C 共用体 C typedef

C 位域

如果程序的结构中包含多个开关量，只有 TRUE/FALSE 变量，如下：

```
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1。在这种情况下，C 语言提供了一种更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成：

```
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

现在，上面的结构中，status 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 status 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。让我们看看下面的实例来理解这个概念：

实例

```
#include <stdio.h>
#include <string.h>
/* 定义简单的结构 */
struct
{
unsigned int widthValidated;
unsigned int heightValidated;
} status1;
/* 定义位域结构 */
struct
{
unsigned int widthValidated : 1;
unsigned int heightValidated : 1;
} status2;
int main( )
{
printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
```

```
return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

位域声明

在结构内声明位域的形式如下：

```
struct
{
  type [member_name] : width ;
};
```

下面是有关位域中变量元素的描述：

元素	描述
type	整数类型，决定了如何解释位域的值。类型可以是整型、有符号整型、无符号整型。
member_name	位域的名称。
width	位域中位的数量。宽度必须小于或等于指定类型的位宽度。

带有预定义宽度的变量被称为**位域**。位域可以存储多于 1 位的数，例如，需要一个变量来存储从 0 到 7 的值，您可以定义一个宽度为 3 位的位域，如下：

```
struct
{
  unsigned int age : 3;
} Age;
```

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成。让我们来看下面的实例：

实例

```
#include <stdio.h>
#include <string.h>
struct
{
unsigned int age : 3;
} Age;
int main( )
{
```

```
Age.age = 4;
printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
printf( "Age.age : %d\n", Age.age );
Age.age = 7;
printf( "Age.age : %d\n", Age.age );
Age.age = 8; // 二进制表示为 1000 有四位，超出
printf( "Age.age : %d\n", Age.age );
return 0;
}
```

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

← C 共用体　　C typedef →

2 篇笔记

写笔记

文中例子解析：

```
struct
{
    unsigned int age : 3;
} Age;

/*age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成*/
Age.age = 4;
printf("Sizeof( Age ) : %d\n", sizeof(Age));
printf("Age.age : %d\n", Age.age);

// 二进制表示为 111 有三位，达到最大值
Age.age = 7;
printf("Age.age : %d\n", Age.age);

// 二进制表示为 1000 有四位，超出
Age.age = 8;
printf("Age.age : %d\n", Age.age);
```

如果超出范围，则直接丢掉了，存不进去。

petter 8个月前 (07-26)

结构体内存分配原则

原则一：结构体中元素按照定义顺序存放到内存中，但并不是紧密排列。从结构体存储的首地址开始，每一个元素存入内存中时，它都会认为内存是以自己的宽度来划分空间的，因此元素存放的位置一定会在自己大小的整数倍上开始。

原则二： 在原则一的基础上，检查计算出的存储单元是否为所有元素中最宽的元素长度的整数倍。若是，则结束；否则，将其补齐为它的整数倍。

测试实例：

```
#include <stdio.h>

typedef struct t1{
    char x;
    int y;
    double z;
}T1;

typedef struct t2{
    char x;
    double z;
    int y;
}T2;

int main(int argc, char* argv[])
{
    printf("sizeof(T1) = %lu\n", sizeof(T1));
    printf("sizeof(T2) = %lu\n", sizeof(T2));

    return 0;
}
```

输出：

```
sizeof(T1) = 16
sizeof(T2) = 24
```

解析

```
sizeof(T1.x) = sizeof(T2.x) = 1;
sizeof(T1.y) = sizeof(T2.y) = 4;
sizeof(T1.z) = sizeof(T2.z) = 8;
```

T1: 若从第 0 个字节开始分配内存，则 T1.x 存入第 0 字节，T1.y 占 4 个字节，由于第一的 4 字节已有数据，所以 T1.y 存入第 4-7 个字节，T1.z 占 8 个字节，由于第一个 8 字节已有数据，所以 T1.z 存入 8-15 个字节。共占有 16 个字节。

T2: 若从第 0 个字节开始分配内存，则 T1.x 存入第 0 字节，T1.z 占 8 个字节，由于第一的 8 字节已有数据，所以 T1.z 存入第 8-15 个字节，T1.y 占 4 个字节，由于前四个 4 字节已有数据，所以 T1.z 存入 16-19 个字节。共占有 20 个字节。此时所占字节不是最宽元素（double 长度为 8）的整数倍，因此将其补齐到 8 的整数倍，最终结果为 24。

karma 7个月前 (08-09)