

—

C语言是面向过程的语言，数据和方法是抽离的，处于用方法去操作数据的阶段

C++是一门非完全面向对象的语言

C#是一种最新的、面向对象的编程语言，C#是由C和C++衍生出来的面向对象的编程语言
而java才是一门完全面向对象的语言

这句话怎么理解呢？

首先我们了解一种语言的诞生背景才能在合适的环境下更好的去使用它。为了克服了C语言编写大型程序时的不足，大规模的程序使C++应运而生，它是由C语言发展来的，或者可以理解成对C语言的一种完善。

面向对象：面向对象是相对于面向过程来讲的，面向对象方法，把相关的数据和方法组织为一个整体来看待，从更高的层次来进行系统建模，更贴近事物的自然运行模式

—
—

写C++程序最熟悉不过的就是标准的输入输出流iostream

VS2010之后版本的编译器是不支持iostream.h头文件的

iostream.h与<iostream>

下面的代码为什么在VC2010下面编译不过去？

```
#include <iostream.h>

int main()
{
    cout<<"Hello World."<<endl;

    return 0;
}
```

错误信息：fatal error C1083: 无法打开包括文件：“iostream.h”：No such file or directory

造成这个错误的原因在于历史原因，在过去C++98标准尚未订立的时候，C++的标准输入输出流确实是定义在这个文件里面的，这是C风格的定义方法，随着C++98标

准的确定，`iostream.h`已经被取消，至少在VC2010下面是这样的，取而代之的是我们要用`<iostream>`头文件来代替，你甚至可以认为`<iostream>`是这样定义的：

```
namespace std
{
    #include "iostream.h"
}
```

也就是说C++98标准丢弃了`iostream.h`

在C++中`cout`的用法和C中的`printf`函数是一样功能
但`cout`实际上是在`iostream`文件中定义的一个全局对象

三

C++函数的重载（C语言中函数是无法重载的）

函数的重载两个因素：

- ①参数列表的个数
- ②参数类型（参数个数相同的情况下亦可）

特别需要注意的是只靠返回值是不行的

以及参数设默认值也是不能重载的

函数重载在不同的编译器中对函数重命名的方式是不同的
在VS编译器中如下：

```

1  #include <iostream>
2  using namespace std;
3
4  int Add(int a, int b)
5  {
6      return a + b;
7  }
8
9  double Add(double a, double b)
10 {
11     return a + b;
12 }
13
14
15 double Add(double a, int b)
16 {
17     return a + b;
18 }
19
20 double Add(int a, double b)
21 {
22     return a + b;
23 }
24
25 void Add(void)
26 {
27     ;
28 }
29

```

```

?Add@@YAHHH@Z
?Add@@YANH@Z
?Add@@YANNH@Z
?Add@@YANN@Z
?Add@@YAXXZ

```

? + 函数名 + @ @ + 命名空间 + 参数加返回值类型 + @
Y(Z)

四

C++程序中给函数前加上 extern "C"

extern "C" 包含双重含义，从字面上即可得到：首先，被它修饰的目标是“extern”的；其次，被它修饰的目标是“C”的。

被extern "C"限定的函数或变量是extern类型的；

extern关键字

extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。

通常，在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字extern声明。例如，如果模块B欲引用该模块A中定义的全局变量和函数时只需包含模块A的头文件即可。这样，模块B中调用模块A中的函数时，在编译阶段，模块B虽然找不到该函数，但是并不会报错；它会在链接阶段中从模块A编译生成的目标代码中找到此函数。

与extern对应的关键字是static，被它修饰的全局变量和函数只能在本模块中使用。因此，一个函数或变量只可能被本模块使用时，其不可能被extern “C” 修饰。

被extern “C”修饰的变量和函数是按照C语言方式编译和链接的

首先看看C++中对类似C的函数是怎样编译的。

作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。函数被C++编译后在符号库中的名字与C语言的不同。例如，假设某个函数的原型为：

```
void foo( int x, int y );
```

该函数被C编译器编译后在符号库中的名字为foo，而C++编译器则会产生像foo_int_int之类的名字（不同的编译器可能生成的名字不同，但是都采用了相同的机制，生成的新名字称为“mangled name”）。

** foo_int_int这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。** 例如，在C++中，函数void foo(int x, int y)与void foo(int x, float y)编译生成的符号是不相同的，后者为foo_int_float。

C/C++在linux下主函数返回void是通不过编译的

五

变量的引用（表象理解就是起了别名）

在语法概念上引用就是一个别名，没有独立空间，和其引用实体共用同一块空间底层实现上实际是有空间的，因为引用是按照指针方式来实现的。

函数中局部变量千万不得用引用返回：（如下代码，会有很大的BUG）

```
int value;
int& fun(int a, int b)
{
```

```

        value = a + b;
        return value;    //归还了局部变量value空间，但是VS编译器为了效率没有将原
    来临时变量value的空间置成随机值拿到的结果是对了，但是换个编译器可能结果就会出
    错，作为一个严格的人是不允许这样做的不被允许的事的，那么要成为优秀的程序员就更不
    能这样做了
}
int main()
{
    int &val = fun(1,2);
    fun(10,20);
    cout<<"val = "<<val<<endl; //????
    return 0;
}

```

实际上引用的实现就是指针的实现：

我们来看下引用和指针的汇编代码对比：

<code>int a = 10;</code>	<code>int a = 10;</code>
<code>mov dword ptr [a], 0Ah</code>	<code>mov dword ptr [a], 0Ah</code>
<code>int& ra = a;</code>	<code>int* pa = &a;</code>
<code>lea eax, [a]</code>	<code>lea eax, [a]</code>
<code>mov dword ptr [ra], eax</code>	<code>mov dword ptr [pa], eax</code>
<code>ra = 20;</code>	<code>*pa = 20;</code>
<code>mov eax, dword ptr [ra]</code>	<code>mov eax, dword ptr [pa]</code>
<code>mov dword ptr [eax], 14h</code>	<code>mov dword ptr [eax], 14h</code>

C++primer中对 对象的定义：对象是指一块能存储数据并具有某种类型的内存空间

一个对象a，它有值和地址&a，运行程序时，计算机会为该对象分配存储空间，来存储该对象的值，我们通过该对象的地址，来访问存储空间中的值

指针p也是对象，它同样有地址&p和存储的值p，只不过，p存储的数据类型是数据的地址。

如果我们要以p中存储的数据为地址，来访问对象的值，则要在p前加解引用操作符“*”，即*p。

对象有常量（const）和变量之分，既然指针本身是对象，那么指针所存储的地址也有常量和变量之分，常量指针是指，指针这个对象所存储的地址是不可以改变的，而指向常量的指针的意思是，不能通过该指针来改变这个指针所指向的对象。

正确引用写法：

```
int    r;
int &b  =  r;
```

错误引用写法:

```
int    r;
int &b;
b  =  r;
```

我们可以把引用理解成变量的别名。定义一个引用的时候，程序把该引用和它的初始值绑定在一起，而不是拷贝它。计算机必须在声明r的同时就要对它初始化，并且，r一经声明，就不可以再和其它对象绑定在一起了。

实际上，你也可以把引用看做是通过一个常量指针来实现的，它只能绑定到初始化它的对象上。

六 `typeid, auto`

`typeid(变量).name`

可以知道变量的类型是什么:

```
int a = 10;
cout<<typeid(a).name()<<endl;           //string
```

`auto`

在早期C/C++中`auto`的含义是：使用`auto`修饰的变量，是具有自动存储器的局部变量，但遗憾的是从来没有

人去使用它，大家可思考下为什么？

C++11中，标准委员会赋予了`auto`全新的含义即：`auto`不再是一个存储类型指示符，而是作为一个新的类型

指示符来指示编译器，`auto`声明的变量必须由编译器在编译时期推导而得。

```

void fun(auto p)                                //错误 1      error C3533: “auto” : ①参数不
能为包含 “auto” 的类型
{}

int main()
{
    auto ar[] = {1, 2, 3, 4, 5};                //      4      IntelliSense: ②
“auto” 类型不能出现在顶级数组类型中
    int a = 10;
    auto x = &a;
    auto *y = &a; //int *y = &a;
    auto &z = a;
    cout<<typeid(x).name()<<endl;
    cout<<typeid(y).name()<<endl;
    cout<<typeid(z).name()<<endl;

    fun(a);
    return 0;
}

```

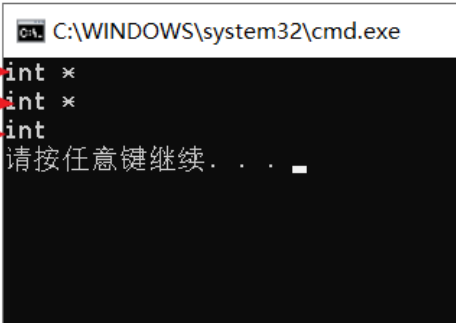
用auto声明指针类型时，用auto和auto*没有任何区别，但用auto声明引用类型时必须加&

```

int main()
{
    //auto ar[] = {1, 2, 3, 4, 5};
    int a = 10;
    auto x = &a;
    auto *y = &a; //int *y = &a;
    auto &z = a;
    cout<<typeid(x).name()<<endl;
    cout<<typeid(y).name()<<endl;
    cout<<typeid(z).name()<<endl;

    //fun(a);
    return 0;
}

```



for循环后的括号由冒号“：”分为两部分：第一部分是范围内用于迭代的变量，第二部分则表示被迭代的范围

```
void TestFor()
{
    int array[] = { 1, 2, 3, 4, 5 };
    for(auto& e : array) //引用可以直接修改数组中的元素
        e *= 2;
    for(auto e : array)
        cout << e << " ";
    return 0;
}
```

七 内联函数

以inline修饰的函数叫做内联函数，编译时C++编译器会在调用内联函数的地方展开，没有函数压栈的开销，内联函数提升程序运行的效率

函数前增加inline关键字将其改成内联函数，在编译期间编译器会用函数体替换函数的调用，简单的函数设成内联函数可以大大降低函数调用的时间（inline是一种以空间换时间的做法，省去调用函数额开销。所以代码很长或者有循环/递归的函数不适宜使用作为内联函数）inline不建议声明和定义分离，分离会导致链接错误。因为inline被展开，就没有函数地址了，链接就会找不到

不能递归

不能循环

八 空指针

C++11标准之前NULL以前就是一个宏

不同的编译器厂商对于NULL的实现可能不太相同，而且直接扩展NULL，可能会影响以前旧的程序

这两个因素共同促使C++11提供了代表一个指针空值常量:nullptr 其类型为nullptr_t

使用:nullptr不需要引头文件, nullptr是C++11作为新关键字

扩展:sizeof(nullptr) 与 sizeof((void*)0)所占的字节数相同----- sizeof(nullptr)
= 4

表示指针空值时建议最好使用nullptr

九 类中的六个默认成员函数

顺便提一下初始化和赋值的区别:

```
Test t3 = t2; //初始化
```

```
Test t4;
```

```
t4 = t3; //赋值
```

```
class Test
```

```
{
```

```
};
```

①构造函数-->1. 构造对象 2 . 初始化 3 . 类型转换

```
Test t1(10);
```

②拷贝构造函数-->对象初始化对象

```
Test t2(t1);
```

③赋值重载函数-->对象赋值

```
t3 = t2;
```

④析构函数

⑤取地址重载函数

```
Test *pt = &t;
```

```
Test* operator&()
```

```

{
    return this;
}

```

⑥取常对象地址函数

```

const Test ct;
const Test *pt1 = &ct;

```

const Test* operator&() const (常方法限定的是this指针,常方法指向的内存空间不能改变)

```

{
    return this;
}

```

+ 初始化列表

```

class Time
{
public:
    Time(int h, int m, int s)
    {
        cout<<"Time::Time()"<<endl;
        hour = h;
        minute = m;
        second = s;
    }
private:
    int hour;
    int minute;
    int second;
};

```

```

class Date
{
public:
    Date(int y, int m, int d) : t(20,44,30)
    {
        cout<<"Date::Date()"<<endl;
    }
}

```

```

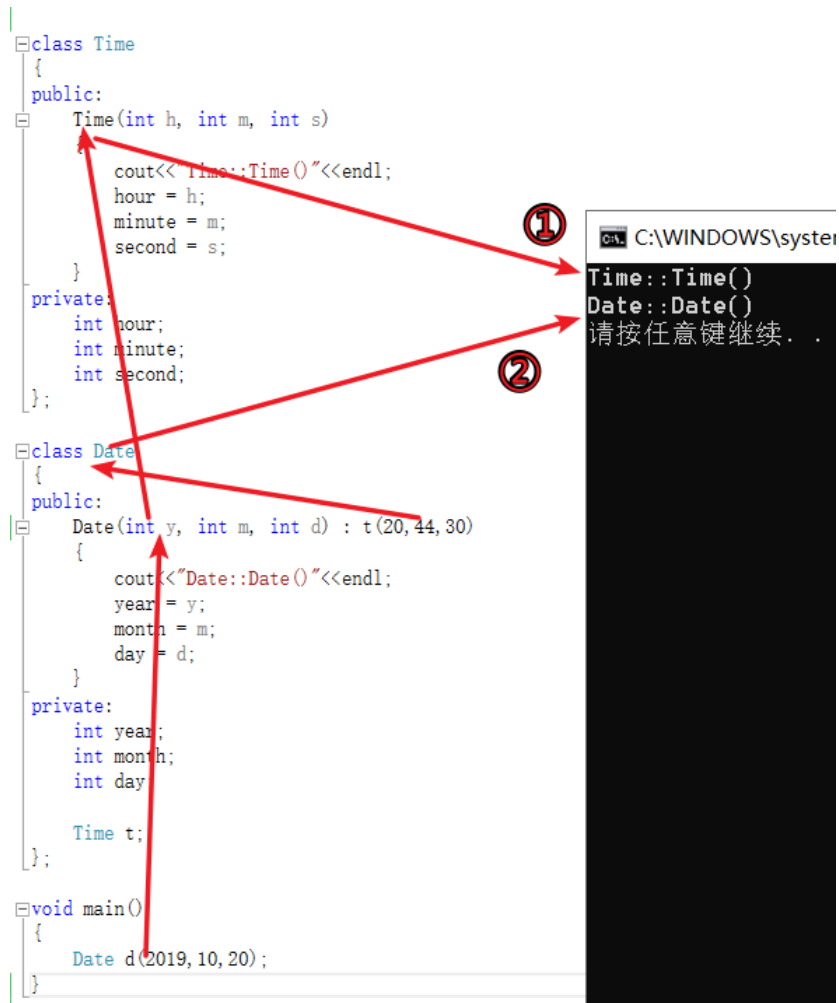
        year = y;
        month = m;
        day = d;
    }
private:
    int year;
    int month;
    int day;

    Time t;
};

void main()
{
    Date d(2019,10,20);
}

```

类的数据成员可以是其它类的对象-->构造顺序与初始化列表顺序无关



两种必须使用初始化列表的情况：

①引用的数据成员

②const成员变量

考虑到效率问题也会使用初始化列表

十一 对枚举常量作为类的数据成员的巧妙使用：

```
class SeqList
{
public:
    SeqList(int sz=DEFAULT_SEQLSIT_SIZE)
    {
        capacity = sz > DEFAULT_SEQLSIT_SIZE ? sz : DEFAULT_SEQLSIT_SIZE;
        base = (int *)malloc(sizeof(int) * capacity);
        size = 0;
    }
    ~SeqList()
    {
        free(base);
        base = NULL;
        capacity = size = 0;
    }
private:
    enum{DEFAULT_SEQLSIT_SIZE=10};
    int *base;
    int capacity;
    int size;
};

void main()
{
    SeqList mylist;
}
```

