



C++程序设计精要教程

华中科技大学

第2章 类型、常量及变量

◆2.1 C++的单词

单词包括常量、变量名、函数名、参数名、类型名、运算符、关键字等。

关键字也被称为保留字，不能用作变量名。

预定义类型如int等也被当作保留字。

char16_t表示双字节字符类型，支持UTF-16。

char32_t表示四字节字符类型，支持UTF-32。

char16_t x = u'马'; char32_t y = U'马';

wchar_t表示**char16_t**，或**char32_t**。

nullptr表示空指针。

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 [↵]	[↵]

第2章 类型、常量及变量

◆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。假定VS1029采用X86编译模式。
- void: 字节数不定。常表示函数无参或无返回值。
- 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 $-128 \sim 127$ 。
- short: 两字节有符号整数类型, 取值 $-32768 \sim 32767$ 。
- int: 四字节有符号整数类型, 取值 $-2^{31} \sim 2^{31}-1$ 。
- long: 四字节有符号整数类型, 取值 $-2^{31} \sim 2^{31}-1$ 。
- float: 四字节有符号单精度浮点数类型, 取值 $-10^{38} \sim 10^{38}$ 。
- double: 八字节有符号双精度浮点数类型, 取值 $-10^{308} \sim 10^{308}$ 。

注意: 默认一般整数常量当作为int类型, 浮点常量当作double类型。

第2章 类型、常量及变量

◆2.2 预定义类型及值域和常量

- char、short、int、long前可加unsigned表示无符号数。
- long int等价于long; long long占用八字节。
- sizeof(short) ≤ sizeof(int) ≤ sizeof(long) , sizeof(double) ≤ sizeof(long double)。
- 自动类型转换路径: char → unsigned char → short → unsigned short → int → unsigned int → long → unsigned long → float → double → long double。
- 数值零自动转换为布尔值false, 数值非零转换为布尔值true。
- 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);

第2章 类型、常量及变量

预定义类型的数值输出，如：`#include<stdio.h>`后`printf("%d", 4);`;

- double常量：0.9, 3., .3, 2E10, 2.E10, .2E10, -2.5E-10
- char: %c; short, int: %d; long: %ld; 其中%开始的输出格式称为占位符。
- 输出无符号数用u代替d(十进制)，八进制数用o代替d，十六进制用x代替d
- 整数表示宽度如`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”表示左对齐、总宽度8(包括符号位和小数部分)，其中精度为2位小数。
- 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: 指向只读字符的指针即`const char *`，上述“abc”的类型。
- 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。

第2章 类型、常量及变量

◆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。
- 变量定义：描述变量的类型及名称，同时进行初始化。只能定义一次。
- 说明实例：`extern int x; extern int x; //变量可以说明多次`
- 定义实例：`int x=3; extern int y=4; int z; //全局变量z的初始值为0`
- 模块静态变量：使用`static`在函数外部定义的变量。可通过单目 `::` 访问。
- 局部静态变量：使用`static`在函数内部定义的变量。

```
static int x, y; //模块静态变量x、y定义，默认初始值均为0
int main( ) {
    static int y;          //局部静态变量y定义， 初始值y=0
    return  ::y+x+y; //分别访问模块静态变量y，模块静态变量x，局部静态变量
}
```

第2章 类型、常量及变量

◆2.3 变量及其类型解析

- 只读变量：使用 **const** 或 **constexpr** 说明或定义的变量，定义时必须同时初始化。当前程序只能读不能修改其值。
- constexpr** 变量必须用常量表达式初始化，**编译将出现该变量的地方优化为常量。**
- 易变变量：使用 **volatile** 说明或定义的变量，可以后初始化。当前程序没有修改其值，但是变量的值变了。不排除其它程序修改。
- const** 实例：`extern const int x; const int x=3; //定义必须显式初始化x`
- volatile** 例：`extern const int y; volatile int y; //可不显式初始化y，全局y=0`
- 若y=0，语句if(y==2)是有意义的，因为易变变量y可以变为任何值。

第2章 类型、常量及变量

◆2.3 变量及其类型解析

- 在多任务环境下，定义 `const volatile int z=0` 是有意义的，不排出其它程序修改z使其值易变。
- 作为类型修饰符，`const`和`volatile`可以定义函数参数和返回值。
- 保留字 `inline` 用于定义：函数返回值。
- `inline`函数外部变量的作用域和`inline`函数外部静态变量一样，都是局限于当前代码文件的，相当于默认加了`static`。
- 用 `inline` 定义的内容，不能被优化。

第2章 类型、常量及变量

指针及其类型理解

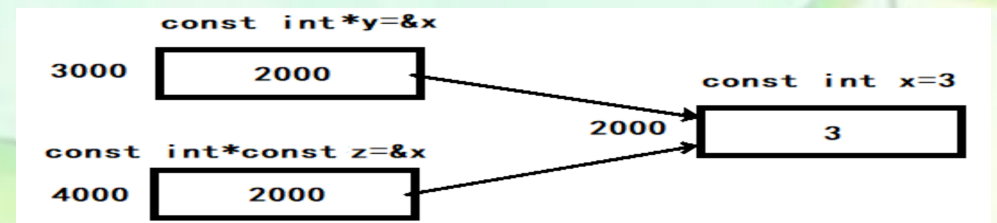
- 指针类型的变量使用*说明和定义，例如：int x=0; int *y=&x;。
- 指针变量y存放的是变量x的地址，&x表示获取x的地址运算，表示y指向x。
- 指针变量y涉及两个实体：变量y本身，y指向的变量x。
- 变量x、y的类型都可以使用const、volatile以及const volatile修饰。

const int x=3; //不可修改x的值

const int *y=&x; //可以修改y的值，但是y指向的const int实体不可修改

const int *const z=&x; //不可修改z的值，且z指向的const int实体也不可改

- x的值为3，地址为2000。
- y和z的值均为2000，表示y和z都指向x。
- y可被修改指向别的变量，但z不行。



第2章 类型、常量及变量

指针及其类型理解

- 指针变量还可以指向指针变量，从而形成多重指针。
- 例如：`short *p=&a`定义p是一个指针变量
`short **r=&p`定义r是一个双重指针变量
变量r存储的是p的地址00001028
- VS2019在X86编译模式下，使用4个字节表示地址
- 根据表2.7有关*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，如下图所示：



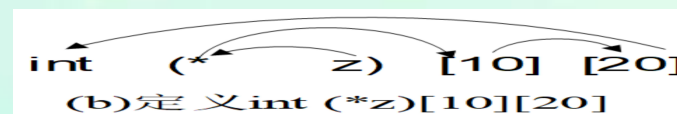
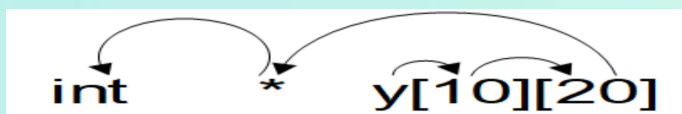
- *r取r指向的单元的值00001020(a的地址)，故**r才能取a的值1，而**r=7修改a

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

第2章 类型、常量及变量

指针及其类型理解

- 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。
- 如：int *y[10][20]；在y的左边是*，右边是[10]，据表2.7知[]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
- 但括号()可提高运算符的优先级，如：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。
- 其第(1)个解释应为z是一个指针，注意z与y的第(1)个解释的不同。
- 指针移动：y[m][n]+1移动到下一整数，z+1移动到下一10*20整数数组。

第2章 类型、常量及变量

指针使用注意事项

- 所指单元值只读的指针(地址)不能赋给所指单元值可写的指针变量。

- 例如: `const int x = 3` (等价 `int const x = 3`) ;

`const int *y = &x; //对`

`int *z = y; //错: y是所指单元值只读的指`

针

`z = &x; //错: &x是所指单元值只读的地址`

证明: (1)假设 `int *z=&x` 正确 (应用反正法证明)

(2)由于 `int *z` 表示 `z` 指向的单元可写, 故 `*z=5` 是正确的

(3)而 `*z` 修改的实际是变量 `x` 的值, `const int x` 规定 `x` 是不可写的。

矛盾。

- 所指单元值可写的指针(地址)能赋给所指单元值只读的指针变量: `y=z`;

- 所指单元值易变的指针(地址)不能赋给所指单元值可写的指针变量, 反之成立。即将前例的 `const` 换成 `volatile` 或者 `const volatile` 结论一样。

第2章 类型、常量及变量

`const int (*)` ↔ `int const (*)`

`int y = 0;`

`const int x = 1;` ↔ `int const x = 1;`

`const int *p = &x;` ↔ `int const *p = &x;`

`const int x; x = 1; //?`

✗

`const int *p; p = &y; //?`

✓

`int *const q1 = &x; //?`

✗

`int *const q1 = &y; //?`

✓

`int *const q1; q1 = &y; //?`

✗

`const int *const q2 = 0; //?`

✓

`const int *const q2 = &x; //?`

✓

`const int *const q2 = &y; //?`

✓

`const`、`volatile`、`int`,任意组合是等价的

`volatile int a = 0;`

`while(a == 0);`

`volatile const int b1 = 0;`

`volatile int const b2 = 0;`

`const volatile int b1 = 0;`

`const int volatile b2 = 0;`

`int const volatile b1 = 0;`

`int volatile const b2 = 0;`

`int b = 0;`

`volatile int *p1 = &b; //?`

✓

`int *p2 = &a; //?`

✗

//p1指向1个volatile int对象（可能会被其他线程改变），但其他线程不可能改变这个对象b。

//p2指向1个int对象（不会被其他线程改变），但实际上这个对象可能随时会被其他线程改变。

第2章 类型、常量及变量

指针使用注意事项

- void *p表示p指向的存储单元的字节数可以是任何大于0的整数。
- 因此，任何类型的存储单元的地址(指针) 都可以赋值给p。

int x=3; p=&x;

double y=4; p=&y;

- 同理可知：delete <操作数>; //该操作数一定是void *类型，因为它可接受任何类型的指针或地址。
- 但是：在向p所指的存储单元赋值时，不能修改任意个字节数的值：即“*p=值”是错误的。必须明确指出所修改的存储单元的**类型**：使用**强制类型转换**。
- 例如，*(**int** *)p=5; *(**double** *)p=3.2; //(int *)将p转换为指向4字节整型单元

第2章 类型、常量及变量

引用 是变量的别名，可以用来修饰变量、函数参数、函数返回值。

引用变量 包括：

- 对左值的引用：`[const] 类型名 &变量名 = 左值表达式;`
- 对右值的引用：`[const] 类型名 &&变量名 = 纯右值表达式;`
`const 类型名 &变量名 = 右值表达式;`

- ▶引用变量只能在定义时初始化1次，初始化之后对引用变量的赋值运算是对该引用变量所代表的存储单元进行赋值。
- ▶用左值初始化的引用变量：被引用的对象（内存单元）的别名（引用变量本身不占用内存）。
- ▶用右值初始化的引用变量：编译器为被引用的对象申请一个缓存，该引用变量就是这个缓存的别名（引用变量本身不占用内存）。

第2章 类型、常量及变量

- 理解引用变量

逻辑上：变量的别名（不占用存储单元）

实现上：被编译为指针（占用存储单元）

- 右值引用变量 `&&` 是C++11增加的新特性，主要用于移动语义（将1个对象内的资源迁移到另外1个对象，如移动构造、移动赋值等）。用 `&&` 表示移动语义是一种编程约定（不是语法）。例如，`void copy(A &&a, A &b)` 一般表示移动拷贝，当类A中含有内部缓冲区时，则将对象a的缓冲区移动到对象b中去（对象b内部不需申请新的缓冲区）。

第2章 类型、常量及变量

```
int x = 1; int *p = &x;
```

```
const int y = 2;
```

```
int &q = x;
```

```
int *&q = p;
```

```
const int &q = x;
```

```
const int &q = y;
```

```
const int &q = 0;
```

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

```
const int &&q = y;
```

```
const int &&q = x++;
```

//注意: y本质上是一个左值

//q是x的别名, 可以通过q重新赋值

//q是指针p的别名, 可以通过q修改p所指的内容

//q是x的别名, 但不能通过q修改x的值

//q是y的别名, 不能通过q重新赋值

//申请1个存储单元, 将0保存到该存储单元,

//q是这个存储单元的别名, 不能重新赋值

//申请1个存储单元, 将x的值拷贝到该存储单元,

//q是这个存储单元的别名, q能重新赋值 (但不会改变x)

//错: y本质上是一个左值

//申请1个存储单元, 将x的值拷贝到该存储单元

const int x = 1 是左值还是右值?

本质上是左值, 因为x占有存储单元。

通过特殊方法可以修改x的值: ***(int *)&x = 0;**

第2章 类型、常量及变量

int x = 3;	
const int y = 4;	
volatile int z = 5;	
int &u = x;	//对, 可以 u = 4
int &u = y;	//错, 如果对的话则可以通过u可以改变y
const int &u = x;	//对, 不能通过u改变x
const int &u = y;	//对, 不能通过u改变y
long &u = x;	//错, 等价 long &u = (long)x, 而 (long)x 是右值
int &&u = x;	//错, 只能用右值去初始化
int &&u = x--;	//对, x--是右值
int &&u = --x;	//错, --x是左值
int &&u = y;	//错, y本质上是左值
long &&u = y;	//对, 等价 long &&u = (long)y, 而 (long)y 是右值
int &&u = 3;	//对, u可以重新赋值, 如 u = 1
const int &&u = 3;	//对, 但u不能重新赋值
volatile int &u = x;	//对, 见前几页ppt中的 volatile int *p1 = &b;
int &u = z;	//错, 见前几页ppt中的 int *p2 = &a;

第2章 类型、常量及变量

`int & &u;` //错: 引用变量u去引用无内存的引用, `int &&u;` ?

`int & *v;` //错: 指针v不能指向引用变量 (无内存)

`int *p; int *&v = p;` //对: v是指针p的引用, 可以使用*v, v[1]等

`int x = 3;`

`int &y = x;` //对: y共享x的内存, y=5将使x=5, x=6将使y=6

`int &z = y;` //对: z引用y所引用的变量x, 注意z、y同类型。非z引用y。

`int && *p;` //错: p不能指向没有内存的引用

`int && &q;` //错: `int &&`没有内存, 不能被q引用

`int & &&r;` //错: `int &`没有内存, 不能被r引用。

`int && &&s;` //错: `int &&`没有内存, 不能被s引用

`int &&t[4];` //错: 数组的元素不能为`int &&`: 数组内存空间为0。

`int &s[3];` //错: 数组应当有内存, 而这里数组元素是无内存的引用

`int a[3]; int (&t)[3] = a;` //对: 数组有内存, t是数组a的别名

`const int a[3] = {1, 2, 3}; int (&&t)[3] = a;` //错: a本质上是左值。

`int (&&t)[3] = {1, 2, 3};` //对: {1, 2, 3} 是左值

第2章 类型、常量及变量

- 若函数返回的不是左值引用，则一定是右值；

```
int &&x = printf("abcdefg"); //对：printf()返回无址右值
```

```
int &&a = 2; //对：引用无址右值
```

```
int &&b = a; //错：a是有名有址的（a是1个缓存的别名）
```

```
int && f( ) { return 2; }
```

```
int &&c = f( ); //对：f返回的是无址引用，是无址的
```

- 位段成员是无址（右值）的

```
int &&x = printf("abcdefg"); //对：printf()返回1个右值
```

```
struct A { int a; /*普通成员：有址*/
```

```
int b : 3; /*位段成员：无址*/
```

```
} p = { 1, 2 };
```

```
int &&q = p.a; //错
```

```
int &&r = p.b; //对：位段只能是右值
```


第2章 类型、常量及变量

元素、下标及数组

- 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在前一个的值加1

```
enum WEEKDAY { Sun, Mon, Tue, Wed, Thu, Fri, Sat };  
WEEKDAY w1 = Sun, w2(Mon);           //可用限定名WEEKDAY::Sun, Sun=0,  
mon=1
```

- 也可以为枚举元素指定值，哪怕是重复的整数值。
- 如果使用“enum class”或者“enum struct”定义枚举类型，则其元素必须使用类型名限定限定元素名

```
enum struct RND { e=2, f=0, g, h };    //正确: e=2, f=0, g=1, h= 2  
RND m = RND::h;                       //必须用限定名RND::h  
int n = sizeof(RND::h);                //n=4, 枚举元素实现为整数
```

第2章 类型、常量及变量

元素、下标及数组

- 数组元素按行存储，对于“`int a[2][3] = { {1,2,3}, {4,5,6} };`”，先存第1行再存第2行

a: 1, 2, 3, 4, 5, 6 //第1个元素为`a[0][0]`，第2个为`a[0][1]`，第4个为`a[1][0]`

- 若上述`a`为全局变量，则`a`在数据段分配内存，1,2...6等初始值存放于该内存。
- 若上述`a`为静态变量，则`a`的内存分配及初始化值存放情况同上。
- 若上述`a`为函数内定义的局部非静态变量，则`a`的内存存在栈段分配，而初始化值则在数据段分配，最终函数使用栈段的内存。
- C++数组并不存放每维的长度信息，因此也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`a`代表数组的首地址**，其代表的类型为 `int [2][3]` 或 `int(*)[3]`。

第2章 类型、常量及变量

元素、下标及数组

- 一维数组可看作单重指针，反之也成立。例如：

```
int b[3]; //*(b+1)等价于访问b[1]
```

```
int *p = &b[0]; //*(p+2)等价访问p[2]，也即访问b[2]
```

- 字符串常量可看做以'\0'结束存储的字符数组。例如“abc”的存储为

'a'	'b'	'c'	'\0'
-----	-----	-----	------

//字符串长度即strlen("abc")=3，但需要4个自己

存储。

```
char c[6] = "abc"; //sizeof(c)=6, strlen(c)=3, "abc"可看作字符数组
```

(有问题)

```
char d[ ] = "abc"; //sizeof(d)=4, 编译自动计算数组的大小,  
strlen(d)=3
```

```
const char *p="abc"; //sizeof(p)=sizeof(void*)=4, p[0]='a', "abc"看作字符指  
针
```

第2章 类型、常量及变量

◆2.4 运算符及表达式

- C++运算符、优先级、结合性见表2.7。优先级高的先计算，相同时按结合性规定的计算顺序计算。可分如下几类：
 - 位运算：按位与&、按位或|、按位异或^、左移、右移。左移1位相当于乘以2，右移1位相当于除以2。
 - 算数运算：加+、减-、乘*、除/、模%。
 - 关系运算：大于、大等于、等于、小于、小等于
 - 逻辑运算：逻辑与&&、逻辑或||
- C++逻辑值自动转换为整数0或1，因此， $1 < 2$ 的值为1， $2 < 1$ 的值为0。

第2章 类型、常量及变量

赋值、选择与自增和自减运算

- 赋值语句也是C++一种表达式。对于 `int x(2); x=x+3;` 赋值语句中的表达式：
 - `x+3` 是加法运算表达式，其计算结果为传统右值5。
 - `x=5` 是赋值运算表达式，其计算结果为传统左值 `x` (`x` 的值为5)。
 - 由于计算结果为传统左值 `x`，故还可对 `x` 赋值7，相当于运算：`(x=x+3)=7`；结果为左值 `x`。
- 选择运算使用“?:”构成，例如：`y=(x>0)?1:0;` 翻译成等价的C++语句如下。

```
if(x>0) y=1;  
else    y=0;
```
- 前置运算“++`c`”、后置运算“`c`++”为自增运算；相当于 `c=c+1`，前置运算“--`c`”、后置运算“`c`--”为自减运算，相当于 `c=c-1`。

第2章 类型、常量及变量

赋值、选择与自增和自减运算

- 前置运算先运算后取值，结果为传统**左值**；后置运算先取值后运算，结果为传统**右值**。
 - $x+3$ 是加法运算表达式，其计算结果为传统右值5。
 - $x=5$ 是赋值运算表达式，其计算结果为传统左值 x (x 的值为5)。
 - 由于计算结果为传统左值 x ，故还可对 x 赋值7，相当于运算： $(x=x+3)=7$ ；结果为左值 x 。
- 选择运算使用“?:”构成， 例如： $y=(x>0)?1:0$ ；翻译成等价的C++语句如下。

```
if(x>0) y=1;  
else    y=0;
```
- 当数学表达式的分母或分子有优先级低于除法的运算符如+、-时，在转换为对应的C++表达式时应使用括号括起分母或分子。

第2章 类型、常量及变量

◆2.5 结构与联合

- 结构是使用struct定义的一组数据成员，每个成员都要分配相应的内存。
 - 数据成员可以是基本数据类型如char、int等。
 - 数据成员也可以是复杂的结构或联合成员。
 - 所有成员都可被任意函数访问。
 - 类型不大于int的成员可以定义为使用若干位二进制的位段类型。
- 联合是使用union定义的一组数据成员，所有成员共用最大成员分配的内存。
 - 数据成员可以是基本数据类型、结构或联合类型
 - 所有成员都可被任意函数访问
 - 类型不大于int的成员可以定义为使用若干位二进制的位段类型
- 结构和联合还可以包含函数成员。

第2章 类型、常量及变量

◆2.5 结构与联合

- 在vs2019的x86编译模式下，定义如下结构。

```
struct Person {           //sizeof(Person)=sizeof(const char*)+sizeof(int)
    const char *name;      //定义实例数据成员name，不允许修改name指向的姓名
    int birthYear;         //定义实例数据成员birthYear
} liShiZhen;              //定义Person类型的同时定义该类型的变量liShiZhen
struct Person huaTuo;      //定义Person类型后，struct可省略
```

- 在vs2019的x86编译模式下，定义如下联合。

```
union Long { //自定义union类型Long。成员c、s、x共享内存
    char c;   //c共享x的内存
    short s;  //s共享x的内存
    long x;   //sizeof(Long) = max(sizeof(char), sizeof(short), sizeof(long))=sizeof(long)
} a; Long b;  //修改任意成员，都会同时修改其它成员
```