结构类型

2020年7月23日 13:13

常量符号化

```c
#include <stdio.h>

const int red = 0;
const int yellow = 1;
const int green = 2;

int main(int argc, char const *argv[])
{
    int color = -1;
    char *colorName = NULL;

    printf("输入你喜欢的颜色的代码: ");
    scanf("%d", &color);
    switch ( color ) {
    case red: colorName = "red"; break;
    case yellow: colorName = "yellow"; break;
    case green: colorName = "green"; break;
    default: colorName= "unknown"; break;
    }
    printf("你喜欢的颜色是%s\n", colorName);

    return 0;
}
```

- 用符号而不是具体的数字来表示程序中的数字

枚举

```c
#include <stdio.h>

enum COLOR {RED, YELLOW, GREEN};

int main(int argc, char const *argv[])
{
    int color = -1;
    char *colorName = NULL;

    printf("输入你喜欢的颜色的代码: ");
    scanf("%d", &color);
    switch ( color ) {
    case RED: colorName = "red"; break;
    case YELLOW: colorName = "yellow"; break;
    case GREEN: colorName = "green"; break;
    default: colorName= "unknown"; break;
    }
    printf("你喜欢的颜色是%s\n", colorName);

    return 0;
}
```

- 用枚举而不是定义独立的const int变量

枚举

- 枚举是一种用户定义的数据类型，它用关键字 enum 以如下语法来声明：

  enum 枚举类型名字 {名字0, ..., 名字n} ;

- 枚举类型名字通常并不真的使用，要用的是在大括号里的名字，因为它们就是就是常量符号，它们的类型是int，值则依次从0到n。如：

  enum colors { red, yellow, green } ;

- 就创建了三个常量，red的值是0，yellow是1，而green是2。
- 当需要一些可以排列起来的常量值时，定义枚举的意义就是给了这些常量值名字。

```c
#include <stdio.h>

enum color { red, yellow, green};

void f(enum color c);

int main(void)
{
    enum color t = red;

    scanf("%d", &t);
    f(t);

    return 0;
}

void f(enum color c)
{
    printf("%d\n", c);
}
```

- 枚举量可以作为值
- 枚举类型可以跟上enum作为类型
- 但是实际上是以整数来做内部计算和外部输入输出的

套路：自动计数的枚举

```c
#include <stdio.h>

enum COLOR {RED, YELLOW, GREEN, NumCOLORS};

int main(int argc, char const *argv[])
{
    int color = -1;
    char *ColorNames[NumCOLORS] = {
        "red", "yellow", "green",
    };
    char *colorName = NULL;

    printf("输入你喜欢的颜色的代码: ");
    scanf("%d", &color);
    if ( color >=0 && color < NumCOLORS ) {
        colorName = ColorNames[color];
    } else {
        colorName= "unknown";
    }
    printf("你喜欢的颜色是%s\n", colorName);

    return 0;
}
```

- 这样需要遍历所有的枚举量或者需要建立一个用枚举量做下标的数组的时候就很方便了

枚举量

- 声明枚举量的时候可以指定值
  - enum COLOR { RED=1, YELLOW, GREEN = 5};

```c
#include <stdio.h>

enum COLOR {RED=1, YELLOW, GREEN=5, NumCOLORS};

int main(int argc, char const *argv[])
{
    printf("code for GREEN is %d\n", GREEN);

    return 0;
}
```

枚举只是int

```c
#include <stdio.h>

enum COLOR {RED=1, YELLOW, GREEN=5, NumCOLORS};

int main(int argc, char const *argv[])
{
    enum COLOR color = 0;

    printf("code for GREEN is %d\n", GREEN);
    printf("and color is %d\n", color);

    return 0;
}
```

- 即使给枚举类型的变量赋不存在的整数值也没有任何warning或error

枚举

- 虽然枚举类型可以当作类型使用，但是实际上很(bu)少(hao)用
- 如果有意义上排比的名字，用枚举比const int方便
- 枚举比宏（macro）好，因为枚举有int类型

在函数内/外?

```c
#include <stdio.h>

struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today;

    today.month = 07;
    today.day = 31;
    today.year = 2014;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);

    return 0;
}
```

- 和本地变量一样，在函数内部声明的结构类型只能在函数内部使用
- 所以通常在函数外部声明结构类型，这样就可以被多个函数所使用了

声明结构类型

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    struct date {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 07;
    today.day = 31;
    today.year = 2014;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);

    return 0;
}
```

初学者最常见的
错误：漏了这个分号！

声明结构的形式

```c
struct point {
    int x;
    int y;
};

struct point p1, p2;
```

p1 和 p2 都是point
里面有x和y的值

```c
struct {
    int x;
    int y;
} p1, p2;
```

p1 和 p2都是一种无名结构，里面有x和y

```c
struct  point {
    int x;
    int y;
} p1, p2;
```

p1和p2都是point
里面有x和y的值t

对于第一和第三种形式，都声明了结构point。但是第二种形式没有声明point，只是定义了两个变量

结构变量

```c
struct date today;
today.month=06;
today.day=19;
today.year=2005;
```

month	11
day	23
year	2007

结构的初始化

```c
#include <stdio.h>

struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today = {07,31,2014};
    struct date thismonth = {.month=7, .year=2014};

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);
    printf("This month is %i-%i-%i.\n",
        thismonth.year,thismonth.month,thismonth.day);

    return 0;
}
```

没指明的，默认是0

结构成员

- 结构和数组有点像
- 数组用[]运算符和下标访问其成员
  - a[0] = 10;
- 结构用.运算符和名字访问其成员
  - today.day
  - student.firstName
  - p1.x
  - p1.y

结构运算

- 要访问整个结构，直接用结构变量的名字
- 对于整个结构，可以做赋值、取地址，也可以传递给函数参数
  - p1 = (struct point){5, 10}; // 相当于p1.x = 5; p1.y = 10;
  - p1 = p2; // 相当于p1.x = p2.x; p1.y = p2.y;

强制类型转换

数组无法做这两种运算！

```c
struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today;

    today = (struct date){07,31,2014};

    struct date day;

    day = today;
    day.year = 2015;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);
    printf("The day's date is %i-%i-%i.\n",
        day.year,day.month,day.day);
```

```
Today's date is 2014-7-31.
The day's date is 2015-7-31.
[Finished in 0.3s]
```

```c
struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today;

    today = (struct date){07,31,2014};

    struct date day;

    day = today;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);
    printf("The day's date is %i-%i-%i.\n",
        day.year,day.month,day.day);
```

```
Today's date is 2014-7-31.
The day's date is 2014-7-31.
[Finished in 0.3s]
```

复合字面量

- today = (struct date) {9,25,2004};
- today = (struct date) {.month=9, .day=25, year=2004};

结构指针

- 和数组不同，结构变量的名字并不是结构变量的地址，必须使用&运算符
- struct date *pDate = &today;

结构作为函数参数

```c
int numberOfDays(struct date d)
```

- 整个结构可以作为参数的值传入函数
- 这时候是在函数内新建一个结构变量，并复制调用者的结构的值
- 也可以返回一个结构
- 这与数组完全不同

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

struct date {
    int month;
    int day;
    int year;
};

bool isLeap(struct date d);
int numberOfDays(struct date d);

int main(int argc, char const *argv[])
{
    struct date today, tomorrow;

    printf("Enter today's date (mm dd yyyy):");
    scanf("%i %i %i", &today.month, &today.day, &today.year);

    if ( today.day != numberOfDays(today) ) {
        tomorrow.day = today.day+1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    } else if ( today.month == 12 ) {
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year+1;
    } else {
        tomorrow.day = 1;
        tomorrow.month = today.month+1;
        tomorrow.year = today.year;
    }

    printf("Tomorrow's date is %i-%i-%i.\n",
        tomorrow.year, tomorrow.month, tomorrow.day);

    return 0;
}

int numberOfDays(struct date d)
{
    int days;

    const int daysPerMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    if ( d.month ==2 && isLeap(d) )
        days = 29;
    else
        days = daysPerMonth[d.month-1];

    return days;
}

bool isLeap(struct date d)
{
    bool leap = false;

    if ( (d.year %4 ==0 && d.year %100 !=0 ) || d.year%400 ==0 )
        leap = true;

    return leap;
}
```

输入结构

- 没有直接的方式可以一次scanf一个结构
- 如果我们打算写一个函数来读入结构
  - —>
- 但是读入的结构如何送回来呢?
- 记住C在函数调用时是传值的
  - 所以函数中的p与main中的y是不同的
  - 在函数读入了p的数值之后，没有任何东西回到main，所以y还是 {0, 0}

```c
#include <stdio.h>

struct point {
    int x;
    int y; };

void getStruct(struct point);
void output(struct point);
void main( ) {
    struct point y = {0, 0};
    getStruct(y);
    output(y); }

void getStruct(struct point p) {
    scanf("%d", &p.x);
    scanf("%d", &p.y);
    printf("%d, %d", p.x, p.y); }

void output(struct point p) {
    printf("%d, %d", p.x, p.y); }
```

解决的方案

- 之前的方案，把一个结构传入了函数，然后在函数中操作，但是没有返回回去
  - 问题在于传入函数的是外面那个结构的克隆体，而不是指针
    - 传入结构和传入数组是不同的
- 在这个输入函数中，完全可以创建一个临时的结构变量，然后把这个结构返回给调用者

```c
void main( )
{
    struct point y = {0, 0};
    y = inputPoint( );
    output(y);
}
```

```c
struct point inputPoint(void)
{
    struct point temp;
    scanf("%d", &temp.x);
    scanf("%d", &temp.y);
    return temp;
}
```

也可以把y的地址传给函数，函数的参数类型是指向一个结构的指针。
不过那样的话，访问结构的成员的方式需要做出调整。

结构指针参数

```c
void main( )
{
    struct point y = {0, 0};
    inputPoint(&y);
    output(y);
}
```

```c
struct point* inputPoint(struct point *p)
{
    scanf("%d", &(p->x));
    scanf("%d", &(p->y));
    return p;
}
```

- 好处是传入传出只是一个指针的大小
- 如果需要保护传入的结构不被函数修改
  - const struct point *p
- 返回传入的指针是一种套路

指向结构的指针

```c
struct date {
    int month;
    int day;
    int year;
} myday;

struct date *p = &myday;

(*p).month = 12;
p->month = 12;
```

- 用->表示指针所指的结构变量中的成员

```c
struct point* getStruct(struct point*);
void output(struct point);
void print(const struct point *p);

int main(int argc, char const *argv[])
{
    struct point y = {0, 0};
    getStruct(&y);
    output(y);
    output(*getStruct(&y));
    print(getStruct(&y));
}

struct point* getStruct(struct point *p)
{
    scanf("%d", &p->x);
    scanf("%d", &p->y);
    printf("%d, %d", p->x, p->y);
    return p;
}

void output(struct point p)
{
    printf("%d, %d", p.x, p.y);
}

void print(const struct point *p)
{
    printf("%d, %d", p->x, p->y);
}
```

```c
*getStruct(&y) = (struct point){1,2};
```

这也可以！！

传进一个指针，在函数内部对指针处理之后再把指针返回

好处：将来可以串在其他函数的调用之中

结构数组

```c
struct date dates[100];
struct date dates[] = {
    {4,5,2005},{2,4,2005}};
```

```c
#include <stdio.h>

struct time {
    int hour;
    int minutes;
    int seconds;
};

struct time timeUpdate(struct time now);

int main(void)
{
    struct time testTimes[5] = {
        {11,59,59}, {12,0,0}, {1,29,59}, {23,59,59}, {19,12,27}
    };
    int i;

    for ( i=0; i<5; ++i ) {
        printf("Time is %.2i:%.2i:%.2i",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate(testTimes[i]);

        printf(" ...one second later it's %.2i:%.2i:%.2i\n",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);
    }

    return 0;
}
struct time timeUpdate(struct time now)
{
    ++now.seconds;
    if ( now.seconds == 60 ) {
        now.seconds = 0;
        ++now.minutes;

        if ( now.minutes == 60 ) {
            now.minutes = 0;
            ++now.hour;

            if ( now.hour ==24 ) {
                now.hour = 0;
            }
        }
    }
```

结构中的结构

```c
struct dateAndTime {
    struct date sdate;
    struct time stime;
};
```

嵌套的结构

```c
struct point {
    int x;
    int y;
};
struct rectangle {
    struct point pt1;
    struct point pt2;
};
```

如果有变量

struct rectangle r;

就可以有：

r.pt1.x、r.pt1.y,

r.pt2.x 和 r.pt2.y

如果有变量定义：

struct rectangle r, *rp;

rp = &r;

那么下面的四种形式是等价的：

r.pt1.x

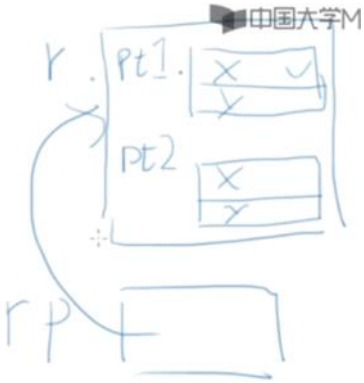
rp->pt1.x

(r.pt1).x

(rp->pt1).x

但是没有rp->pt1->x （因为pt1不是指针）

如果有变量定义：
```
struct rectangle r, *rp;
rp = &r;
```
那么下面的四种形式是等价的：
```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```
但是没有rp->pt1->x （因为pt1不是指针）



r.pt1.x
rp->pt1.x

结构中的结构的数组

```
#include <stdio.h>

struct point{
    int x;
    int y;
};

struct rectangle {
    struct point p1;
    struct point p2;
};

void printRect(struct rectangle r)
{
    printf("<%d, %d> to <%d, %d>\n", r.p1.x, r.p1.y, r.p2.x, r.p2.y);
}

int main(int argc, char const *argv[])
{
    int i;
    struct rectangle rects[ ] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}; // 2 rectangles
    for(i=0;i<2;i++) printRect(rects[i]);
}
```

```
#include <stdio.h>

struct point{
    int x;
    int y;
};

struct rectangle {
    struct point p1;
    struct point p2;
};

void printRect(struct rectangle r)
{
    printf("<%d, %d> to <%d, %d>\n", r.p1.x, r.p1.y, r.p2.x, r.p2.y);
}

int main(int argc, char const *argv[])
{
    int i;
    struct rectangle rects[ ] = {
        {{1, 2}, {3, 4}},
        {{5, 6}, {7, 8}}
    }; // 2 rectangles
    for(i=0;i<2;i++) {
        printRect(rects[i]);
    }
}
```

自定义数据类型 (typedef)

- C语言提供了一个叫做 ***typedef*** 的功能来声明一个已有的数据类型的新名字。比如：

```
typedef int Length;
```

使得 ***Length*** 成为 ***int*** 类型的别名。

- 这样， ***Length*** 这个名字就可以代替int出现在变量定义和参数声明的地方了：

```
Length a, b, len ;
Length numbers[10] ;
```

Typedef

声明新的类型的名字

- 新的名字是某种类型的别名
- 改善了程序的可读性

```
typedef long int64_t;
typedef struct ADate {
  int month;
  int day;
  int year;
} Date;

int64_t i = 100000000000;
Date d = {9, 1, 2005};
```

重载已有的类型名字
新名字的含义更清晰
具有可移植性

简化了复杂的名字

typedef

```
typedef struct {
    int month;
    int day;
    int year;
} Date;
```

typedef

```
typedef  int Length; // Length就等价于int类型

typedef *char[10] Strings;     // Strings 是10个字符串的数组的类型

typedef struct node {
   int data;
   struct node *next;
 } aNode;

或

typedef struct node aNode; // 这样用aNode 就可以代替 struct node
```

联合

选择:

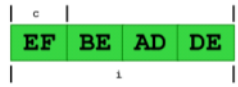
成员是

- 一个 int i 还是
- 一个 char c

sizeof(union …) =

sizeof(每个成员)的最大值

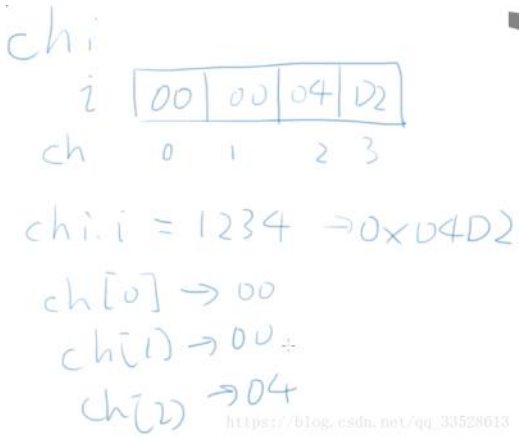
```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```



联合

联合起来使用同一个空间

- 存储
  - 所有的成员共享一个空间
  - 同一时间只有一个成员是有效的
  - union的大小是其最大的成员
- 初始化
  - 对第一个成员做初始化




```
3  typedef union {
4      int i;
5      char ch[sizeof(int)];
6  } CHI;
7
8  int main(int argc, char const *argv[])
9  {
10     CHI chi;
11     int i;
12     chi.i = 1234;
13     for ( i=0; i<sizeof(int); i++ ) {
14         printf("%02hhX", chi.ch[i]);
15     }
16     printf("\n");
17
18     return 0;
19 }
20
21
```

```
D2040000
[Finished in 2.1s]
```

得到一个整数/浮点数等等内部各个字节

文件操作！！

联合

union自己并不知道当时其中哪个成员是有效的

```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;


elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;


如果 (elt1 当前是char) …
```

?

程序怎么能知道当时elt1和elt2里面到底是int还是char?

?

最好的答案：另一个变量来表达这个事情

union的用处

```c
#include <stdio.h>

typedef union {
    int i;
    char ch[sizeof(int)];
} CHI;

int main(int argc, char const *argv[])
{
    CHI chi;
    int i;
    chi.i = 1234;
    for ( i=0; i<sizeof(int); i++ ) {
        printf("%02hhX", chi.ch[i]);
    }
    printf("\n");

    return 0;
}
```

这个结果表明我们所用的CPU是小端的