華中科技大學

深入理解变量

许向阳

xuxy@hust.edu.cn

内容

变量 就是 对象

类型、地址、存储位置

大小、值、访问特性

生命周期、作用域

- 变量及其类型解析
- const 、static 修饰符
- 引用变量 &、&&

- 全方位理解变量的好处
- 使用 const、引用的优点

重点和难点

变量（对象）空间的分配与回收时间

static 变量的生命周期

有 const 约束的变量的 定义和访问

引用变量 &、 && 的定义和访问

变量类型的解析、转换

2.3 变量及其类型解析

- 变量如何定义？
- 变量的空间分配在何处？
- 变量的空间在何时分配？何时回收？
- 变量的空间如何初始化？
- 变量有何访问特性？（存储可变特性）
- 变量的类型？
- 变量的地址表达形式？地址类型的转换？
- 变量的使用？值类型的转换？

[存储位置特性] [存储可变特性]

类型名　变量名 [= 初始值];

2.3 变量及其类型解析

使用方式

- auto
- register
- static
- extern

使用地点

占的字节数

分配和回收空间的时间

- const
- constexpr
- volatile
- mutable

初始化与赋值

地址

变量的存储位置特性

在函数内部定义局部变量

- **auto** 默认值
- static

在函数外定义全局变量

- static 定义的文件内使用
- **extern** 默认值

堆栈段 VS 数据段

只能出现一种存储位置特性

栈与数据段

```
#include <iostream>
int g = 10;
void f1(int u,int v)
{    int  x ;
     g = u;   x = v;
     printf("value of g,x : %d %d\n", g, x);
     printf("address of g,x : %p %p\n", &g, &x);
}
void f2()
{    int p = 20;          int q = 30;          f1(p, q);
}

int main()
{    f1(1, 2);
     printf("\n f2 ...\n");
     f2();
}
```

```
value of g,x : 1 2
address of g,x : 008AA000 012FF930

 f2 ...
value of g,x : 20 30
address of g,x : 008AA000 012FF844
```

Q： 两次调用函数**f1**，**g, x** 的地址有无变化？

&g	0x008aa000 {变量的地址.exe!int g} {1}
&x	0x012ff930 {2}

&g	0x008aa000 {变量的地址.exe!int g} {20}
&x	0x012ff844 {30}

栈与数据段

```
void f1(int u,int v)
{    int  x ;
     g = u;   x = v;
     ……
}
```

```
g = u;
mov       eax,dword ptr [ebp+8]
mov       dword ptr ds:[008AA000h],eax
x = v;
mov       eax,dword ptr [ebp+0Ch]
mov       dword ptr [ebp-0Ch],eax
```

总结：在执行程序中，变量的地址表达形式是固定的。

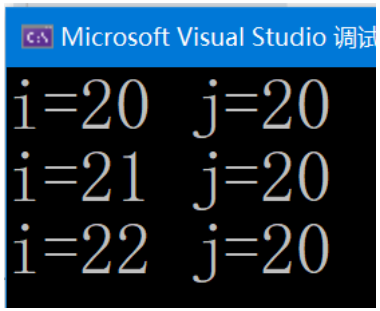
全局变量与局部变量的地址表达形式完全不同。

数据段空间是持久的，在整个程序的生命周期存在。

栈空间具有临时性，在函数开始运行时分配，函数运行结束时释放。

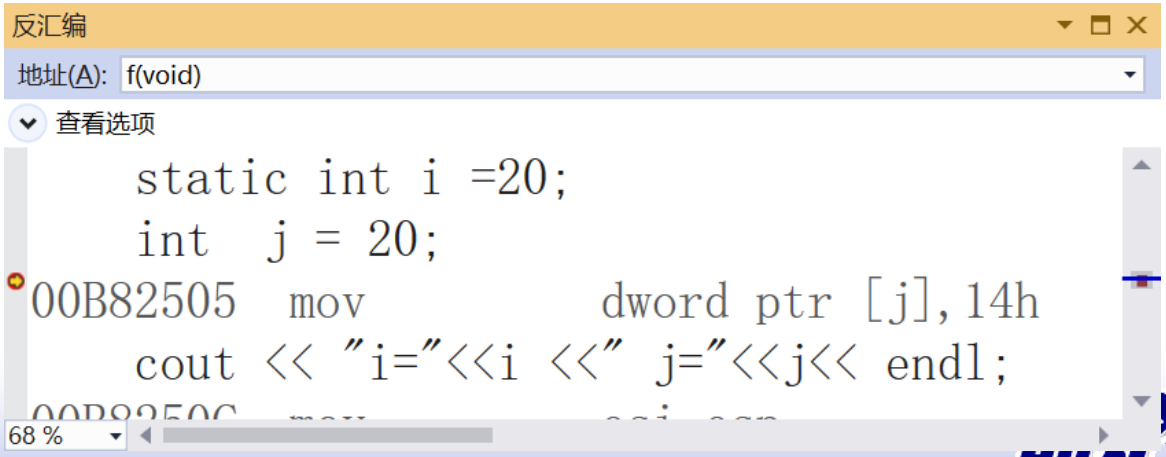
静态(static)与 非静态的局部变量

```
#include <iostream>
using namespace std;
void  f()
{    static int i =20;
     int  j = 20;
     cout << "i="<<i <<" j="<<j<< end
     i++;          j++;
}
int main()
{
     f();
     f();
     f();
     return 0;
}
```



函数内的局部静态变量 ：

- 在程序编译时，分配空间并且置了初值。
- 等同全局变量，分配空间
- 生命周期，与函数无关
- 作用域在函数内



静态(static)与 非静态的局部变量

编程：统计一个函数被调用的次数。
每次调用该函数时，显示是第几次调用。

定义全局变量？
定义静态局部变量？

```
void  f()
{    static int i =0;
     i++;
     cout <<"第 "<< i <<" 次调用"<<endl;
}
```

编程：与全局变量一样长的生命周期。
作用域只在函数内，封装性好。

2.3 变量及其类型解析

变量的存储可访问特性

- const
- constexpr
- volatile
- mutable

编译器

只读数据段：段中的数据只能读，不能改

数据段/堆栈段：可读，可改

const 的作用是给编译器看的！用于语法检查！

const 修饰的变量可以通过类型转换进行修改的!

变量的存储空间分配和回收的时间

生命周期

全局变量的生命周期

局部变量

- 静态局部变量的生命周期
- 非静态局部变量的生命周期

变量的存储空间分配和回收的时间

非静态局部变量的生命周期

- 并不是执行到定义语句时，才分配空间，而是一进入函数，就分配了空间。
- 编译的时候，编译器就做了“计划”，将局部变量放在何处（在栈中的一个相对位置）；
- 执行到函数时，相当于函数“取得栈的使用权”，函数内的局部变量“计划的空间”都“变成了现实”。
- 函数结束时，函数被“剥夺栈的使用权”，此时释放变量所占用的空间。

变量的存储空间分配和回收的时间


```
void f()
{
    int x = 10;
    for (int y = 0;y < 10;y++)
        x++;
    cout << x << endl;
    for (int z = 0;z < 5;z++)
        x++;
    cout << x << endl;
    int u = 20;
}
```

- 在此处设断点，监视窗口能否看到x ,u的值？
- 看到的x、u的值是多少？
- 能否看到 y,z 的值？

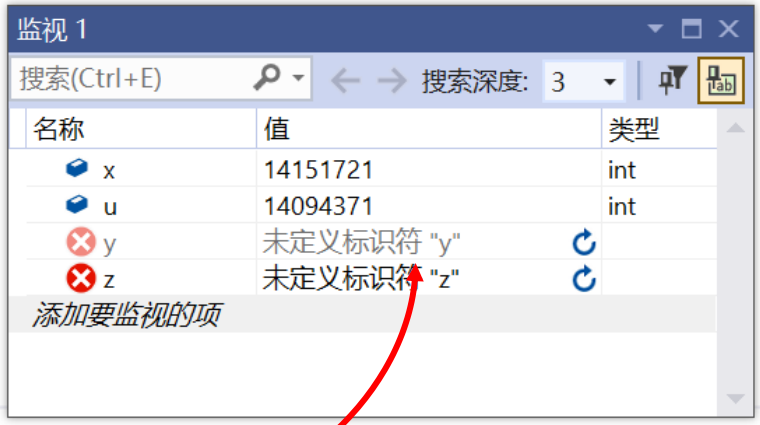
y、z的地址空间何时分配？

y、z的地址一样吗？

Q：如何验证？

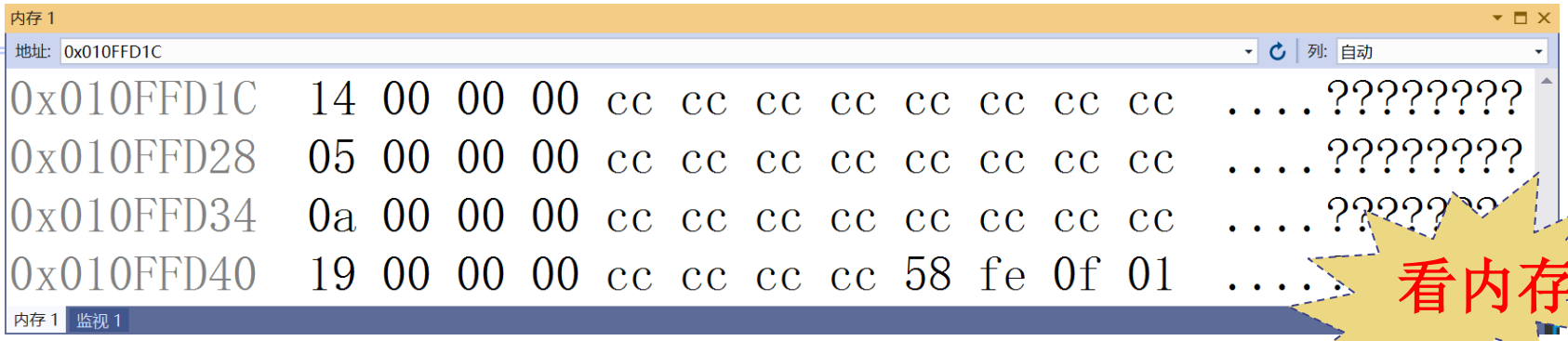
变量的存储空间分配和回收的时间


```
void f()
{
    int x = 10;
    for (int y = 0;y < 10;y++)
        x++;
    cout << x << endl;
    for (int z = 0;z < 5;z++)
        x++;
    cout << x << endl;
    int u = 20;
}
```



- y, z 作用域是在语句内；
- y, z 的 生命周期与x, u 相同； **如何证明？**
- 不可访问 与 不存在是两个完全不同的概念

变量的存储空间分配和回收的时间



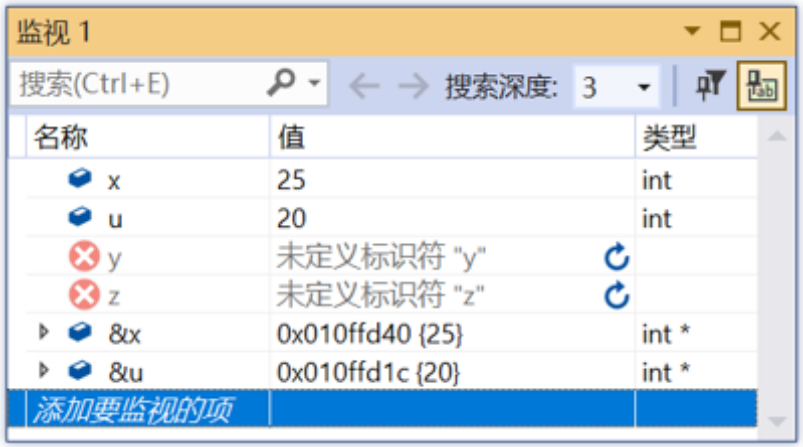
看内存

&u = 010ffd1c u=20 14 00 00 00

&z = 010ffd28 z=5 05 00 00 00

&y = 010ffd34 y=10 0a 00 00 00

&x = 010ffd40 x=25 19 00 00 00



在函数结束处设置断点，看 x、u 的地址、看内存

变量的存储空间分配和回收的时间

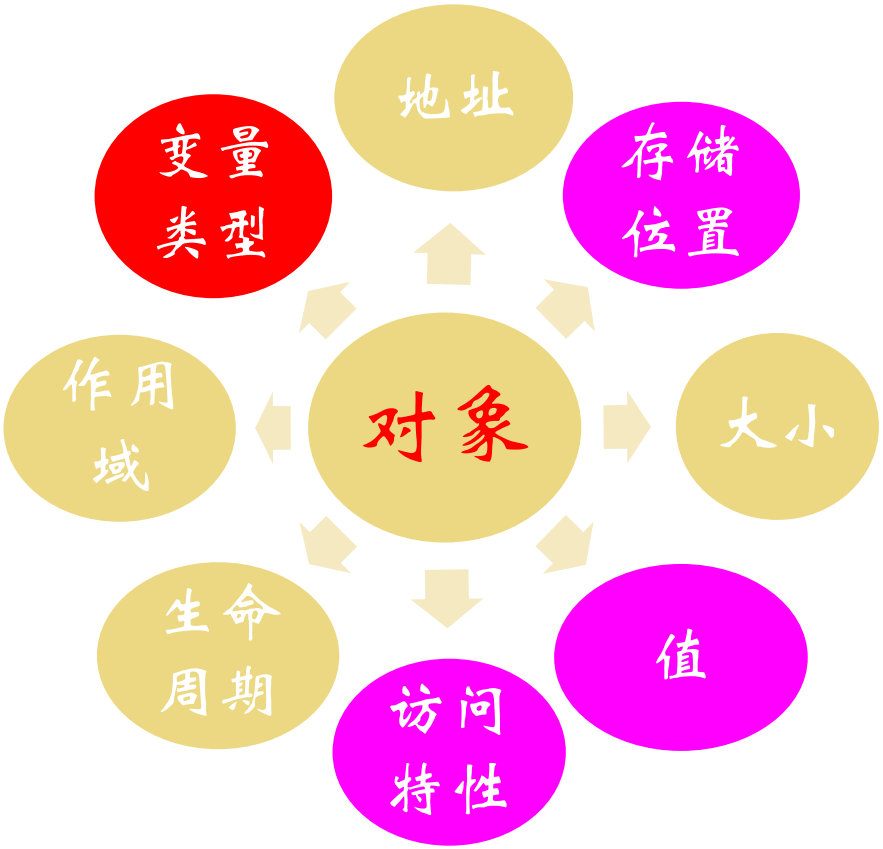

```
int x = 0;
            mov    dword ptr [ebp-4],0
 int y = 10;
            mov     dword ptr [ebp-8],0Ah
 for (int i = 0;i < 10;i++) x++;
            mov     dword ptr [ebp-0Ch],0
          ……
 for (int i = 100;i < 1000;i++) y++;
            mov     dword ptr [ebp-10h],64h
          ……
```

两个同名的语句级作用域变量 i, 分配的空间并不相同！

不可访问 与 不存在 是两个概念！

作用域　VS 生命周期

2.3 变量及其类型解析



C++ 中，变量就是对象

何时为对象分配空间？
何时释放对象的空间？

构造函数与分配空间有关吗？

析构函数与释放对象空间有关吗？

构造函数与析构函数的作用是什么？

2.3 变量及其类型解析

变量说明

- 描述变量的类型及名称，但没有初始化；
- 可以说明多次；

extern int x; ≠ int x;

变量定义

extern int x=1; = int x=1;

如果只有说明，没有定义，LINK时报错

无法解析的外部符号

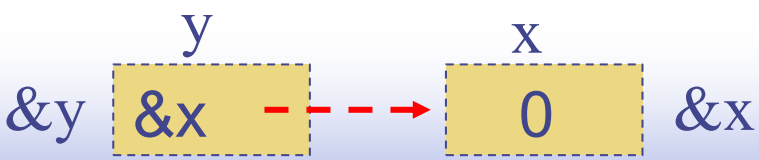
指针及其类型理解

- 指针类型的变量 使用*说明和定义

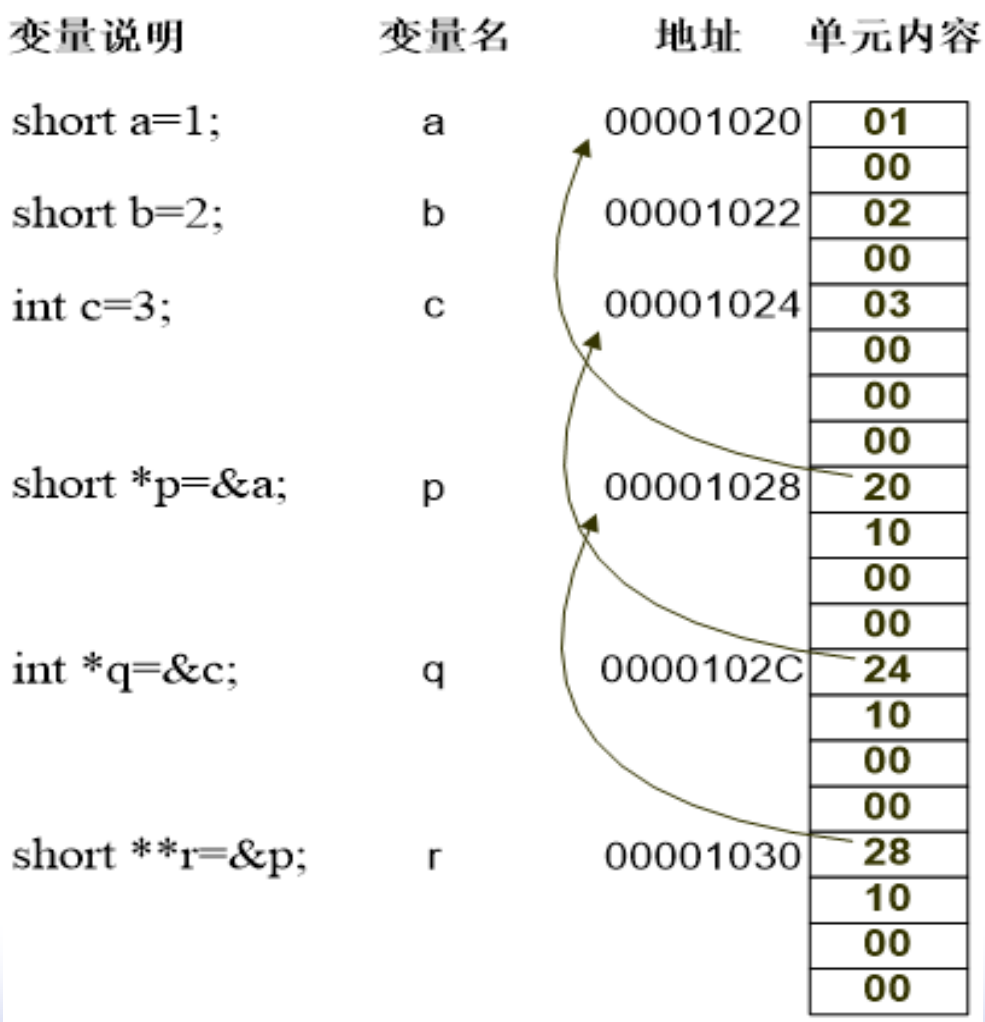
```
int    x=0;
int  *y=&x;  //指针变量y存放的是变量x的地址;
                // &x表示获取x的地址运算
                //  表示y指向x。
int    *p;
p=&x;
```

- 指针变量 涉及两个实体，

变量本身、以及 变量指向的变量。



2.3 变量及其类型解析



指针及其类型理解

*的结合性“自右向左”，故先解释右边的指针，再向左解释左边的指针

2.3 变量及其类型解析

数组

数组元素按行存储；

未存放每维的长度信息，没有办法自动实现下标越界判断；

每维下标的起始值默认为0；

数组名 代表数组的首地址；

一维数组名其代表的元素类型的指针 。

2.3 变量及其类型解析

字符串常量可看做以’\0’结束存储的字符数组。
strlen(“abc”)=3，但需要4个字节存储。
char c[6]=“abc”;　　//sizeof(c)=6，strlen(c)=3,
char d[ ]=“abc”;　　//sizeof(d)=4，
const char*p=“abc”;//sieof(p)=sizeof(void*)=4,
　　char　x=p[0];
　　p[0]=‘1’;　　// 语句错，不能给常量赋值
　　“abc”看作字符指针
　“abc”[0]　→　‘a’,
　*(“abc”+1)　→‘b’。

注意：sizeof, 与 strlen 的差别

2.3 变量及其类型解析

一维数组及其类型理解

int x[10];

解释: (1) x是一个10元素数组;

(2) 每个数组元素均为int 类型;

x[0], x[1], x[2] 都是 int 类型

x 可以看成 指向 int 类型的指针

元素访问 : x[5] ↔ *(x+5)

int *p; p = x; ↔ p = &x[0]; *(p+5)

二维数组及其类型理解

int x[10][20];

解释: (1) x是一个10元素(x[0]……x[9])数组;

(2) 每个元素(**x[0]**……x[9]) 又是 20个元素的数组;

x[0][0]、x[0][1] ……x[0][19]

(3) 每个元素 是 int 类型;

x[0][0] , x[0][1] …… 都是 int 类型

x[0], ……., x[9] 都代表 int [20] 的类型

x 可以看成是 int [20] 类型的指针，即 int (*)[20]

int *p; p = x[2]; ↔ p = &x[2][0];

int (*q)[20]; q=x; ↔ q = &x[0];

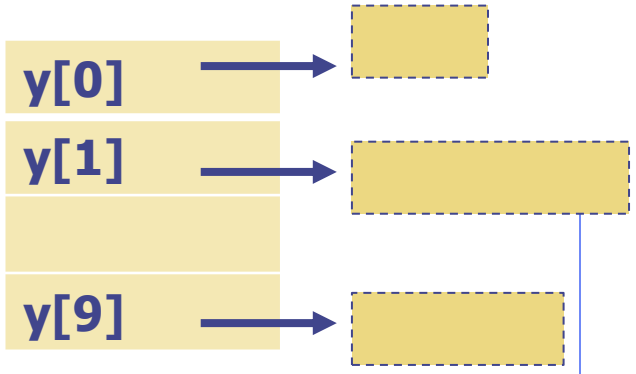
int **w; w = &p; p是 int *, &p 是 int ** 类型

指针数组及其类型理解

在一个类型表达式中，先解释优先级高，若优先级相同，则按结合性解释。

int *y[10];　　由 10个指针 排成的数组

在y的左边是*，右边是[10]，[ ]的优先级更高。



解释: (1) y是一个10元素数组;

(2) 每个数组元素均为指针;

(3) 每个指针都指向一个整数.

y[0], y[1], ….,y[9]都指向一个整数

指针数组及其类型理解

int *y[10][20];

在y的左边是*，右边是[10]，[ ]的优先级更高。

解释: (1) y是一个10元素数组；

(2) 每个数组元素均为20元素数组；

(3) 20个元素中的每个元素均为指针；

(4) 每个指针都指向一个整数。

数组指针及其类型理解

括号( )可提高运算符的优先级

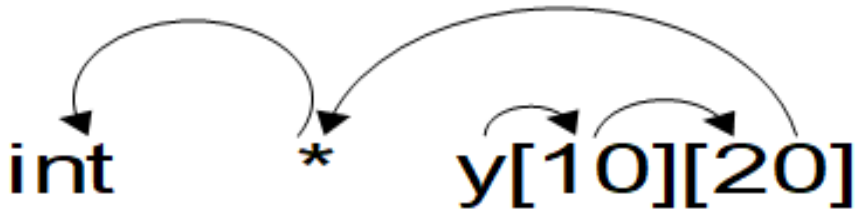
int (*z)[10][20];

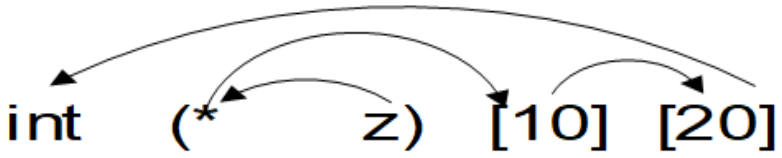
(…)、[10]、[20]的运算符优先级相同，按照结合性，应依次从左向右解释。

z是一个指针;

指向一个 [10][20] 的整型数组

2.3 变量及其类型解析





(b)定义int (*z)[10][20]

指针数组

- 本质是数组
- 每个元素是一个指针
- 分配 10*20*4个字节的空间

数组指针

- 本质是一个指针
- 只是一个变量
- 指向的是一个数组
- 分配 4个字节的空间

枚举类型

```
enum  WEEKDAY {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

WEEKDAY zzzzz = Sun;
mov    dword ptr [zzzzz],0
```

- 枚举一般被编译为整型，枚举元素对应的整型常量值；
- 第一个枚举元素的值默认为0，后一个元素的值默认值依次加 1。

```
typedef  int  WEEKDAY;
const      int  Sun=0,  Mon=1, Tue=2,…Sat=6;
```

2.3 变量及其类型解析

变量的存储可访问特性

只读变量　**const**

2.4 const 修饰符

const 约束

被const约束的单元，逻辑上不应被修改；

仅仅用于 编译器 进行语法检查和优化；

运行时，被语法约束的单元能否被“偷偷”修改，取决于数据分配在何处！

只读数据段中的内容不能被修改()；

数据段、堆栈段中的内容可以修改！

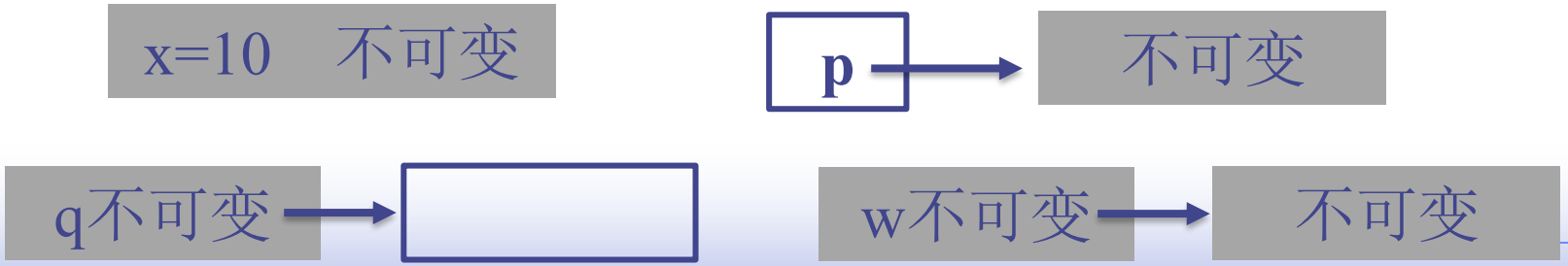
编译 VS 执行

“……”，字符串常量放在只读数据段，它们没有名字

2.4 const 修饰符

const 约束的语法、语义：单元不能修改 (灰色的框)

```
const  int  x =10;     x是 const int 类型
int   const   x=10;    // int const = const int
const char *p;         // p 是 const char * 类型    常量指针
                       //   p 是指向一个常量串的指针;
char * const q = (char *)malloc(10);      指针常量
                    //   q是一个常量，它指向一个可修改的串
 const char * const  w = “hello”;
```



2.4 const 修饰符

const 约束的单元不能修改

- 被const约束的变量在定义时必须初始化
- 被const约束的单元不应出现在赋值号的左边

```
const  int  x =10;
x=20;   // 不能给常量赋值      const int y; // 必须初始化
const char *p;       // p 占4个字节，上面没有const约束
*p = 'a';    // 不能给常量赋值
p = "good";     p="hello";    // p本身可变，可多次赋值
char * const q = (char *)malloc(10);
*q = 'a';     q[1]='b';
q =  (char *)malloc(20);   // 不能给常量赋值
char * const w;        // 必须初始化
```

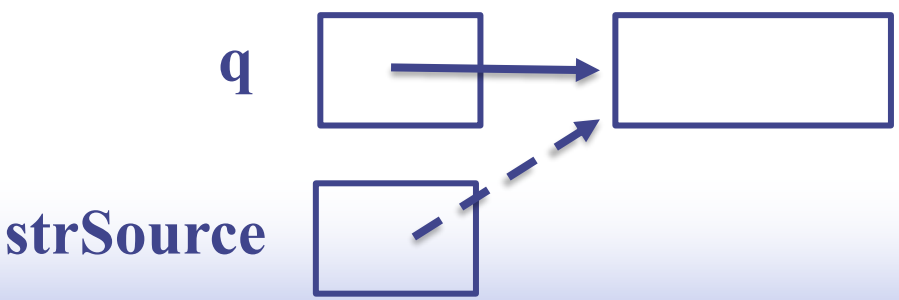
2.4 const 修饰符

注意：被约束单元不得修改，
是指不能通过加了约束限制的变量来修改。

```
char *strcpy( char *strDestination, const char *strSource );
char   p[20],  *q=(char *)malloc(20);
strcpy_s(p, q);
strcpy_s(p, “hello”);      // “hello” 对应的参数是 该串的地址
                           //   是 一个 const char * 的地址
```

参数传递： char * strDestination = p;

const char *strSource = q;　　strSource=“hello”;



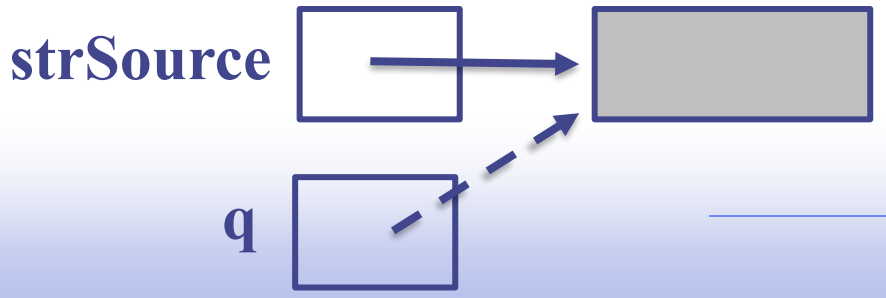
Q：能否修改q指向的串？
能否通过 strSource 修改指向的串？

2.4 const 修饰符

可以将一个指针赋给一个常量指针，
但不能将一个常量指针赋给一个指针

直观理解： 被操作的数据为客体；操作数据的人（即变量）为主体。一个只读数据不能交给具有写操作权限的人来操作；反过来，一个可写的数据可以交给一个只有读权限的人操作。

```
const  char *strSource;          char  *q;

 q= strSource;      // 无法从const char * 转换为 char *;

 strSource= q;
```



2.4 const 修饰符

总结

- 定义一个常量，必须在定义时赋初值；
- 常量不能在赋值号左边出现；
- 常量可以在赋值号右边出现；
- 常量指针只能赋值给一个常量指针；不能赋给一般的指针；
- 普通的指针可以赋值常量指针

```
const char *p;          p为常量指针，p本身可变
char  * const q =…. ;   q为常量，指针常量
```

2.4 const 修饰符

```
char  s[]="good";
char *p = new char[10];

const char  *pc = s;
pc[3] = 'g';
pc = p;
p=pc;
p=(char *)pc;

char  * const  cp = s;
cp[3] = 'g';
cp = p;
p = cp;
```

测验：判断语句的对错
判断的理由？

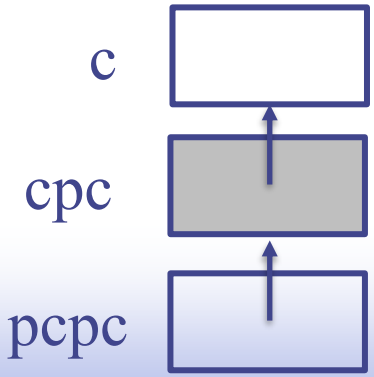
2.4 const 修饰符

const 修饰函数参数，防止在函数中修改参数数据。

```
char  * const * pcpc;

char   * * q;
```

不能修改 *pcpc 中的内容；可修改 pcpc、 ** pcpc。

```
char  c;
char  * const  cpc=&c;
pcpc =&cpc;
```



2.4 const 修饰符

讨论：

- 对于程序（或函数中）不需要（或不允许）变化的变量，加上const , 有什么好处？
- 能不能通过什么手段，修改const 变量的值呢？
- 如何记忆有关规则？

2.4 const 修饰符

- 能不能通过其他手段，修改const 变量的值呢？

```
int  xx;
cin >> xx;                          // 输入 xx  为 100
const int yy = xx;
cout <<“yy = ”<< yy << endl;    // 显示 yy = 100
*(int*)& yy = 123;
cout <<"yy = "<< yy << endl;   // 显示 yy = 123
```

原理：

- yy是 const int 类型， &yy 是 const int * 类型
- 采用强制地址类型转换 (int *),
  将 const int * ,转换为 int *，无 const约束了
- *(int *) 访问 yy

2.4 const 修饰符

讨论: 编译器对 地址类型转换的翻译
地址类型转换不改变地址的值,
转换的主要目的是让编译器通过语法检查。

```
*(int*)& yy = 123;
    mov    dword ptr [yy], 7Bh
```

等价写法:

```
*const_cast<int*>(& yy) = 123;  // mov  dword ptr [yy],7Bh
```

注意: int (yy)=123; // yy 重定义，不同类型的修饰符
const int(yy)=123; // yy重定义，多次初始化
int(yy)=123; 等同 int yy=123;

2.4 const 修饰符

华中科技大学

对const 约束的变量，编译器的优化

```
const int yy = 100;
cout <<“yy = ”<< yy << endl;    // 显示 yy = 100
*(int*)& yy = 123;
cout <<"yy = "<< yy << endl;
```

显示的 yy =?　　显示的 yy =100

原理：

编译器看到 yy 是一个const int, 又给定了值100;
就认为 yy不再会改变，后面直接用 100来代换了 yy。
*(int *)&yy语句是执行了的，调试时看得到yy=123;
但 cout 的结果是 yy =100. 定义 const int yy=xx; 就无法给 yy 一个常量值。

2.4 const 修饰符

讨论：为什么代码生成时，有时优化常量，有时不优化？

```
const int yy = 100;     VS     int  xx=100;
                               const int yy = xx;
```

下面同样一段程序的执行结果不同

```
cout <<“yy = ”<< yy << endl;    // 显示 yy = 100
*(int*)& yy = 123;
cout <<“yy = ”<< yy << endl;   // 显示100  VS  显示 123
```

```
volatile  const int yy = 100;  // 后面访问 yy时，都要直接
                               // 访问对应的内存单元
```

上面程序 显示　　yy =100
　　yy = 123

2.4 const 修饰符

华中科技大学

测试题：

```
char   *  const  p = new char[10];
char   q[20];
```

如何 让 p 能指向q，或者另一个新申请的空间？

```
*(char **)&p = q ;
*(char **)&p = new char[20];
*const_cast<char**>(&p) = q;
*const_cast<char**>(&p) = new char[20];
```

2.4 const 修饰符

测试题：

```
char* pc = "hello";   // 编译时报错
                      // 无法从 const  char[6] 转换为 char *
```

如何修改，使之无语法错误？

```
const  char * pc="hello";      // 方法 1

char * pc=(char *)"hello";   // 方法 2
```

虽然语法上，方法 1和2都正确，但都有潜在危险。

方法2 ： 直接通过 pc[i] 修改只读区的数据；

方法1 ： 通过强制类型转换，修改只读区的数据；

2.4 const 修饰符

华中科技大学

测试题：运行结果是什么？为什么？

char * pc=(char *)“hello”; // 方法 2

pc[0]=‘H’; // 在执行时出现问题

```
char* pc =(char *) "hello";
pc[0] = 'H';
return 0;
}
```

已引发异常

引发了异常: 写入访问权限冲突。
pc 是 0x529BEC。

原理：

“Hello”在只读数据存储区，其中的内容是不能修改的。

pc 指向了一个只读数据存储单元，不能修改pc指向的单元。

2.4 const 修饰符

测试题：对于下面的各种问题，如何写出更安全的程序？

```
char * p=(char *)“hello”;   // 有风险
                            // 直接通过 p 修改只读数据区
const char *pc = “hello”;    //  也有风险
        // 通过 pc数据类型的转换， 修改只读数据区
        pc[0]=‘H’;                  // 编译报错
        *(char *)&(pc[0])=‘H’;  // 运行报错
        *(char *)pc=‘H’;            // 运行报错
        *(char *)(pc+1)=‘H’;      // 运行报错
char  pa[10]=“hello”;     char pa[ ]=“hello”  // 保险做法
```

2.4 const 修饰符

讨论：const 与 #define 相比，有何优点？

- const 常量有数据类型，编译器可对其进行类型安全检查
- 宏常量无数据类型，没有类型安全检查
- 宏替换时，可能出现预想不到的错误

```
#define  doubled(x)   x*2

doubled(1+2) =  ?
```

- 在调试时，可对const 常量进行调试

2.4 const 修饰符

修饰一个对象

```
const   day   national_day(1949,10,1);
```

修饰一个类中的数据成员

```
class  day {  const  int  x=10;};
```

修饰一个类中的函数成员参数

```
class  day {  const  int  x=10;
                … f(const char *p);
};
```

修饰一个类中的函数成员

```
class  day {   int getyear( ) const;  };
```

2.4 const 修饰符

- 定义一个常量，必须在定义时赋初值；
- 常量不能在赋值号左边出现；
- 常量取地址后，变成一个常量指针(是一个指针，指向常量)；
  const int → const int *，不能修改指向的内容
- 常量可以赋值给一个普通变量，但常量指针不能赋值给普通指针； int ← const int; int * ← const int * //error
- 常量指针 ≠ 指针常量
- const 主要是编译时用于语法检查；
- 对于只读数据段中的数据，如“hello”这样的常量串，在运行时不得修改，否则程序崩溃。

2.4 const 修饰符

const 与 constexpr 的比较

- 两者都可以在变量定义之前，定义只读变量，无差别；
- const 可以用于函数参数的说明，constexpr 不行；
- const 和 constexpr 都可以用在函数返回类型之前，但两种写法的意义不同。

  const char * f(….); // 不能 *f(…)='c';

  constexpr char * f(…); // 可以 *f(…)='c';

  const 是存储可变特性

  constexpr 是用于编译器对函数优化。

```
constexpr int  increase(int x)
{  return x+1; }
constexpr int y=increase(1);  // 编译生成的语句为 y=2;
```

2.5 引用变量

2.5.1 有址引用变量 &

2.5.2 无址引用变量 &&

2.5 引用变量

引用变量的定义、初始化、使用

- 引用变量
- 引用参数
- 返回值引用
- 有址引用
- 无址引用

有址引用变量

编程：写一个函数，交换两个整型变量的值

```
int a=10;
int  b=20;
swap( ….., …..);          // 执行后 ， a=20, b=10
```

函数参数 应该是变量a、 b 的地址

```
swap(int *x, int *y)          swap( &a, &b);
{
   int  t=*x;
   *x=*y;
   *y=t;
}
```

有址引用变量


```
swap(int *x, int *y)
{   int  t=*x;
    *x=*y;
    *y=t;
}

int a=10;
int  b=20;
swap(&a, &b);
```

引用参数

```
swap(int &x, int &y)
{   int  t=x;
    x=y;
    y=t;
}

int a=10;       int  b=20;
swap(a, b);
```

参数传递，等同

```
int  &x=a;   int  &y=b;
```

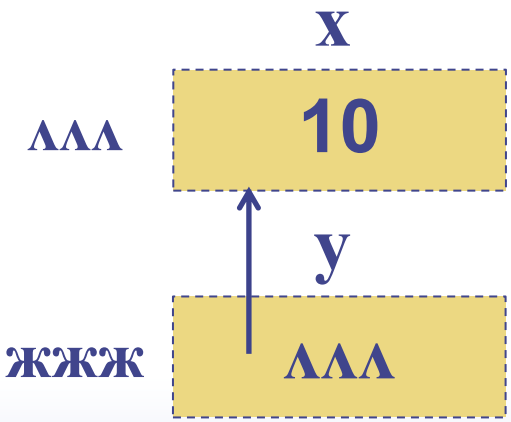
Q：在一个程序中，这两个函数能同时存在吗？

swap(a,b) 不会与 swap(int *x, int *y) 匹配，

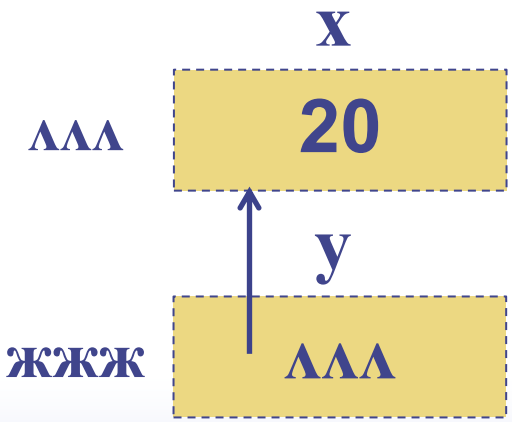
int *x=a; 是错误语句 // 无法从 int 转换为 int *

有址引用变量

```
int   x=10;

int   &y = x;        （1）

y=20;                （2）
```



执行(1) 后

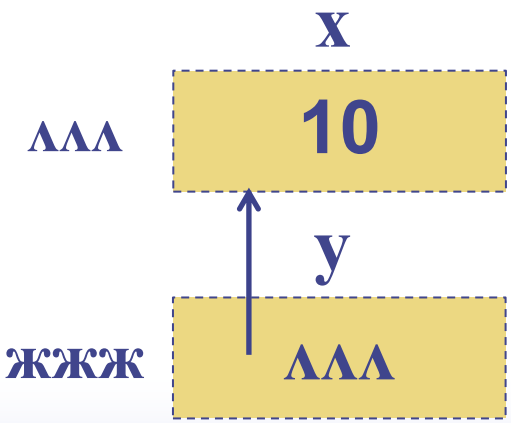


执行(2) 后

有址引用变量

引用变量

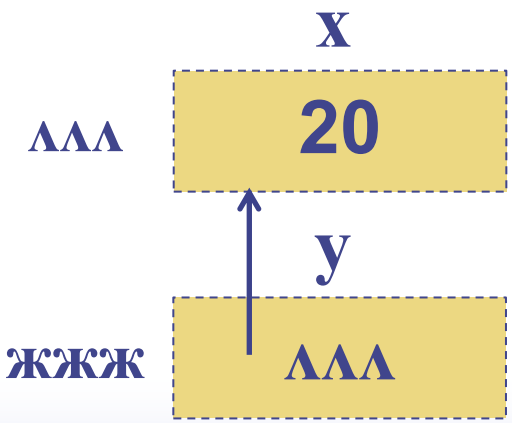
```
int   x=10;

int   &y = x;        （1）

y=20;                （2）
```



执行(1) 后

与指针的对比

```
int   *y;
y =  &x;

*y = 20;
```



执行(2) 后

为什么定义引用变量时就一定要初始化？

定义引用变量时的 = 与使用引用变量时的 = 的差别？

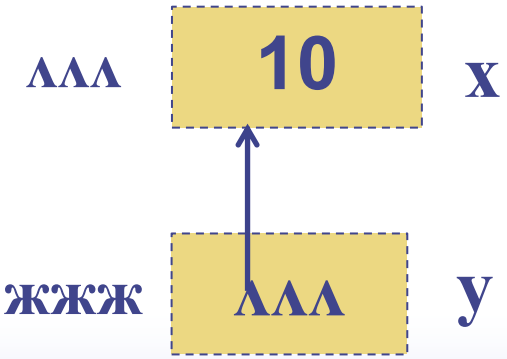
有址引用变量

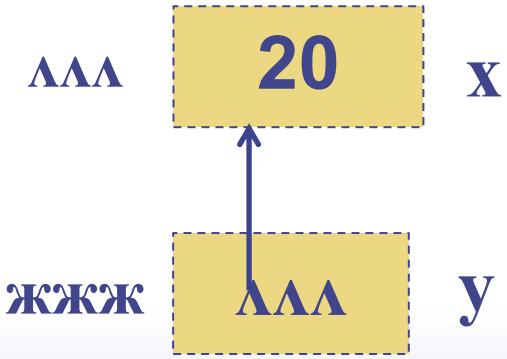

```
int   x=10, t;
int   &y = x;

y=20;


t = y;
```

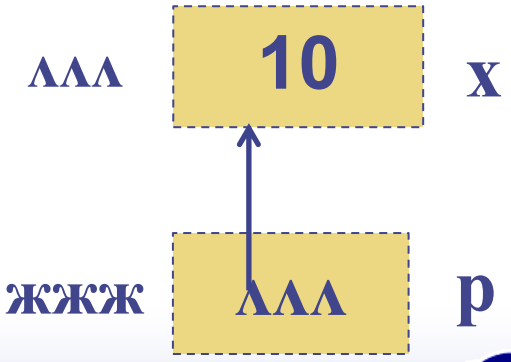
```
int   x=10, t;
int   *p=&x;

*p=20;
// *(&x)=20;

t = *p;
// t=20;
```

使用引用变量，等同使用被引用的变量。

内部实现的机制是指针。





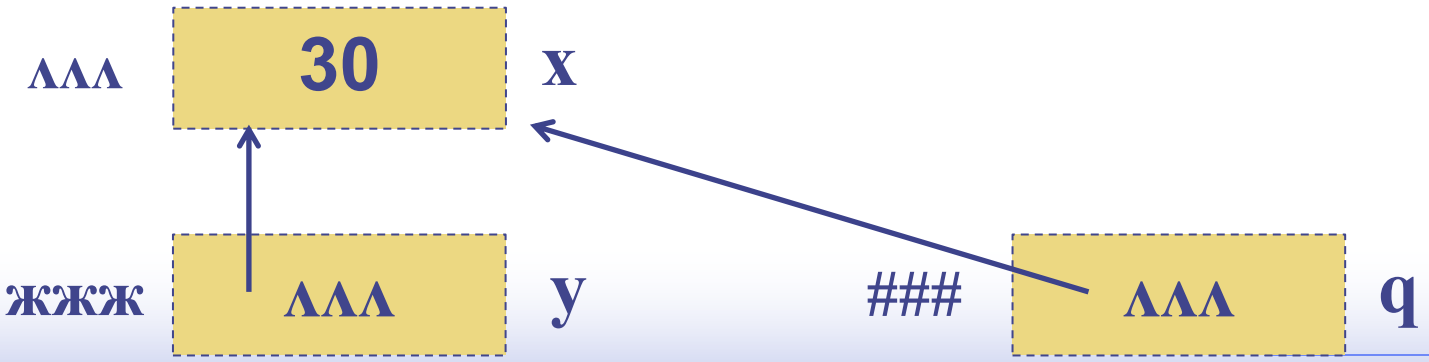


有址引用变量

```
int   x=10, t;
int   &y = x;
y=20;        // x=20;
t = y;       //  t=x;

int  *q = &y;      // 等同于 q=&x;
*q=30;     =>  *(&y) =30;     => y=30;
                                    ↓
           =>   *(&x) =30;    => x=30;
```

使用引用变量，等同使用被引用的变量。

对引用变量取地址，等同对被引用的变量取地址。



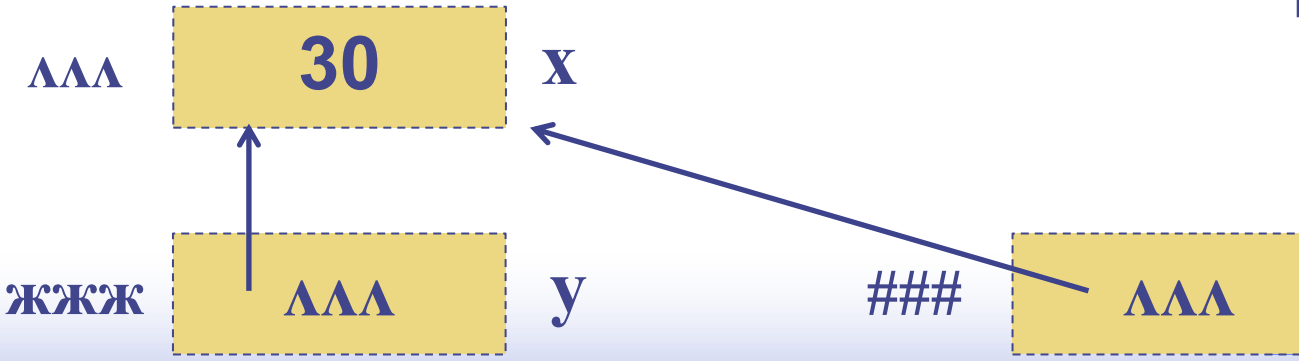
有址引用变量

对引用变量的再引用 要追溯到最原始的被引对象；不同于 二级指针

```
int   x=10, t;
int   &y = x;

int  &z =y;    →  int &z=x;

z=30;
```

```
int  *yy;
yy = &x;

int *zz;
zz = &(*yy);
   =yy = &x;

t=x;
t=y;
t=z;
```

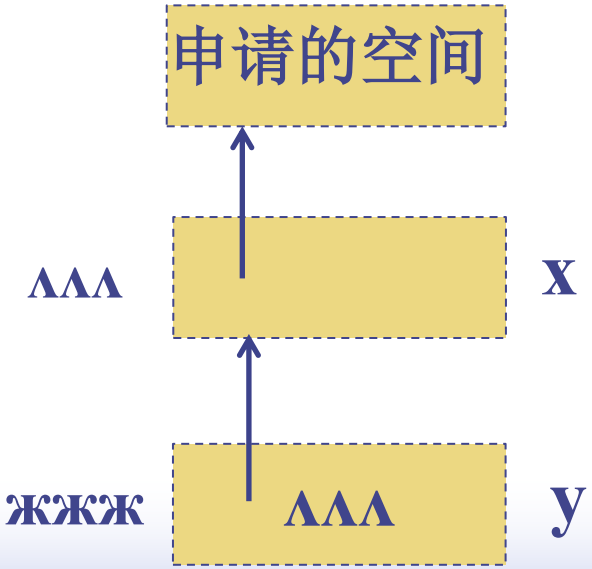
三者等价

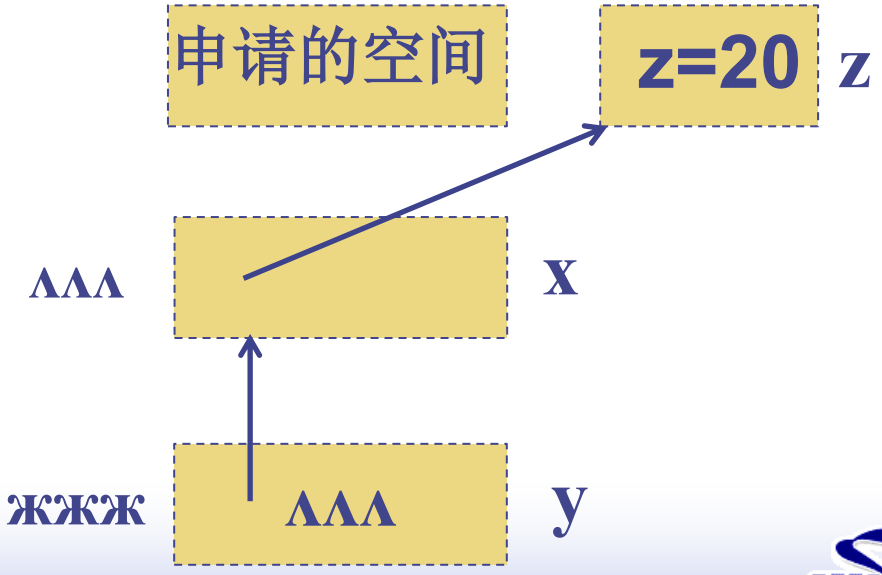


有址引用变量

对指针变量的引用

```
int   *x =(int*)malloc(sizeof(int)) ,     z=20;
int   * &y = x;
y=&z;     //  x=&z;     *(&x)=&z;
*y=30;    //  *x=30;
```

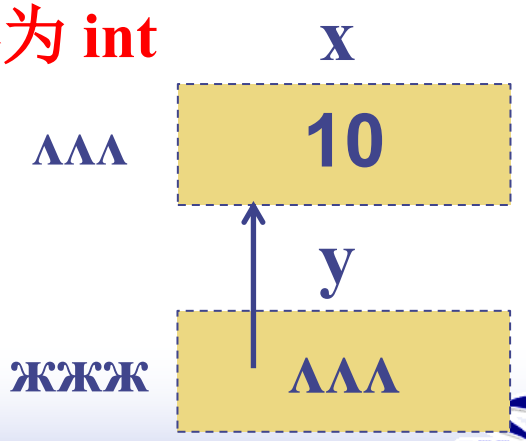




有址引用变量

定义引用变量时就一定要正确初始化

```
int   x=10;

int   &y;        // 错误语句，必须初始化引用
int   &y=20;   //错误语句，无法从“int”转换为“int &”
int   &y=(int *)malloc(sizeof(int));
                    // 无法从“int *”转换为“int &”
y =&x;     // 错误语句，无法从 int * 转换为 int

int  &y=x;
int   &y=*(int *)malloc(sizeof(int));
```



有址引用变量

important

- 定义引用变量时就一定要正确初始化；
- 引用变量中存放的是被引用变量的地址；
  其本质是指针；
- 在有串接（多级）引用时，都是指向最终被引用的变量；
  与二级（多级）指针有很大的差别；
- 使用引用变量，操作对象都是被引用的变量；
  取引用变量的地址，就是取被引用变量的地址；
  “*引用变量”，等同 “*被引用变量”

有址引用变量

测试：

int x; int &y=x;

const int u=10; const int &v=u; 能否去掉红色的const？

不行，无法从 const int 转化为 int &

int *p=NULL;

void f(int * &q) { q=(int *)malloc(10*sizeof(int); }

执行 f(p) 后，结果如何？ 若去掉 &，结果又如何？

p 指向了10个整数构成的空间 VS p 不变。

const int *s ; const int * &t=s; // 不能去掉红色的const

&s 是 const int * * 类型

有址引用变量


```
struct student
{       char name[20];
        int  age;
        int  weight;
};

student  xu;
print_info1(xu);
print_info2(xu);
```

```
void print_info1(student &s)
{
        cout<< s.name<<endl;
}

void print_info2(student s)
{
        cout<<s.name<<endl;
}
```

Q：两个函数调用传递的参数分别是什么？
采用引用参数有何好处？

Q：能否将print_info2的名字，改成print_info1？为什么？

有址引用变量

返回结果为引用

```
int & fr ( ) {
    int  t=25;
    return t;
}

int    a=fr( );
```

```
int  fv ( ) {
    int  t=25;
    return t;
}

int      a=fv( );
```

两者都显示 a=25

但是两者内部实现有差异。fr返回的是t的地址;
fv返回的是t的值;

fr : warning C4172: 返回局部变量或临时变量的地址

有址引用变量

返回结果为引用

```
int & fr ( ) {
    int  t=25;
    return t;
}
```

```
int t=25;
int &const frv = t;
```

=

```
int     a=fr( );    a=frv;

fr( )=20;           frv=20;
```

```
int  fv ( ) {
    int  t=25;
    return t;
}
```

```
int t=25;
int  const fvv = t;
```

=

```
int       a=fv( );    a=fvv;

fv( )=20;           fvv=20;
// 错误
```

有址引用变量

返回结果为引用

```
int & f( ) {
    int  t=25;
    return t;
}

int   x=10;
int   &y = x;     //  y中内容是x的地址，y 引用 x
int  u  = y;      //  u中内容 = x中的内容，即 10
int  &v=y;        //  v中内容是 x的地址，v 引用x

int    a=f();     // 执行后  a=25;
int    &b = f();  // 执行后b 中的
                  内容 为 t 的地址
```

与传统指针相比： int a; int &y=a ; int *p=&a;

int u = y ; int u=*p; int &v=y ; int &v= (*p)=a;

有址引用变量

返回结果为引用

```
int & f( ) {
    int  t=25;
    return t;
}
int  x = f( );
```

```
int    g( ) {
    int  t=25;
    return t;
}
int  y = g( );
```

// x，y 的值都是25，但实现的方法不同
// f 函数编译有警告

```
int  &u= f( );
```

```
int  &v = g( );
```

//无法从int 转换为 int &

&v 是有址引用；

int &v 是传统左值有址引用

故 = 右边应为有址左值

有址引用变量

返回结果为引用

```
int & f(int t ) {
    t=t+10;
    return t;
}

a=f(10)+f(20) ;
          // 显示 a = 60
b = f(20)+f(10);
          // 显示 b = 40
```

```
int  f(int  t ) {
    t=t+10;
    return t;
}

a=f(10)+f(20) ;
          // 显示 a = 50
b = f(20)+f(10);
          // 显示 b = 50
```

warning C4172: 返回局部变量或临时变量的地址

有址引用变量

返回结果为引用

```
int & f(int t ) {
    t=t+10;
    return t;
}
```

```
a=f(10)+f(20) ;
          // 显示 a = 60
b = f(20)+f(10);
          // 显示 b = 40
```

a=f(10)+f(20)

先执行 f(10),
返回 函数f中变量t 的地址

再执行 f(20)
返回 函数f中变量t 的地址

根据第1个返回地址，取相应单元的内容，为 30
根据第2个返回地址，取相应单元的内容，为 30
故 a =60

warning C4172: 返回局部变量或临时变量的地址

有址引用变量

返回结果为引用

```
int & f(int t ) {
    t=t+10;
    return t;
}
```

将函数视为一个变量

int & f=t;

```
a=f(10)+f(20) ;
            // 显示 a = 60
push      0Ah
call      f (08D1190h)
add       esp,4
mov       esi,eax
push      14h
call      f (08D1190h)
add       esp,4
mov       ecx,dword ptr [esi]
add       ecx,dword ptr [eax]
mov       dword ptr [a],ecx
```

有址引用变量

返回结果为引用

```
int & f(int t ) {
    int *p=(int *)malloc(sizeof(int));
    *p=t+10;
    return  *p;
}

a=f(10)+f(20) ;
          // 显示 a = 50
b = f(20)+f(10);
          // 显示 b = 50
```

删除引用& 后，结果同样正确。

有址引用变量

讨论：

- 指针和引用有什么差别？
- 传引用比传指针安全，为什么？

- 引用在创建时，必须初始化，即引用到一个有效的对象；
  指针在定义时，可以不 初始化

- 指针可以为NULL
  引用必须与合法的存储单元关联，不存在NULL引用

- 引用一旦被初始化为指向一个对象，就不能再改变为另一个对象的引用；指针是可变的

有址引用变量

讨论：

- 引用 比 指针安全

```
int  *p= (int *)malloc(10*sizeof(int));        free(p);

int &q = *(int *)malloc(10*sizeof(int));       free(&q);
```

p[i] = 20;　　*(p+i)=20;

*(&q+i)=20;　　=> *(& (*p)+i)=20 => *(p+i)=20;

使用 q 相当于 *p;

p本身可改；　p=(int *)malloc(20 *sizeof(int));

q本身不可改；q = 20; 实际上改的是 q 引用的单元

有址引用变量

讨论：引用 类型

```
const  int  p= 10;
*(int *)&p =20;          (int &)p=20;
                         int  &q = *(int *)&p;
                    or   int  &q = (int &)p;
                         q=20;
```

int &q=p; //错，无法从const int 转换为 int &

```
int  x;
int * const q =&x;
*(int **)&q= &x;         (int *&)q=&x;
```

无址引用变量

&&定义 无址引用

```
int &&x=2;
const int  &&w=3;
```

- 无址引用&& 的本质与 有址引用&的本质相同；
- 引用变量中，存放的也是被引用的地址；
- 可以通过引用变量，修改被引用的对象；

差别：一个是用名变量的地址；
　　一个是临时对象的地址。

用途：移动构造、移动赋值

有址引用 与无址引用

- 有址引用

一般：被引用的对象有地址　int x;　int &y=x;

可通过引用变量 修改，引用变量是一个左值；

传统左值有址引用

特殊：　被引用的对象有地址　int x; const int &y=x;

不能通过引用变量 修改，引用变量是一个右值；

传统右值有址引用

特　例：　const int &w=2;　　不允许 int &w =2;

被引用对象存放在一个临时地址单元中；是一个无名地址；不能通过 引用变量去修改，

传统右值有址引用

(更合适的说法：传统右值无址引用，const int &&w=2)

有址引用 与无址引用

- 无址引用

一般：被引用的对象无（有名）地址

可通过引用变量 修改，引用变量是一个左值；

传统左值无址引用　　int &&x = 2;　　x=3;

特殊：　被引用的对象无地址

不能通过引用变量 修改，引用变量是一个右值；

传统右值无址引用　　const int && y=4;

int　p;　int &&q = p; // 错，被引用对象有（有名）地址

const int p=10;　const int &&q=p; //错，　被引用对象有地址

有址引用 与无址引用


```
const  int  & p =2;
const  int  & p = x;        int  x;
```

传统右值有址引用，既可以接收 无址的地址，也可以接收有名的地址，因而，可以广泛地作为函数参数。

```
const  char  *p = “hello”;
const char   *p = q;          char *q;
```

有址引用 与无址引用

Q: 下面的表达式，哪些是传统左值？哪些是传统右值？

int x;	const int y=10;
++x;	x++
	x+3
	(float)x
*(float *)&x	
*(int *)&y	y
int &u=x;	const int &v=2;
u	v

SUMMARY

[存储位置特性] [存储可变特性] 类型名　变量名　[= 初始值];

- auto
- register
- static
- extern
- const
- constexpr
- volatile
- mutable

static 变量的作用域、生命周期、用途、用法

const 常量的含义、语法、优化、执行

被约束单元不可变，但可将单元中的内容拷贝给别人

不能将被约束单元的地址 送给 不受约束的指针

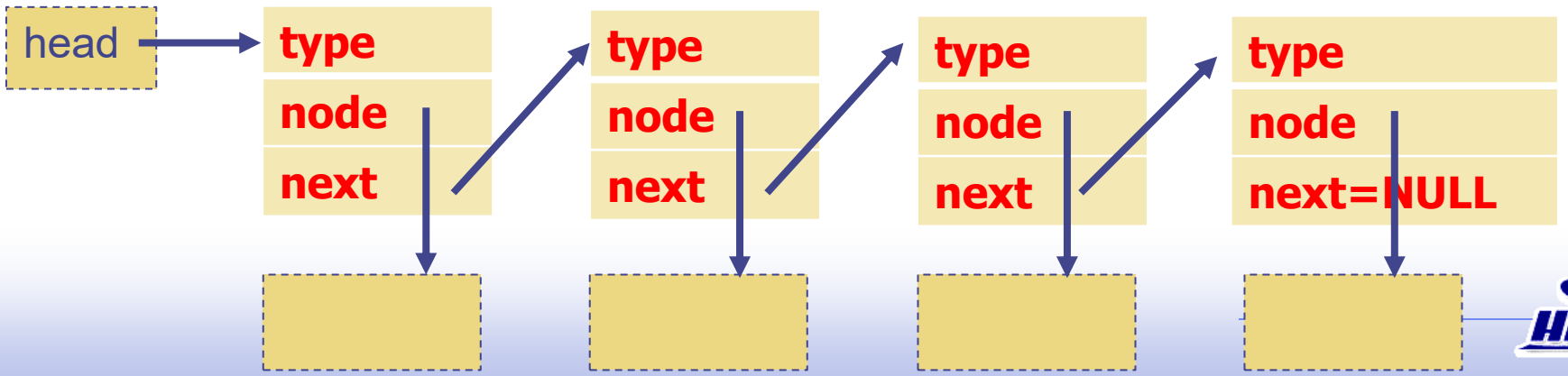
引用变量　&、&&：

本质是指针、使用时直接当成被引用的变量（对象）

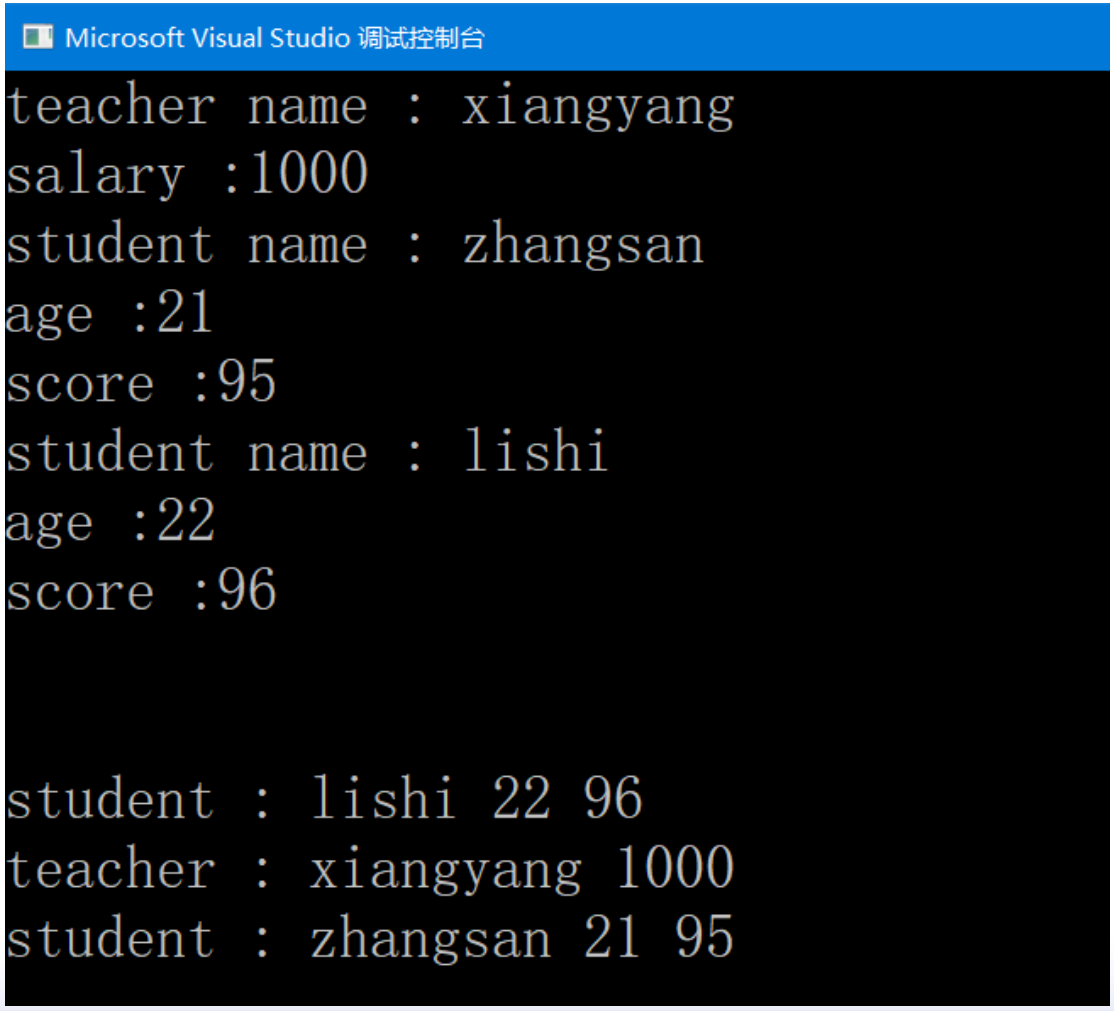
编程作业

异质链表

```
struct student {
    char name[10];    int  age;   int  score;  };
struct teacher {
    char name[10];     float salary;  };

struct person {
    int  type;   // type=1 , student; =2 teacher
    void* node;
    person* next;
};
```



编程作业


Microsoft Visual Studio 调试控制台
teacher name : xiangyang
salary :1000
student name : zhangsan
age :21
score :95
student name : lishi
age :22
score :96
student : lishi 22 96
teacher : xiangyang 1000
student : zhangsan 21 95

编程作业

请指出下面两个函数定义、函数体及函数调用语句的差异

```
student & new_student() {…… }        // 生成一个新学生
teacher* new_teacher() { …… }        //  在函数中输入相关信息
```

请补充 void display_info(const person * head) 的函数实现体。

改写下面的函数，第1个参数不使用引用，注意修改相应的函数的调用语句

```
void insert_person(person * & listhead, int type, void *p)
```