4.5 共用体/联合体 (union)

它也是一种用户自定义的数据类型, 它的定义与结构体非常相似, 但是它的特点是共用体中的所有成员占用同一段内存空间, 也就是说, 同一时刻， 只有一个成员是有效的, 其他成员都是无效的.

- 共用体占用的空间为最大成员的长度
  - 比如 共用体中有3个成员, 类型分别为 int, long 和 long long, 此时, 共用体的长度为 long long 类型站用的空间
  - 因为我们要保证成员一定能存放.

使用场景

- 如果我们有一个数据, 但是这个数据的类型可能是 int, 也可能是 long, 也可能是 double, 那么我们就可以使用共用体来定义这个数据, 这样就可以节省内存空间了. --> 一般在嵌入式编程中会用到
- 可以用联合体来查看计算机属于大端模式还是小端模式

共用体的定义格式如下:

```
union 共用体名
{
    成员列表;
};
```

如:

```
union one4all
{
    int int_val;        // 这3个成员中, 在使用的时候只有1个成员是有效的, 如果用了 int_val, long_val 和 double_val 就都不能用了
    long long_val;
    double double_val;
};
```

代码例子1 --> 共用体的定义和用法 (注意看关于内存使用分析)

```
#include <iostream>

using namespace std;
```

```
// 对于下面的联合体，它有两个成员变量 ch 和n，占用的空间为最大的成员变量的空间，也就是4个byte，
//  如果是结构体的话就是所有成员变量的空间之和，为5个byte
union one2all           // 使用 union 关键字
{
    char ch;            // 成员变量也是语句，不要漏了分号
    int n;
};                      // 这是一个语句，不要漏了分号

int main(void)
{
    one2all num;
    cout << "Sizeof(num): " << sizeof(num) << endl; // 输出4

    cout << "现在给 num 的 ch 成员进行赋值\n";
    num.ch = 'A';           // 激活了 ch 成员
    cout << "num.ch: " << num.ch << endl;       // 输出 A
    cout << "num.n: " << num.n << endl;         // 输出随机数（说明了 ch 和 n 共用了同一块内存，但是ch
只占用了一个byte，还有3个byte没有初始化，所以n的值看上去是随机的）

    cout << "\n\n现在给 num 的 n 成员进行赋值\n";
    num.n = 97;           // 激活了 ch 成员
    cout << "num.n: " << num.n << endl;         // 输出97
    cout << "num.ch: " << num.ch << endl;       // 输出 a，（说明了 ch 和 n 共用了同一块内存，n占用了4
个 byte，把整块内存用完了，所以 ch 直接读取了 n 的值，也就是97，97对应的ASCII码就是a）

    return 0;
}
```

结构体中嵌套共用体 (注意看匿名共用体的部分)

代码例子1 --> 普通的嵌套使用

```
#include <iostream>
using namespace std;

strcut widget
{
    char brand[20];
    int type;
    union id                // 结构体的定义中，嵌套定义了共用体
    {
        long id_num;
        char id_char[20];
    } id_val;               // 定义的同时，创建了一个 id_val 的对象，这样，在结构体成员访问的时候就可以直
接使用了
};

int main(void)
{
    widget price;
    if (price.type == 1)
        cin >> price.id_val.id_num;         // 需要使用两次成员变量访问符号
    else
        cin >> price.id_val.id_char;        // 需要使用两次成员变量访问符号
}
```

代码例子2 --> 使用匿名的共用体

```
#include <iostream>
using namespace std;

struct widget
{
    char brand[20];
    int type;
    union                           // 这里没有写成员名，就是匿名的共用体，
    {
        long id_num;
        char id_char[20];
    };                              // 注意，此时没有初始化联合体对象，这时候，id_num 和 id_char 被直接认为是结构体成员
};

int main(void)
{
    widget price;
    if (price.type == 1)
        cin >> price.id_num;          // 只需要使用一次成员变量访问符号
    else
        cin >> price.id_char;         // 只需要使用一次成员变量访问符号
}
```

4.6 枚举类型 (enum, enumeration 的缩写)

- 枚举类型的也是一种用户自定义的数据类型, 但它最常见的目的并不是为了自定义数据类型, 而是为了定义符号常量.
    - 这样就不用像const那样一个个去定义符号常量了, 比较方便, 但是枚举类型对符号常量的取值有限制: 只能是整型, 且是从0开始的， 连续的整型数字, 没法直接定义出一个浮点型的符号常量.

4.6.1 枚举类型的定义方式是:

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
    // 枚举类型里, red, orange这些值对应着"从0开始的, 连续的整型数字", 依次递增
```

- 上面代码一共完成了2个步骤:
    1. 让specturm 成为新类型的名称

2. 将red, orange, yellow, green, blue, violet, indigo, ultraviolet 这些符号常量关联到这个新类型上, 它们对应整数值 (0, 1, 2, 3, 4, 5, 6, 7), 这些常量被称为枚举量(enumerator)

4.6.2 枚举类型的特殊属性:

1. **在没有进行强制类型转换** 的前提下, 枚举类型的赋值只能在枚举量之间进行, 不能赋值给其他类型的变量.
    - 例如: `spectrum color = red;` 是可以的, 但是 `spectrum color = 1;` 是不可以的, 赋值的时候只能使用上面 spectrum 指定的8个枚举量
        - 但是有些编译器在这种情况下是可以通过编译的, 它可能只弹一个 warning, 但是这种赋值操作在传统的C++中是错的, 为了**保证代码可移植性**, 尽量不要这么做, 因为有些平台能跑通, 有些平台会报错.
2. 枚举类型没有定义算术运算, 不允许做算术运算.
3. 枚举量是整型, 可以被提升为 `int` 类型, 但是 int 类型不能自动转换为枚举类型.

- 例如:

```
spectrum band = orange;
++orange;                           // 报错, 枚举量不能进行算术运算
band = orange + red;                // 报错, 枚举量不能进行算术运算
band = int(orange) + int(red);      // 报错, 虽然转成int之后可以进行计算, 但是int不能赋值给band, 因为int类型比枚举类型高, 枚举类型赋值给int是可以的, 但是 int 不能赋值给枚举类型

//int 类型不能自动转换为 枚举类型
int  color = blue;  // 可以, blue 是枚举量, 自动转成整型
band = color;       // 报错, int 类型不能自动转换为 枚举类型
color = 3 + red;    // 可以, red 是枚举量, 自动转成整型再与3相加, 结果赋给整型 color

// 如果整型是一个有效的枚举量, 则可以通过强制类型转换对枚举类型赋值
band = spectrum(3); // 可以, 3是一个有效的枚举量, 通过强制类型转换, 赋值给枚举类型 band

//如果整型不是一个有效的数据类型, 得到的结果是 **不确定** 的
band = spectrum(40003);  // 40003 不在枚举范围内, 不同的编译器可能会有不同的结果, 但是都是不确定的
```

4.6.3 指定枚举类型的值

1. 显示地指定枚举量的值(必须是整数)

```
// 指定所有的枚举量的值
enum bits {one = 1, two = 2, four = 4, eight = 8};
// 指定部分枚举量的值, 其他的枚举量的值会在前一个枚举量的值的基础上递增
```

```
enum bigstep {first, second = 100, third};  // first = 0, second = 100, third = 101
// 同时创建多个值相同的枚举量
enum {zero, null = 0, one, numero_uno = 1}; // zero 和 null 都是0, one 和 numero_uno 都是1
```

4.6.3.1 (已定义的枚举量的) 枚举量取值的上下限

前面提到, 为了保证代码的可移植性, 不要通过强制类型转换, 将未定义的枚举量复制给枚举变量, 但是, 在新版的C++中已经允许这么做了 (但仍不确保代码会移植到低版本C++中), 此时就涉及到枚举量的合法取值范围.

- 枚举量的合法取值范围确定方法是:
  1. 确定上界: 找到枚举量中最大的枚举量, 然后计算出"比这个枚举量大的, 最靠近这个枚举量的2次幂", 这个二次幂减1 就是取值的上界.
     - 例如: 已经定义的枚举类型的最大枚举量是10, 最靠近10的那个2次幂是16 (2的3次方是8, 4次方是16, 要求比10大, 所以16就是最靠近的2次幂), 然后16 - 1 = 15, 所以上界是15.
  2. 确定下界: (1)如果已经定义的枚举类型的枚举量都是非负数, 那么下界就是0; (2) 如果有负数, 那么下界取"比这个枚举量小的, 最靠近这个枚举量的2次幂 加1".
     - 针对(2), 例如: 最小的枚举量为 -6, 最接近-6且比它小的2次幂为-8, 那么下界就是 -8 + 1 = -7.

第十章还会有更详细的枚举类型讲解