



华中科技大学

第2章 类型、常量及变量

许向阳

xuxy@hust.edu.cn



2.1 C++的单词

2.2 预定义类型及值域和常量

2.3 变量及其类型解析

2.4 运算符及表达式

2.5 结构和联合



重点和难点

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





2.1 C++的单词

单词：由ASCII字符集中的字符来组成。

ASCII：American Standard Code for Information Interchange，美国标准信息交换代码

26个大写字母：A~Z

26个小写字母：a ~ z

10个阿拉伯数字：0 ~ 9

其他符号：+、-、*、/、：、；、？、（、）、
[、]、{、}、&、^、>、< 等等



2.1 C++的单词

单词：常量、变量名、函数名、参数名、类型名、运算符、关键字等。

alignas↵	continue↵	friend↵	register↵	true↵
alignof↵	decltype↵	goto↵	reinterpret_cast↵	try↵
asm↵	default↵	if↵	return↵	typedef↵
auto↵	delete↵	inline↵	short↵	typeid↵
bool↵	double↵	int↵	signed↵	typename↵
break↵	do↵	long↵	sizeof↵	union↵
case↵	dynamic_cast↵	mutable↵	static↵	unsigned↵
catch↵	else↵	namespace↵	static_assert↵	using↵
char↵	enum↵	new↵	static_cast↵	virtual↵
char16_t↵	explicit↵	noexcept↵	struct↵	void↵
char32_t↵	export↵	nullptr↵	switch↵	volatile↵
class↵	extern↵	operator↵	template↵	wchar_t↵
const↵	false↵	private↵	this↵	while↵
constexpr↵	float↵	protected↵	thread_local↵	↵
const_cast↵	for↵	public↵	throw↵	↵

关键字：保留字，
不能用作变量名。



2.2 预定义类型及值域和常量

类型的字节数与硬件、操作系统、编译有关。

如采用 VS1029、x86编译模式，则有：

bool: 单字节布尔类型，取值false和true。

char: 单字节有符号字符类型，取值-128~127。

short: 两字节有符号整数类型，取值-32768~32767。

int (即 long): 四字节有符号整数类型，

float: 四字节有符号单精度浮点数类型

double: 八字节有符号双精度浮点数类型

void: 字节数不定，常表示函数无参或无返回值。

默认整数常量（如123）当作为int类型，

默认浮点常量（如12.3）当作double类型。





2.2 预定义类型及值域和常量

- long int等价于long;
- long long占用八字节;
- $\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(double)}$;



2.2 预定义类型及值域和常量

➤ char、short、int、long前加**unsigned**表示无符号数；

```
int x1;
```

```
signed int x2;
```

```
unsigned int x3;
```

Q: x1= -1; x3= -1; // ff ff ff ff

x1 与 x3 中存储的内容有无差别？

Q: if (x1>0) 的条件成立吗？

if (x3>0) 的条件成立吗？





2.2 预定义类型及值域和常量

- 不同**整数类型**的变量赋值，如何进行转换？

```
int i1;    short s1;    char c1;    bool b1;  
unsigned int i2; unsigned short s2; unsigned char c2;
```

- **字节数少的类型 向 字节多的类型转换**

`i1 = s1;` `short → int` => 赋值给 `int`

`i1 = s2;` `unsigned short → unsigned int` => 赋值给 `int`

step 1: 短变长，有/无符号特性保持不变;

step 2: 相等长度直接赋值





2.2 预定义类型及值域和常量

```
short s1 = 0xffff;
    or    eax, 0FFFFFFFFh
    mov   word ptr [s1], ax
unsigned short s2 = 0xffff;
    mov   eax, 0FFFFh
    mov   word ptr [s2], ax

int  i1;
i1 = s1;
    movsx  eax, word ptr [s1]
    mov    dword ptr [i1], eax

i1 = s2;
    movzx  eax, word ptr [s2]
    mov    dword ptr [i1], eax
```

step1:

短变长，有/
无符号特性保
持不变；

step 2:

相等长度直接
赋值





2.2 预定义类型及值域和常量

```
short s1;      unsigned short s2;
int  i1;      unsigned int  i2;
i1 = s1;
    movsx     eax, word ptr [s1]
    mov       dword ptr [i1], eax
i1 = s2;
    movzx     eax, word ptr [s2]
    mov       dword ptr [i1], eax
i2 = s1;
    movsx     eax, word ptr [s1]
    mov       dword ptr [i2], eax
i2 = s2;
    movzx     eax, word ptr [s2]
    mov       dword ptr [i2], eax
```

step1:

短变长，有/
无符号特性保
持不变；

step 2:

相等长度直接
赋值





2.2 预定义类型及值域和常量

- 不同类型的变量赋值，如何进行转换？

```
int i1;   short s1;   char c1;   bool b1;  
unsigned int i2; unsigned short s2; unsigned char c2;
```

- 字节少的类型 向 字节多的类型转换

step 1: 短变长，它的有/无符号特性保持不变；

step 2: 相等长度直接赋值

- 相同长度的有/无符号 整数类型的转换

直接复制拷贝 （int ↔ float, 要进行类型转换）

- 长度长的类型向短类型转换

截断





2.2 预定义类型及值域和常量

➤ 向 bool 类型的转换

bool 类型只有两个值 0、1。 1 ↔ true; 0 ↔ false
非零数值转换为布尔值 true

```
bool  bt;
```

```
int    y = 0x1000;
```

```
bt = y;
```

```
        cmp    dword ptr [y], 0
```

```
        je     f+72h (04E1842h)
```

```
        mov    byte ptr [ebp-109h], 1
```

```
        jmp    f+79h (04E1849h)
```

```
04E1842h  mov    byte ptr [ebp-109h], 0
```

```
04E1849h  mov    al, byte ptr [ebp-109h]
```

```
        mov    byte ptr [bt], al
```





2.2 预定义类型及值域和常量

- 将一个数转换为布尔类型时，
数值零 自动转换为布尔值false， $0 \rightarrow \text{false}$;
非零数值转换为布尔值true。 $\text{非}0 \rightarrow \text{true}$

Q: `int x=3;`
`if (x)` 写法正确吗？ 条件成立吗？

`cmp x, 0`
`je` 转到 else 分支处





2.2 预定义类型及值域和常量

- 不同类型的变量形成一个表达式，如何计算？

```
int i1; short s1; char c1;  
unsigned int i2; unsigned short s2; unsigned char c2;  
short xxx; int yyy;  
xxx = c1 + s1 + i1 + c2 + s2 + i2;  
yyy = c1 + s1 + i1 + c2 + s2 + i2;
```

- 计算表达式 与 赋值运算无关

if (c1+s1+i1+c2+s2+i2)

- 字节数少的类型 向 字节最多的类型转换

char → int short → int

unsigned char → unsigned int

unsigned short → unsigned int

- 加、减法运算 不区分有符号数、无符号数





2.2 预定义类型及值域和常量

```
short xxx=c1 + s1 + i1 + c2 + s2 + i2;
```

```
movsx      eax, byte ptr [c1]
```

```
movsx      ecx, word ptr [s1]
```

```
add        eax, dword ptr [i1]
```

```
add        ecx, eax
```

```
movzx      edx, byte ptr [c2]
```

```
add        ecx, edx
```

```
movzx      eax, word ptr [s2]
```

```
add        ecx, dword ptr [i2]
```

```
add        eax, ecx
```

```
mov        word ptr [xxx], ax      截断赋值
```

```
int yyy=c1 + s1 + i1 + c2 + s2 + i2;
```

```
mov        dword ptr [yyy], eax
```

//只是最后一条语句不同





2.2 预定义类型及值域和常量

- `if (c1 + s1 + i1 + c2 + s2 + i2 > 3)`
最后的比较 运算 是 无符号比较
- `if (x > y)` `x`, `y` 一个有符号, 一个无符号
无符号比较
warning C4018: “>”: 有符号/无符号不匹配
- `if (x > 5)` 一个有类型, 一个是无类型的数值
按 明确的类型 进行比较
- 乘、除运算, 若有一个运算数是有符号的, 则采用 有符号运算





2.2 预定义类型及值域和常量

➤ 强制类型转换的格式为:

(类型表达式) 数值表达式

➤ 字符常量: 'A', 'a', '9', '\'(单引号), '\\'(斜线),
'\n'(换新行), '\t'(制表符), '\b'(退格)

➤ 整型常量:

9, 04, 0xA (int);

9U, 04U, 0xAU (unsigned int);

9L, 04L, 0xAL (long);

9UL, 04UL, 0xAUL (unsigned long);

9LL, 04LL, 0xALL (long long);

➤ double常量: 0.9, .3, 2E10, 2.E10, -2.5E-10





2.2 预定义类型及值域和常量

预定义类型的数值输出

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

```
printf(“%d”,4); %开始的输出格式称为占位符
```

```
char: %c;
```

```
short, int: %d;    long: %ld;
```

输出:

十进制数用 %d

无符号数十进制数用 %u

八进制数用 %o

十六进制用 %x





2.2 预定义类型及值域和常量

输出数据宽度及对齐

`printf(“%5c”, ‘A’)` 打印字符占5格 (右对齐)。

`%-5d`表示左对齐。

➤ `float: %f; double: %lf`。

`float, double: %e` 科学计数。

`%g` 自动选宽度小的e或f。

➤ 可对`%f`或`%lf`设定宽度和精度及对齐方式

“`%-8.2f`”：左对齐、

总宽度8(包括符号位和小数部分)、
精度为2位小数。





2.2 预定义类型及值域和常量

- 字符串输出：%s。

可设定宽度和对齐：printf(“%5s”, “abc”)。

- 字符串常量的类型：

“abc”

指向只读字符的指针即const char *

注意：strlen(“abc”)=3，但要4个字节存储，最后存储字符 ‘\0’，表示串结束。





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？
- 变量有何访问特性？（存储可变特性）

[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

变量的存储位置特性

在函数内部定义局部变量

- **auto** 默认值
- **register**
- **static**
- **extern**

在函数外定义全局变量

- **static**
- **extern** 默认值

数据段 VS 堆栈段





2.3 变量及其类型解析

变量的存储可访问特性

- `const`
- `constexpr`
- `volatile`
- `mutable`



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;` \neq `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





2.3 变量及其类型解析

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

变量说明	变量名	地址	单元内容
short a=1;	a	00001020	01 00
short b=2;	b	00001022	02 00
int c=3;	c	00001024	03 00 00 00
short *p=&a;	p	00001028	20 10 00 00
int *q=&c;	q	0000102C	24 10 00 00
short **r=&p;	r	00001030	28 10 00 00

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

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 : $x[5] \leftrightarrow *(x+5)$

```
int *p;        p = x;     $\leftrightarrow$     p = &x[0];
```





2.3 变量及其类型解析

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
```



2.3 变量及其类型解析

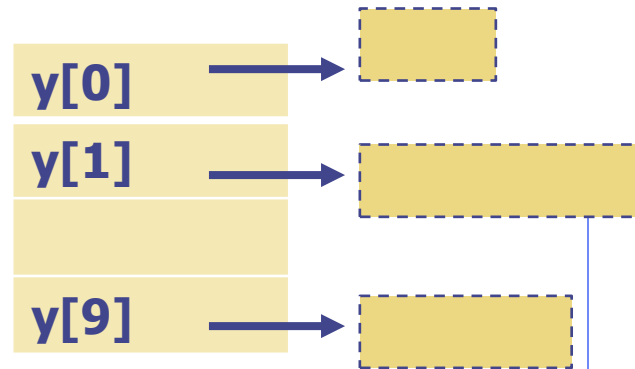
指针数组及其类型理解

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





2.3 变量及其类型解析

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





2.3 变量及其类型解析

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];
```

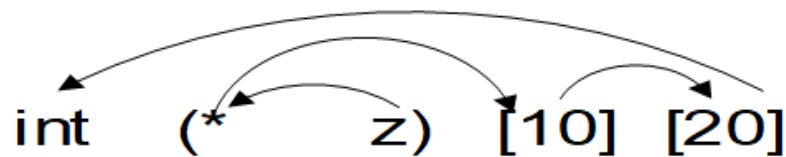
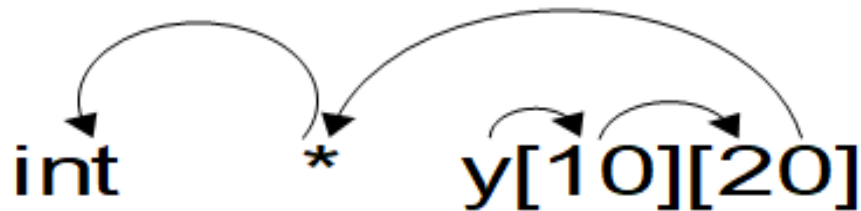
(...)、[10]、[20]的运算符优先级相同，按照结合性，应依次从左向右解释。

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 $10 \times 20 \times 4$ 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



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**





2.3 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";`

`x=10` 不可变



不可变

`q`不可变



`w`不可变

不可变





2.3 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**约束的单元不能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

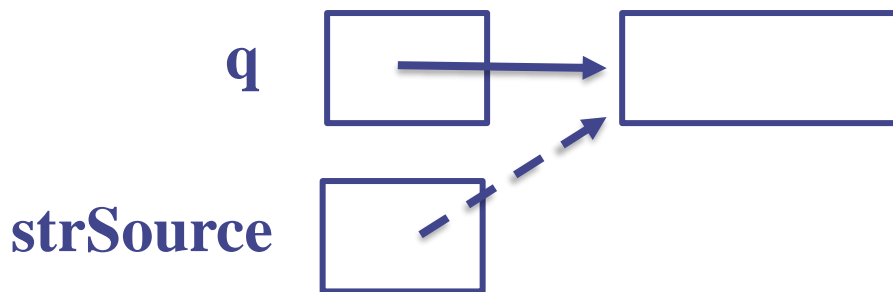
char * const w; // 必须初始化



2.3 const 修饰符

```
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";



正确理解被约束单元不得修改，是不能通过加了约束限制的变量来修改。

Q: 能否通过 q 修改指向的串？

能否通过 strSource 修改指向的串？

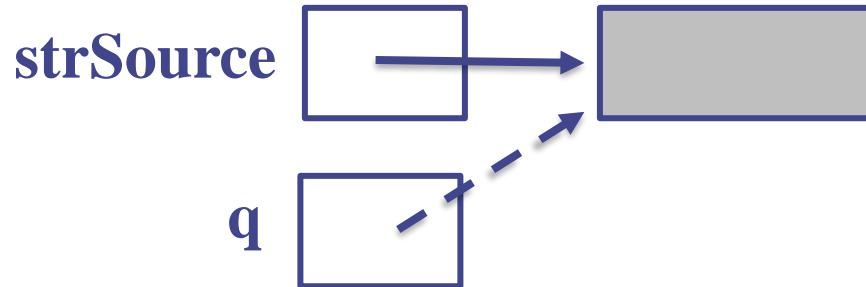
2.3 const 修饰符

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

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

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

```
strSource = q;
```



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。



2.3 const 修饰符

important

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

`char`、`const char`、`const char *`、`char * const` 都是类型





2.3 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？



2.3 const 修飾符

const 修飾函數參數，防止在函數中修改參數數據。

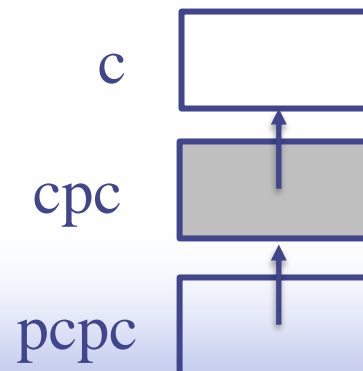
```
char * const * pcpc;
```

```
char ** q;
```



不能修改 ***pcpc** 中的內容；可修改 **pcpc**、**** pcpc**。

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





2.3 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？





2.3 const 修饰符

讨论:

- 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





2.3 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;**





2.3 const 修饰符

讨论：代码生成中的优化， 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。





2.3 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





2.3 const 修饰符

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];
```





2.3 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；






2.3 const 修饰符

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。





2.3 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 * pc=(char *)“hello”; // 有风险`

`// 直接通过 pc 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`





2.3 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





2.3 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; };
```





2.3 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。





2.3 变量及其类型解析

2.3.3 有址引用变量 &

2.3.4 无址引用变量 &&





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

```
int x=10;
```

```
int &y = x;    (1)
```

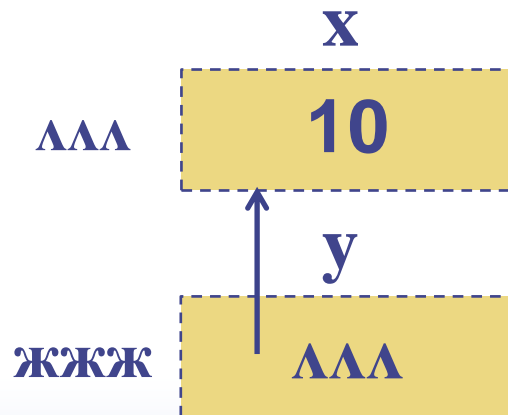
```
y=20;        (2)
```

```
lea  eax,[x]
```

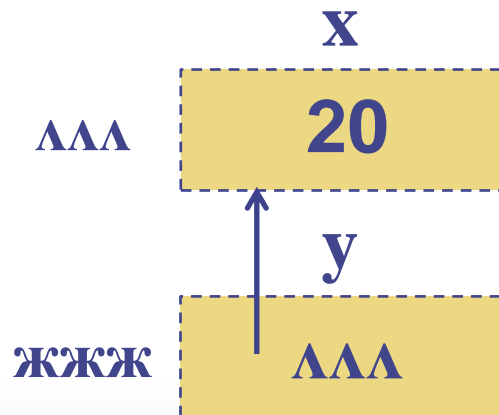
```
mov  dword ptr [y],eax;
```

```
mov  eax,dword ptr [y];
```

```
mov  dword ptr [eax],14h
```



执行(1) 后



执行(2) 后



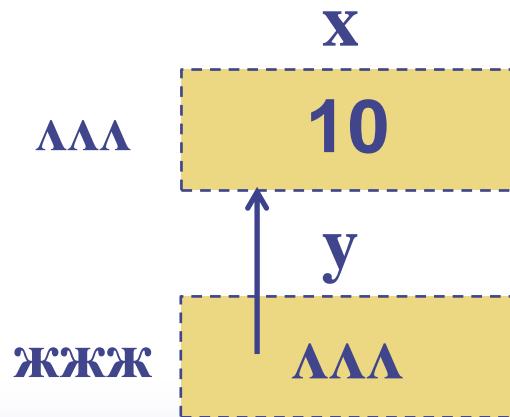
有址引用变量

引用变量

```
int x=10;
```

```
int &y = x;    (1)
```

```
y=20;        (2)
```



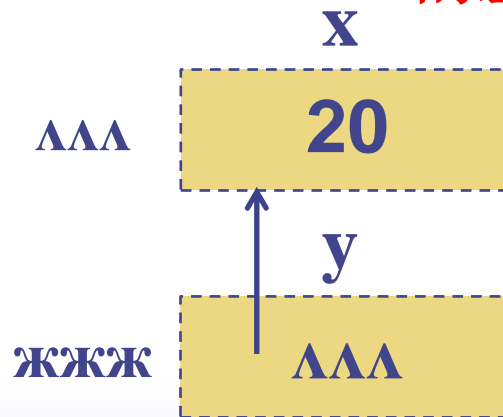
执行(1)后

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?

定义时的 =
与使用时的 =
的差别?





有址引用变量

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

```
y=20;
```

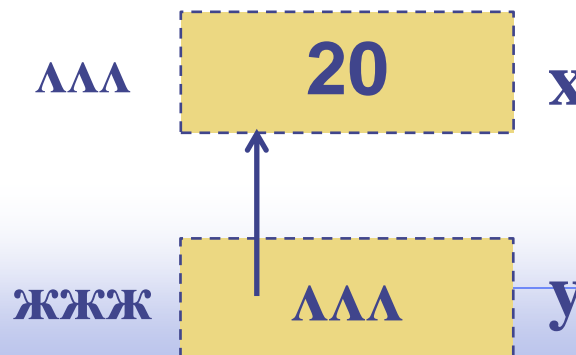
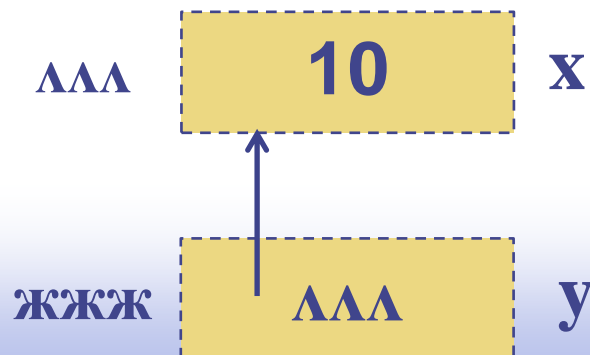
```
t = y;
```

```
lea    eax,[x]  
mov    dword ptr [y],eax;  
  
mov    eax,dword ptr [y];  
mov    dword ptr [eax],14h  
  
mov    eax,dword ptr [y]  
mov    ecx,dword ptr [eax]  
mov    dword ptr [t],ecx
```

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

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

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

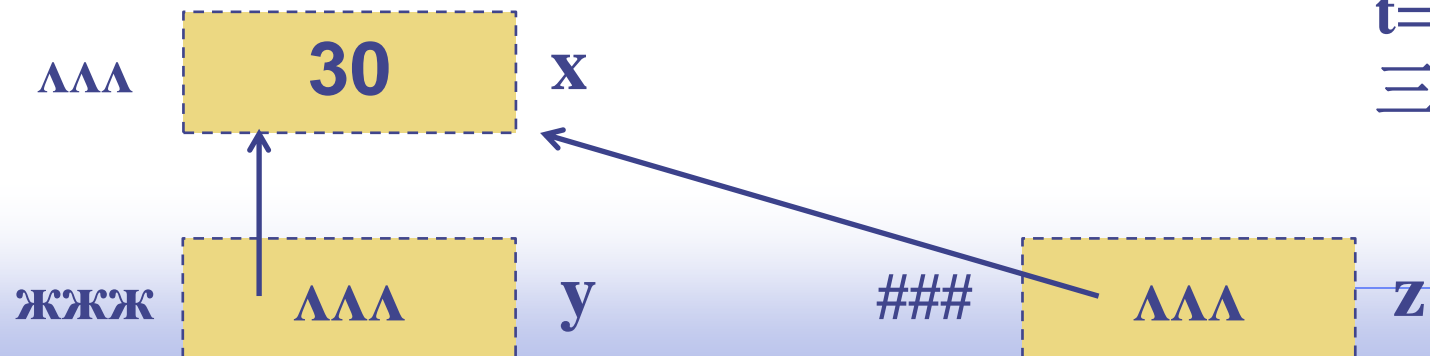


有址引用变量

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

```
int &z =y;      mov eax,dword ptr [y]
                mov  dword ptr [z],eax
```

```
z=30;          mov  eax,dword ptr [z]
                mov  dword ptr [eax],1Eh;
```



```
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

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 &”
```

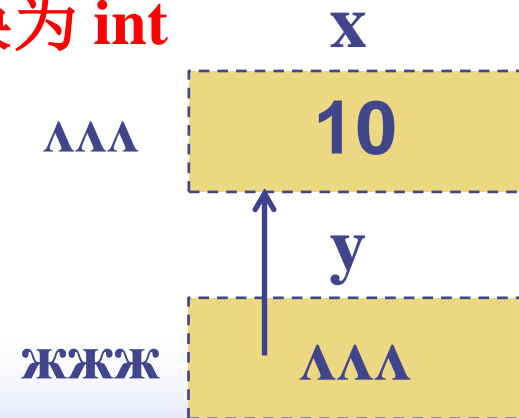
```
int &y=(int *)malloc(sizeof(int));
```

// 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

总结:

- 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

Q: 既然使用引用变量，操作对象都是被引用的变量，
那么定义 引用变量有什么意义？





有址引用变量

引用参数

交换两个整型变量的值的函数

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

```
int a, b;
swap(a,b);
```

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

看汇编代码：观察引用参数传递的是什么？





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);
```

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





有址引用变量

引用参数

```
swap(int *x, int *y)
{
    .....
}

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

```
swap(int &x, int &y)
{
    .....
}

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

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

引用参数

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

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

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

```
void print_info2(struct 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 & f( ) {  
    int t=25;  
    return t;  
}
```

```
lea    eax,[t] // 返回地址
```

```
int    a=f();  
call   f  
mov     eax,dword ptr [eax]  
mov     dword ptr [ebp-0Ch],eax
```

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

```
mov     eax,dword ptr [t]  
// 返回值
```

```
int    a=f();  
call   f  
mov     dword ptr [ebp-0Ch],eax
```

两者都显示 a=25

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=25;  
    return t;  
}
```

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

类比:

```
int x=10;  
int &y = x; // y中内容是x的地址  
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
```

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)`



有址引用变量

返回结果为引用

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

```
int x = f();
```

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

```
int y = g();
```

```
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;  
}
```

```
    t = t + 10;  
mov    eax,dword ptr [t]  
add    eax,0Ah  
mov    dword ptr [t],eax  
    return t;  
lea    eax,[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

```
char * fcharp()
{
    char t[20];
    strcpy_s(t, "hello");
    cout<<"t is "<<t<<endl;
    return t;
}
```

```
char *pc;
pc=fcharp(); // 此处观察 pc 指向的串为 hello
cout<<"pc is"<<pc<<endl;
```

warning C4172: 返回局部变量或临时变量的地址
运行显示 并非 hello





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, 为什么?

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

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

```
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

```
int x;
```

```
const int p= x;
```

```
*(int *)&p =20;
```

```
(int &)p=20;
```

```
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;
```



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;`

`++x;`

`const int y=10;`

`x++`

`x+3`

`(float)x`

`*(float *)&x`

`*(int *)&y`

`y`

`int &u=x;`

`u`

`const int &v=2;`

`v`





2.3.5 元素、下标及数组

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，而枚举元素有相应的整型常量值；
- 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

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个自己存储。

```
char c[6]="abc";    //sizeof(c)=6, strlen(c)=3,
```

```
char d[ ]="abc";    //sizeof(d)=4,
```

```
const char*p="abc";//sizeof(p)=sizeof(void*)=4,
```

```
    p[0]='a',
```

“abc”看作字符指针

```
“abc”[0]='a',      *("abc"+1)='b'。
```





2.4 运算符及表达式

C++运算符、优先级、结合性





2.5 结构与联合

结构 struct

```
struct Person {  
    char* name;  
    int   birthYear;  
    double salary;  
};
```

```
struct Person {  
    char* name;  
    int   birthYear;  
    double salary;  
} zhang, *p;
```

```
struct Person zhang, *p;  
Person zhang, *p;
```

定义结构类型，定义结构变量，
定义结构类型的同时，定义结构变量。





2.5 结构与联合

```
typedef struct Person {  
    char* name;  
    int   birthYear;  
    double salary;  
} PERSON;
```

```
struct Person zhang, *p;  
Person zhang, *p;  
PERSON zhang, *p;
```

```
typedef struct {  
    char* name;  
    int   birthYear;  
    double salary;  
} PERSON;
```

```
PERSON zhang, *p;
```

用 typedef 定义，PERSON 是类型名，而不是变量名





2.5 结构与联合

结构变量的初始化

```
Person zhang = { NULL, 1995, 100 }, *p;    struct Person {  
                                           char* name;  
                                           int  birthYear;  
                                           double salary;  
                                           };
```

Q: 能否 `Person zhang = { "ZhangSan", 1995, 100 };`

无法从 `const char[9]` 转换为 `char *`; `const char * → char *`

Q: 能否 `Person zhang = { (char *) "ZhangSan", 1995, 100 };`

语法正确，但后面若修改 `zhang` 的 `name` 指向的串，会导致程序崩溃。 `strcpy_s(zhang.name, 6, "xu");`





2.5 结构与联合

结构变量的初始化

```
struct Person {  
    char* name;  
    int   birthYear;  
    double salary;
```

Q: 能否 `char* q=(char *)malloc(10);` };
 `Person zhang = { q, 1995, 100 };`

`zhang.name` 与 `q` 指向相同的空间，当`q`指向的空间释放或修改，或者反过来，都会对另一方产生影响，极不安全，或者程序崩溃（如释放`q`，再释放 `zhang.name`）。

Q: 能否 `char q[10];`
 `Person zhang = {q, 1995, 100};`

问题同上。只是 `q` 空间的分配方式有所变化。
此外，执行 `free(zhang.name)` 会直接崩溃。
因为空间不是分配在 堆中。





2.5 结构与联合

结构变量的初始化

```
Person zhang = { NULL, 1995, 100 }, *p;    struct Person {  
                                           char* name;  
                                           int   birthYear;  
                                           double salary;  
                                           };
```

Q: zhang.name 正确的初始化的方法是什么？

可以先初始化为一个空指针； zhang.name = NULL;
为指针分配要指向的空间；
通过串拷贝等方式，为指针指向的空间赋值。

```
zhang.name = (char *)malloc(10);  
strcpy_s(zhang.name, 10, "xu");
```





2.5 结构与联合

位段 Bit field

交通路口有 红灯、绿灯、黄灯。如何表示出指示灯的状态？

定义三个变量，有何优点，有何缺点？

bool red;	char red;	int red;
bool green;	char green;	int green;
bool yellow;	char yellow;	int yellow;

定义一个变量，有何优点，有何缺点？

```
int trafficlights;
#define LIGHT_RED 0x01
#define LIGHT_GREEN 0x02
#define LIGHT_YELLOW 0x04
enum LIGHT { LIGHT_RED = 0x01,
             LIGHT_GREEN = 0x02, LIGHT_YELLOW = 0x04};
```





2.5 结构与联合

位段 Bit field

在结构体或者联合体中以 位为单位 定义成员变量所占的空间

```
struct LIGHT {  
    int red:1    ;  
    int green:1  ;  
    int yellow:1 ;  
} trafficlights;
```

```
T  
trafficlights.red = 1;  
trafficlights.green= 0;
```





2.5 结构与联合

位段 Bit field

在结构体或者联合体中以 位为单位 定义成员变量所占的空间

```
struct A{  
    int i:3;           //i为位段成员  
    int j:4;           //j为位段成员  
    int k;  
} a;                  // sizeof(a) =8;  
a.i = 6;  
a.j = 9;              // 一个字节中存放的信息
```





2.5 结构与联合

联合 union

```
union Long {  
    char    c;  
    short   s;  
    long    x;  
}a;
```

```
Long b;
```



练习题



华中科技大学

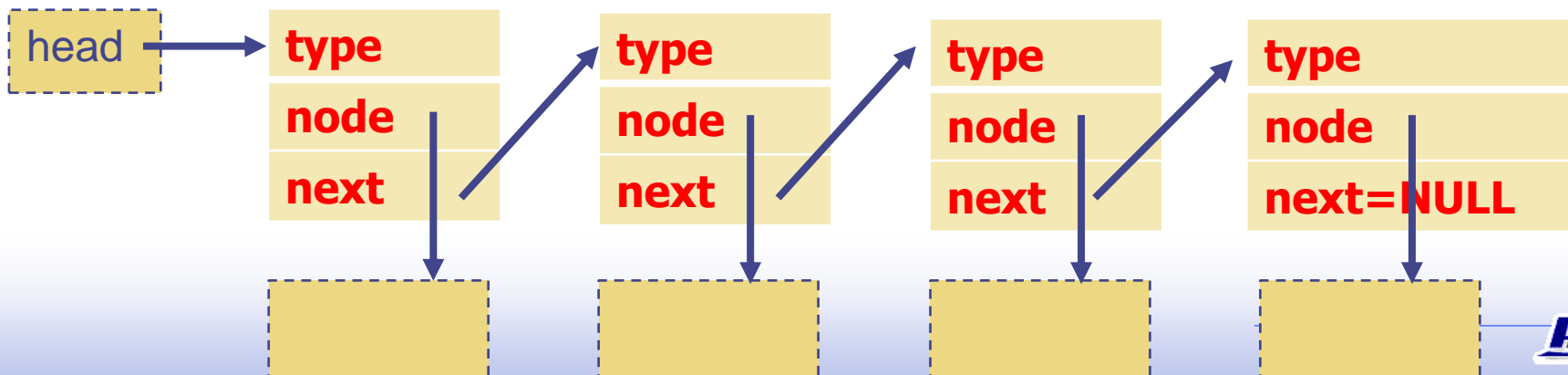
作业 2.8



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



华中科技大学

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() {..... }
```

```
teacher* new_teacher() { ..... }
```

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)
```

