内存中的布局方式

对象的大小：实际对象里边只存储了成员变量

对象的大小：成员变量 + 指针

实际对象中存储成员的情况：对象中只存储了成员变量----实际sizeof的结果

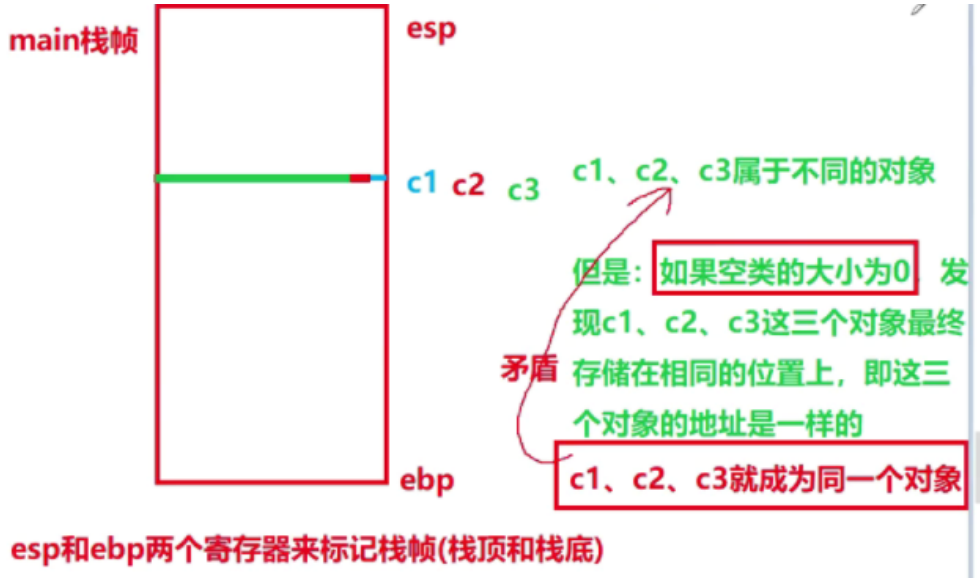
_name
_gender
_age

计算一个类的大小/对象的大小：只需将类中的成员变量加起来，注意内存对齐

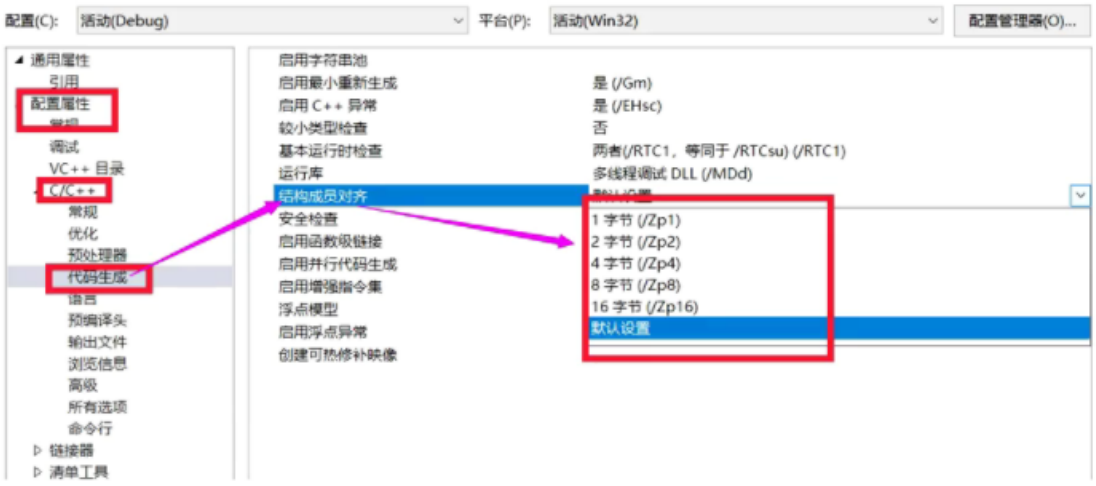
计算一个对象的大小与计算结构体类型变量的大小是一样的

成员函数存放在代码段

空类的大小为 1；目的是为了区分空类定义出来的不同对象



常见面试题：
a、什么是结构体内存对齐？结构体为什么需要内存对齐？
b、默认对其参数？如何对其来进行设置？默认对其参数可以设置为任意值吗？
#pragma pack（n）按照 n 字节对齐



offsetof 宏----------求偏移量

offsetof (A ,c)---------A结构体中c 变量的偏移量

c、如何知道结构体中某个成员变量的相对于起始位置的偏移量?
offsetof

this ： 指针，里边放的是**当前对象**的地址（成员函数执行时，调用该成员函数的对象）
不能给this 指针进行赋值

! this指针的特性

1. this指针的类型：类类型* const
2. 只能在“成员函数”的内部使用
3. **this指针本质上其实是一个成员函数的形参**，是对象调用成员函数时，将对象地址作为实参传递给this形参。所以**对象中不存储this指针。**
4. **this指针是成员函数第一个隐含的指针形参，一般情况由编译器通过ecx寄存器自动传递，不需要用户传递**