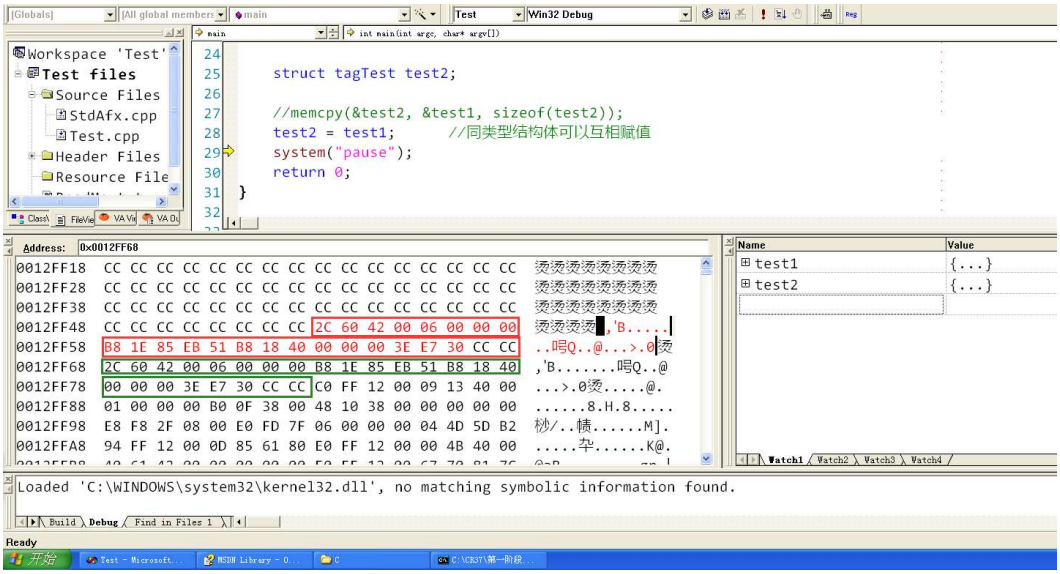
2020/04/28_第19课_结构体、共用体、枚举的使用

笔记本:	C
创建时间:	2020/4/28 星期二 15:12
作者:	ileemi
标签:	共用体, 结构体, 枚举

结构体

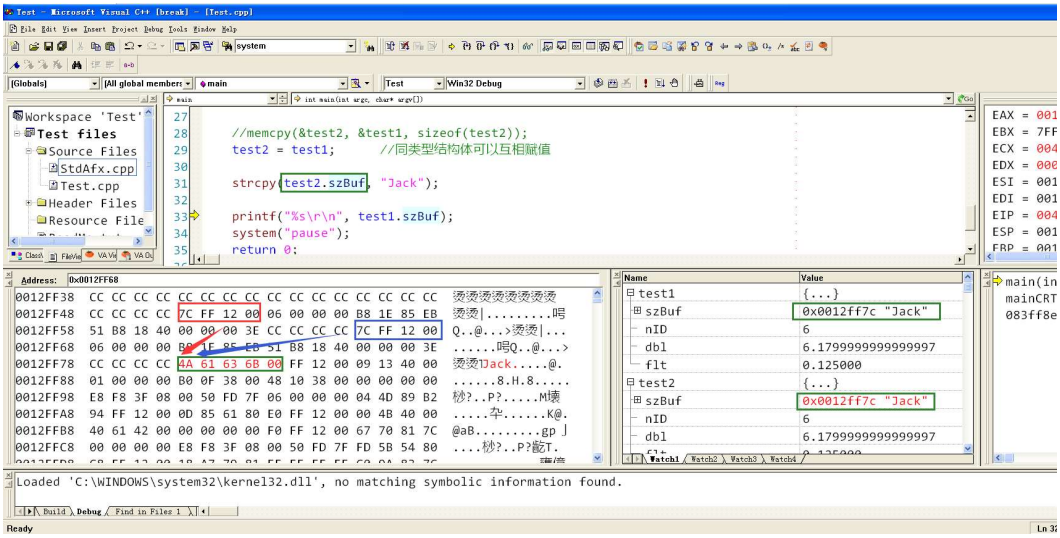
> 同类型结构体可以互相赋值

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

struct tagTest
{
    char *szBuf;
    int nID;
    double dbl;
    float flt;
};
int main(int argc, char* argv[])
{
    struct tagTest test1 = {
        "Tom",
        6,
        6.18,
        0.125f
    };

    struct tagTest test2;

    //memcpy(&test2, &test1, sizeof(test2));
    test2 = test1;      //同类型结构体可以互相赋值，完全等价memcpy
    system("pause");
    return 0;
}
```




```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct tagTest
{
    char *szBuf;
    int nID;
    double dbl;
    float flt;
};
int main(int argc, char* argv[])
{
    char szBuf[] = "Tom";
    struct tagTest test1 = {
        szBuf,
        6,
        6.18,
        0.125f
    };

    struct tagTest test2;

    //memcpy(&test2, &test1, sizeof(test2));
    test2 = test1;      //同类型结构体可以互相赋值

    /*
    修改结构体test2中的成员szBuf后，test1中的成员szBuf也会受影响
    两者指向的是同一个地方
    */
    strcpy(test2.szBuf, "Jack");
```

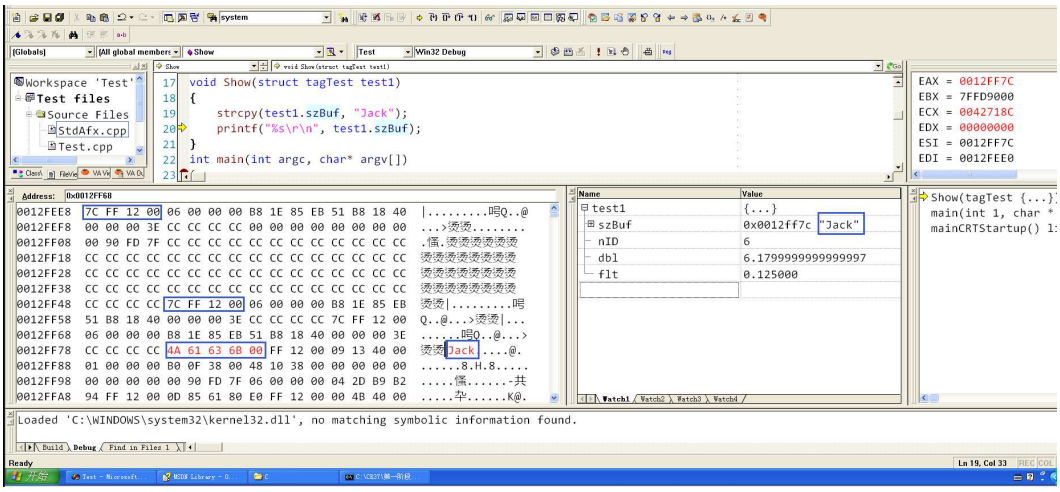
```
    printf("%s\r\n", test1.szBuf);  //Jack
    system("pause");
    return 0;
}
```



函数参数的传递，无论任何类型都是从实参->形参，进行传递的

函数参数的传递，传递指针，传递的就是指针的一份拷贝，传递结构体，就是结构体的一份拷贝，传递其它类型的数据，就是其它类型的一份拷贝。
当结构体类型做函数参数传递的时候：

```
void Show(struct tagTest test1)
{
    strcpy(test1.szBuf, "Jack");
    printf("%s\r\n", test1.szBuf);
}
```



两个结构体变量同时操控同一个文件或者同一块的内存区域

```
struct tagTest test2;

//memcpy(&test2, &test1, sizeof(test2));
```

```
    /*
    如果存在指针成员，这时候赋值了指针的值，而不是指针目标的值（浅拷贝）
    */
    test2 = test1;
```

避免两个结构体变量中的指针成员同时引用同一个资源（一个指针成员修改指向数据内容，另一个指针也会受到影响）的方法：
1、有条件的情况，可以让每个结构体变量中的指针成员各自拥有独立的资源。**深拷贝**（好比篮球场上大伙抢一个篮球，一个人把篮球搞坏后，其他人不能玩，如果有条件下每个人都有一个篮球，就不会收到改影响），将结构体成员指针更改为数组。
2、引用计数，所有结构体变量中的指针成员同时引用同一个资源，但是需要记录引用的情况，满足合理条件则真正处理改资源。

微软系统大量使用 引用计数方法，很多情况下没有空间条件

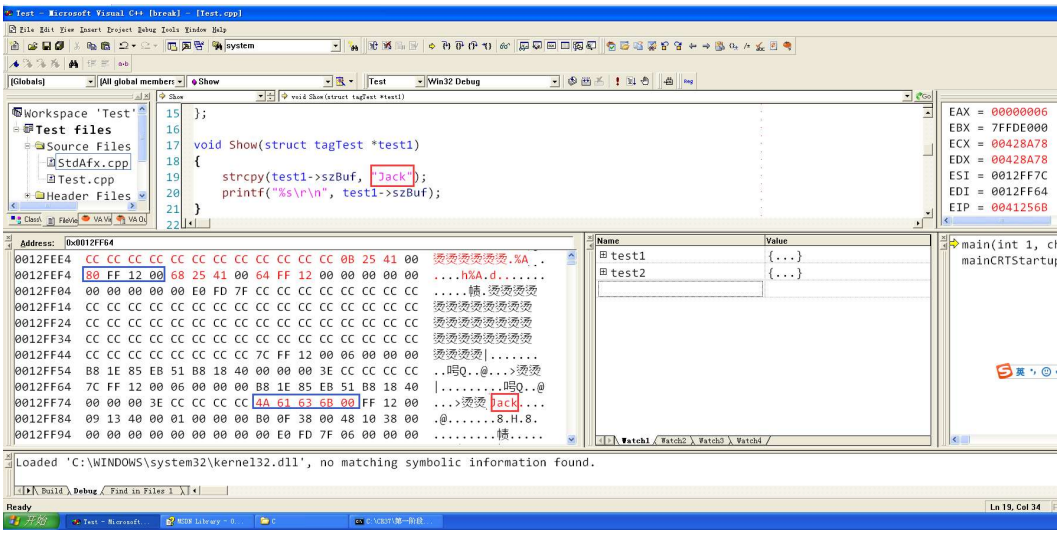
好比Windows下的可执行文件装载到内存创建进程，可执行文件需要引用 kernel32.dll(Windows中非常重要的32位动态链接库文件，属于内核级文件) ，uese32.dll gdi32.dll 引用kernel32.dll

kernel32.dll ->(引用) ntdll.dll
uese32.dll ->(引用) kernel32.dll
gdi32.dll ->(引用) kernel32.dll
...

将函数的参数设置为结构体指针，可以解决浅拷贝带来的问题，间接访问，直接修改实参

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

struct tagTest
{
    char *szBuf;
    int nID;
    double dbl;
    float flt;
};

void Show(struct tagTest *test1)    //传递时复制的是指针（4字节）
{
    /*
    间接访问，直接修改实参
    -> 符号，有间接访问，导致直接访问并修改实参
    */
    strcpy(test1->szBuf, "Jack");
    printf("%s\r\n", test1->szBuf);
}
```

```c
int main(int argc, char* argv[])
{
    char szBuf[] = "Tom";
    struct tagTest test1 = {
        szBuf,
        6,
        6.18,
        0.125f
    };

    struct tagTest test2;

    //memcpy(&test2, &test1, sizeof(test2));
    test2 = test1;      //同类型结构体可以互相赋值

    /*
    修改结构体test2中的成员szBuf后，test1中的成员szBuf也会受影响
    两者指向的是同一个地方
    */
    //strcpy(test2.szBuf, "Jack");

    //printf("%s\r\n", test1.szBuf);  //Jack

    Show(&test1);    //实参->形参
    printf("%s\r\n", test1.szBuf);
    system("pause");
    return 0;
}
```



给函数传递一个指向结构体的指针，远比传递一个结构体要小的多，将其压到堆栈上的效率会提高很多，传递指针在访问结构体成员的时候，必须使用间接访问来访问，结构体越大，把指向它的指针传递给函数的效率就越高。

函数的参数参数为数组名时，传递的是数组的首地址
函数的参数参数为结构体时，传递的是结构体中所有成员的引用

类型定义不占用空间，不产生编译器行为，写给编译器看的

产生变量的时候，产生编译器行为，编译器为其分配空间

结构体的嵌套，当子结构体仅为主结构服务时，可将其至于主结构内，也可将其设为私有。

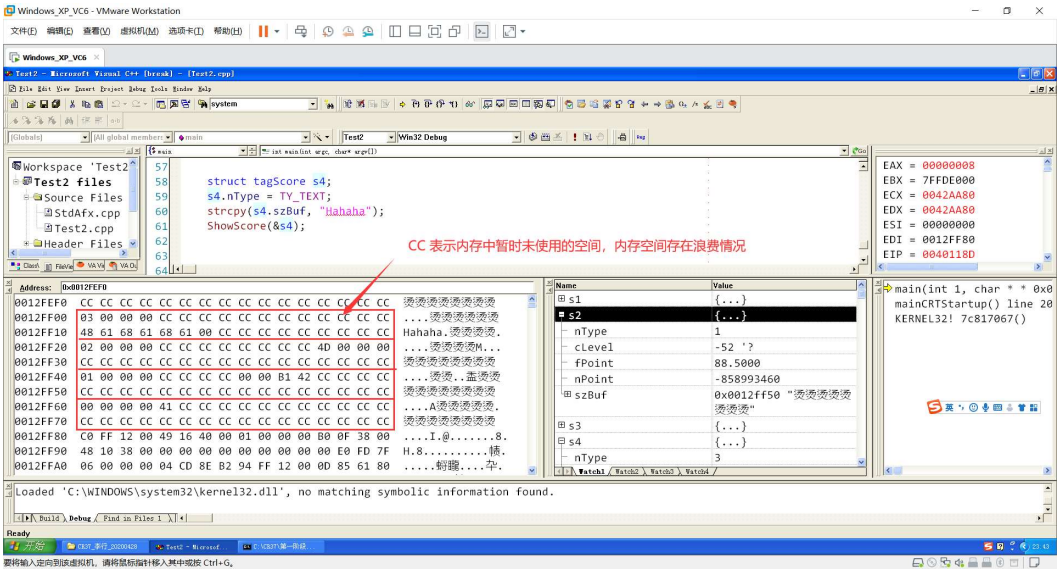
```
#include "stdafx.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct tagTest
{
    char *szBuf;
    int nID;
    struct tagDateOfBirth
    {
        int nYear;
        int nMonth;
        int nDay;
    }Dob;
    double dbl;
    float flt;
};

int main(int argc, char* argv[])
{
    struct tagTest test1 = {
        "Tom"
    };
    //struct tagDateOfBirth dob;  //未定义
    struct tagTest :: tagDateOfBirth dob = {
        2020, 4, 28
    };
    test1.Dob.nMonth = 5;
    dob.nMonth = 5;
    dob.nDay = 1;

    printf("%d %d %d\r\n", dob.nYear, dob.nMonth, dob.nDay);

    system("pause");
    return 0;
}
```

为了分类管理程序，可以使用结构体嵌套，调用对用的成员结构时，需要注意层级关系，可使用 :: 。

结构体内的子结构 不定义类型名时，这个子结构就只能用于父结构私有

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

#define TY_CHAR     0
#define TY_FLOAT    1
#define TY_INT      2
#define TY_TEXT     3
struct tagScore
{
    int nType;      //标记

    char cLevel;
    float fPoint;
    int nPoint;
    char szText[16];
};

void ShowScore(struct tagScore *pScore)
{
    switch (pScore->nType)
    {
    case TY_CHAR:
        printf("%c\r\n", pScore->cLevel);
        break;
    case TY_FLOAT:
        printf("%f\r\n", pScore->fPoint);
        break;
    case TY_INT:
        printf("%d\r\n", pScore->nPoint);
        break;
    case TY_TEXT:
        printf("%s\r\n", pScore->szBuf);
        break;
    }
}
int main(int argc, char* argv[])
{
    struct tagScore s1;
    s1.nType = TY_CHAR;
    s1.cLevel = 'A';
    ShowScore(&s1);

    struct tagScore s2;
    s2.nType = TY_FLOAT;
```

```
    s2.fPoint = 88.5f;
    ShowScore(&s2);

    struct tagScore s3;
    s3.nType = TY_INT;
    s3.nPoint = 77;
    ShowScore(&s3);

    struct tagScore s4;
    s4.nType = TY_TEXT;
    strcpy(s4.szBuf, "Hahaha");
    ShowScore(&s4);

    system("pause");
    return 0;
}
```



Version 2:

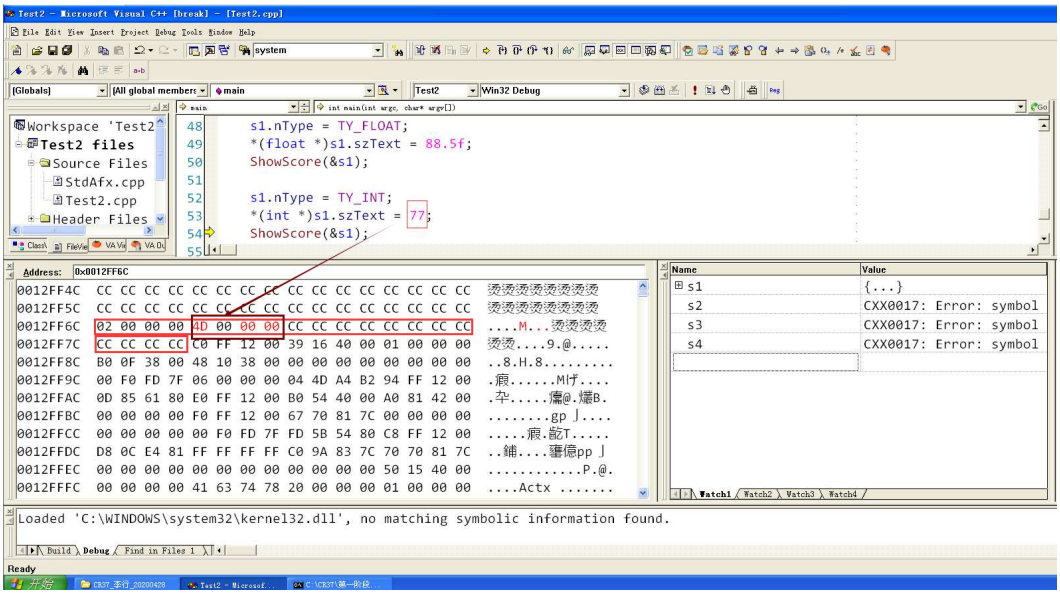
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TY_CHAR     0
#define TY_FLOAT    1
#define TY_INT      2
#define TY_TEXT     3
struct tagScore
{
    int nType;      //标记

//      char cLevel;
//      float fPoint;
```

```
//     int nPoint;
    char szText[16];
};

void ShowScore(struct tagScore *pScore)
{
    switch (pScore->nType)
    {
    case TY_CHAR:
        printf("%c\r\n", *(char *)pScore->szText);
        break;
    case TY_FLOAT:
        printf("%f\r\n", *(float *)pScore->szText);
        break;
    case TY_INT:
        printf("%d\r\n", *(int *)pScore->szText);
        break;
    case TY_TEXT:
        printf("%s\r\n", pScore->szText);
        break;
    }
}
int main(int argc, char* argv[])
{
    struct tagScore s1;
    s1.nType = TY_CHAR;
    *(char *)s1.szText = 'A';
    ShowScore(&s1);

    struct tagScore s2;
    s2.nType = TY_FLOAT;
    *(float *)s2.szText = 88.5f;
    ShowScore(&s2);

    struct tagScore s3;
    s3.nType = TY_INT;
    *(int *)s3.szText = 77;
    ShowScore(&s3);

    struct tagScore s4;
    s4.nType = TY_TEXT;
    strcpy(s4.szText, "Hahaha");
    ShowScore(&s4);

    system("pause");
    return 0;
}
```

Version 2.1:

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

#define TY_CHAR     0
#define TY_FLOAT    1
#define TY_INT      2
#define TY_TEXT     3
struct tagScore
{
    int nType;      //标记

//     char cLevel;
//     float fPoint;
//     int nPoint;
    char szText[16];
};

void ShowScore(struct tagScore *pScore)
{
    switch (pScore->nType)
    {
    case TY_CHAR:
        printf("%c\r\n", *(char *)pScore->szText);
        break;
    case TY_FLOAT:
        printf("%f\r\n", *(float *)pScore->szText);
        break;
    case TY_INT:
        printf("%d\r\n", *(int *)pScore->szText);
        break;
    case TY_TEXT:
        printf("%s\r\n", pScore->szText);
        break;
    }
}
int main(int argc, char* argv[])
{
    struct tagScore s1;
    s1.nType = TY_CHAR;
    *(char *)s1.szText = 'A';
    ShowScore(&s1);

    s1.nType = TY_FLOAT;
```

```
    *(float *)s1.szText = 88.5f;
    ShowScore(&s1);

    s1.nType = TY_INT;
    *(int *)s1.szText = 77;
    ShowScore(&s1);

    s1.nType = TY_TEXT;
    strcpy(s1.szText, "Hahaha");
    ShowScore(&s1);

    system("pause");
    return 0;
}
```

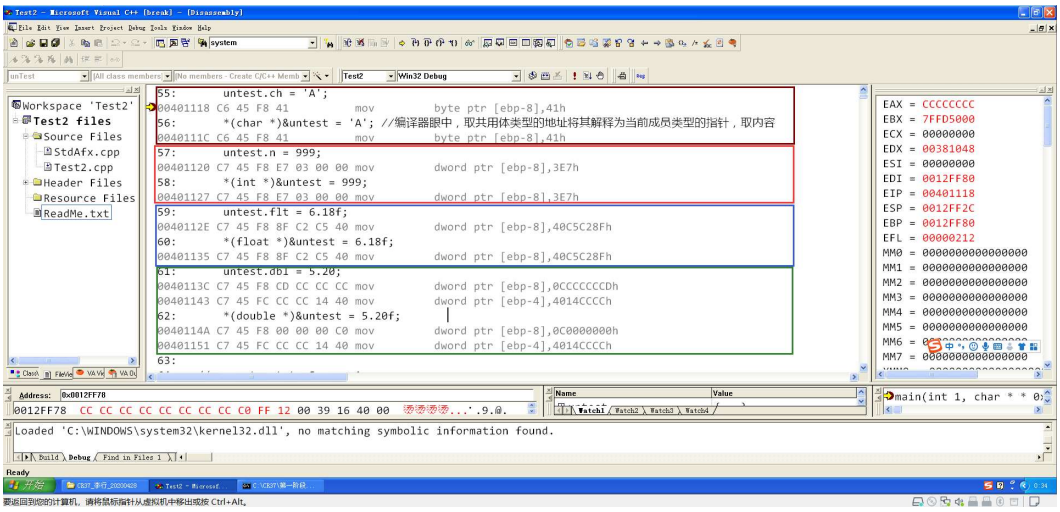


union

C语言提供一种数据存储类型：语法糖 union

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

#define TY_CHAR     0
#define TY_FLOAT    1
#define TY_INT      2
#define TY_TEXT     3

//语法糖
//选择其所有成员中块头最大的类型字节数，作为其实际的占用空间
union unTest
```

```
{
    char ch;
    int n;
    float  flt;
    double dbl;
};
int main(int argc, char *argv[])
{
    union unTest untest;
    untest.ch = 'A';
    /*
    编译器眼中，取共用体类型的地址将其解释为当前成员类型的指针，取内容
    */
    *(char *)&untest = 'A';

    untest.n = 999;
    *(int *)&untest = 999;

    untest.flt = 6.18f;
    *(float *)&untest = 6.18f;

    untest.dbl = 5.20;
    *(double *)&untest = 5.20f;

    system("pause");
    return 0;
}
```



使用共用体对上面的程序再加以修改：

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

#define TY_CHAR     0
```

```c
#define TY_FLOAT    1
#define TY_INT      2
#define TY_TEXT     3

struct tagScore
{
    int nType;      //标记

    union           //私有，外接无法访问使用
    {
        char cLevel;
        float fPoint;
        int nPoint;
        char szText[16];
    }uscore;
};

void ShowScore(struct tagScore *pScore)
{
    switch (pScore->nType)
    {
    case TY_CHAR:
        printf("%c\r\n", pScore->uscore.cLevel);
        break;
    case TY_FLOAT:
        printf("%f\r\n", pScore->uscore.fPoint);
        break;
    case TY_INT:
        printf("%d\r\n", pScore->uscore.nPoint);
        break;
    case TY_TEXT:
        printf("%s\r\n", pScore->uscore.szText);
        break;
    }
}

int main(int argc, char *argv[])
{
    struct tagScore s1;
    s1.nType = TY_CHAR;
    s1.uscore.cLevel = 'A';
    ShowScore(&s1);

    s1.nType = TY_FLOAT;
    s1.uscore.fPoint = 88.5f;
    ShowScore(&s1);
```

```
    s1.nType = TY_INT;
    s1.uscore.nPoint = 77;
    ShowScore(&s1);

    s1.nType = TY_TEXT;
    strcpy(s1.uscore.szText, "Hahaha");
    ShowScore(&s1);

    system("pause");
    return 0;
}
```

enum

仅仅只能代替符号化**整数的宏**，符号化整型常量
会给其成员（枚举常量）安排一个唯一的整数值

可以说是一个功能单一，受控制的宏。

在定义枚举变量时，它的值可以赋值，在不进行强转的情况下只能赋值枚举常量值，编译器会做类型检查

```
    ...
    enum eType
    {
        TY_CHAR,
        TY_FLOAT,
        TY_INT,
        TY_TEXT
    };

    enum eType etype;
    etype = 0;  //编译不通过
    etype = TY_CHAR;
    ...
```

使用枚举对上面的例子再次进行改进：

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

// #define TY_CHAR     0
// #define TY_FLOAT    1
// #define TY_INT      2
```

```
// #define TY_TEXT     3

struct tagScore
{
    int nType;      //标记

    union           //私有
    {
        char cLevel;
        float fPoint;
        int nPoint;
        char szText[16];
    }uscore;

    enum            //私有
    {
        TY_CHAR,
        TY_FLOAT,
        TY_INT,
        TY_TEXT
    }etype;
};

//输出数据
void ShowScore(struct tagScore *pScore)
{
    switch (pScore->nType)
    {
    case tagScore::TY_CHAR:
        printf("%c\r\n", pScore->uscore.cLevel);
        break;
    case tagScore::TY_FLOAT:
        printf("%f\r\n", pScore->uscore.fPoint);
        break;
    case tagScore::TY_INT:
        printf("%d\r\n", pScore->uscore.nPoint);
        break;
    case tagScore::TY_TEXT:
        printf("%s\r\n", pScore->uscore.szText);
        break;
    }
}
int main(int argc, char *argv[])
{
    struct tagScore s1;
    s1.nType = tagScore::TY_CHAR;
    s1.uscore.cLevel = 'A';
```

```
    ShowScore(&s1);

    s1.nType = tagScore::TY_FLOAT;
    s1.uscore.fPoint = 88.5f;
    ShowScore(&s1);

    s1.nType = tagScore::TY_INT;
    s1.uscore.nPoint = 77;
    ShowScore(&s1);

    s1.nType = tagScore::TY_TEXT;
    strcpy(s1.uscore.szText, "Hahaha");
    ShowScore(&s1);

    system("pause");
    return 0;
}
```

变体原理：一个vt 控制类型，一个共用体控制解释方式，上层封装variant_t

tagVARIANT
ComVariant
variant_t
VT_I2