C语言概述

本文档仅对C语言部分概念做出一些简单解释，如果想要了解更多请上网搜索详细信息。

本资料已同步到github！[Vector5D/My_C_Repository: my first repository (github.com)]

计算机的内存分配

1. 代码区（.text）：存储着被装入执行的二进制代码，CPU到该区域取指并执行。
2. 数据区（.data）：存储全局变量，静态全局变量，静态局部变量，字符串常量等。其中，数据区细分为全局变量初始化区，全局变量未初始化区（置为零），字符串常量区。
3. 堆区（.heap）：进程可以在堆区动态的申请一定大小的内存，在使用完后归还给堆区。
4. 栈区（.stack）：函数被调用时分配栈区，用于存放函数的参数值，局部变量等。

C语言程序的执行流程

源文件.c/cpp	->	生成预编译文件.i	->	汇编文件.s	->	二进制目标文件.o/obj	->	可执行文件.exe
	预编译		编译		汇编		链接	

C语言数据类型

针对本台电脑来说，

```
int main() {
    printf("char size:          %d\n", sizeof(char));   //1
    printf("short size:         %d\n", sizeof(short));  //2
    printf("short int size:     %d\n", sizeof(short int));  //2
    printf("int size:           %d\n", sizeof(int));        //4
    printf("long size:          %d\n", sizeof(long));   //4
    printf("long long size:     %d\n", sizeof(long long));  //8

    printf("float size:         %d\n", sizeof(float));  //4
    printf("double size:            %d\n", sizeof(double)); //8
    printf("long double size:       %d\n", sizeof(long double));    //8
    printf("bool size:          %d\n", sizeof(bool));   //1
    return 0;
}
```

void类型不存在长度。

C语言运算符及优先级概述

1.乘等运算符

a *= 2 等价于 a = a * 2;

a *= b + c 等价于 a = a * (b + c)。因为* =优先级没有+优先级高。

2.求余运算符（取模运算符）

求余和取模的区别，对整型变量a,b来说，有下面的计算公式：

1. 求整数商：c = a / b;
2. 计算余数或者模：r = a - c * b;

求余运算在对c取值时，向0方向舍入；取模运算在对c取值时，向负无穷方向舍入。

> 总结：
> a和b符号相同时，求余和取模的结果相同；
> a和b符号不同时，求余结果的符号和a相同，取模结果的符号和b相同。
> 在不同环境下，%的含义不同：C/C++、JAVA为求余；python为取模。

3.自增、自减运算符

在++a、a++单独出现时，两者完全等价。
b=++a是a自增，再把自增后的结果赋值给b。
b=a++是将a的值赋值给b，然后a再自增。
自减运算符同理。

4.运算符优先级

一般来说有以下关系：！ > 算数运算符 > 关系运算符 > && > || > 赋值运算符
&&：与运算符，当第一个操作数为假，整个表达式为假，不进行第二个操作数的运算。
||：或运算符，当第一个操作数为真，整个表达式为真，不进行第二个操作数的运算。
所有运算符中，只有自增、自减运算符可以将改变作用于变量的本身。

C语言部分关键字概述

1.typedef关键字

该关键字将合法的变量声明语句作为某一数据类型的别名。比如：

```
typedef int state;
//state是int的别名，在变量的定义时，state a等价于int a。
typedef int Array[10];
//Array的类型是长度为10的int型数组，Array a等价于int a[10]。
```

typedef关键字还可以给结构体类型定义别名：

```
typedef struct student
{
    /*若干声明*/
}stu;
```

这种声明方式与typedef struct student stu等效，都是将stu作为student结构体数据类型的别名。

2.extern关键字

外部关键字，应用方向是：在一个工程项目下的两个文件，a文件想使用b文件当中定义的全局变量（或是函数），那么做法是，不改变b文件中的内容，在a文件中加入变量的外部声明，这样就可以在a文件当中使用b文件当中的全局变量和函数了。注意，局部变量不能在其他文件中被声明成外部变量，如果全局变量和函数是“静态”的，那么它们将不能在其他文件中被声明为外部变量。

3.static关键字

静态关键字，可以放在局部变量、全局变量、函数的声明之前，改变数据的某些属性。若static加在局部变量的声明之前，意味着该全局变量的生存期得到了延长，原本应该存储在栈区的局部变量获得了“静态”的属性，存储在了数据区，从而不会在函数调用结束之后被释放掉；若static加在全局变量的声明之前，改变的是全局变量的可见性，此时变量的可见性仅限于这个文件，在其他文件当中不能使用；函数和全局变量相似。

4.const关键字

常量关键字，用这个关键字声明的变量称为常变量，做右值，在定义时必须赋初值。

C语言的输入输出

1.三个输出函数（sprintf_s、fprintf、printf)

sprintf_s（将格式化数据写入缓冲区）
将数据格式化成字符串后，不是将其打印在屏幕上，而是存储在buffer指向的缓冲区中。缓冲区的大小应该足够大，以包含整个结果字符串。结束空字符将自动附加在内容之后。下面是例子：

```
int main()
{
    char buffer[50];
    int n, a = 5, b = 3;
    n = sprintf_s(buffer, 50, "%d plus %d is %d", a, b, a + b);
    printf("[%s] is a string %d chars long\n", buffer, n);
    return 0;
}
```

fprintf（将格式化数据写入数据流）
将字符串写入流当中，如果这其中有格式说明符(以%开头的子序列)，格式化并插入到结果字符串中，以替换它们各自的说明符，然后再将其写入到流当中。如果是标准输出流（stdout）的话，会将结果打印到屏幕上，这时与printf功能相同。下面是例子：

```
int main()
{
    int a = 5, b =3;
    fprintf(stdout, "a = %d b = %d", a, b);
    return 0;
}
```

printf（将格式化数据打印到标准输出）
printf函数可以看成是前两个函数的结合。首先使用sprintf_s函数格式化数据将其转换成字符串，存储进系统的缓冲区中，然后使用fprintf函数将格式化完成的数据送入标准输出流中，打印到屏幕上。

2.三个输入函数（sscanf_s、fscanf、scanf)

sscanf_s（从字符串中读取格式化的数据）
从字符串中读取数据并根据参数格式将其存储到附加参数指定的位置，就像使用scanf一样，但是从字符串中读取而不是标准输入。下面是例子：

```c
int main()
{
    char sentence[] = "Rudolph is 12 years old";
    char str[20];
    int i;

    sscanf(sentence, "%s %*s %d", str, &i);
    printf("%s -> %d\n", str, i);

    return 0;
}
```

fscanf（从流中读取格式化的数据）
从流中读取数据，并根据参数格式将数据存储到附加参数所指向的位置。如果是标准输入流（stdin）的话，会从键盘读入数据，这时与scanf功能相同。下面是例子：

```c
int main() {
    double fa;
    double fb;
    char ch;

    fscanf(stdin, "%lf%c%lf", &fa, &ch, &fb);
    printf("%lf", fa + fb);

    return 0;
}
```

scanf（从标准输入流读取格式化数据）
从标准输入流（stdin）中读取数据，并根据参数格式将其存储到附加参数所指向的位置。

安全性问题简述

1.以scanf函数为例

scanf()函数时C语言标准输入函数，在读取数据时不检查边界，所以可能会出现内存访问越界的情况（比如用户分配了5个字节的空间但是输入了10个字节，那么就会导致scanf()读到10个字节，这样的后果是，多出来的部分会被写到别的变量所在的内存空间上去，导致程序的异常）。

解决方法是，不使用scanf()函数，改用scanf_s()函数，使用这个函数时需要用户额外提供一个参数表示最多读取多少为字符，这样就可以解决内存访问越界的情况。目前VS2019已经禁止使用scanf()这个不安全的函数了。

2.空语句

一般的条件判断格式如下：

```c
if(//判断条件)
{
//执行语句
}
```

若出现

```
if(//判断条件) ;
{
//执行语句
}
```

分号会产生空语句问题，空语句就是什么都不做的语句，会破坏条件判断体或者循环体的结构。如上面的例子，正常情况下程序会进行条件判断，若是真则进入执行语句，但是在加了分号的情况下，判断条件是真，程序会执行空语句，反之不执行，不管哪种情况执行语句都会执行且只会被执行一次。

3.死循环

```
for( ; ; )
while(1)
do{...}while(1)
```

C语言在条件判断中，0、false、'\0'、NULL、nullptr表示“假”。除这些以外的情况均表示真。

部分概念解释

1.数组

数组是一种组合结构，我们在描述一个数组的时候，要将数组的类型和数组元素的数量说明才算一个完整的数组定义概念。
在数组定义中，有如下情况出现：int ar[] = {1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }。
要知道这个数组元素的个数，可以使用如下方法：数组的总长度/数组任一元素的长度=数组元素的个数。转换成C语言代码形式：元素个数 = sizeof(ar) / sizeof(ar[0])。

2.结构体

结构体属于用户自定义类型，不能被标准输入函数直接读取，但是可以分别访问其成员，用点运算符访问；如果是结构体指针，可以用箭头运算符访问。

3.文件

内存当中的程序与磁盘当中的文件交流信息（I/O），都会经过文件缓冲区，这个缓冲区用于暂存数据，程序与文件之间的数据通路称为数据流。

文件类型有两种，一种是文本文件，将数据转换成ASCLL码的形式进行存放；另一种是二进制文件，不需要转换数据格式，直接将内存中二进制形式的数据存入文件当中。

4.左值和右值

左值表示这个数据既可以被读取，又可以被赋值；右值表示这个数据只能被读取，不能被赋值。比如，常量、数字、表达式等不能被赋值，因为它们为右值。又如，用户定义的一个int型变量a，a既可以被读取，也可以被赋值，是一个左值。

5.作用域与生存期的区别

作用域是一个变量的有效范围，生存期是一个变量的存在时间（从声明到释放的时间）。

6.sizeof()和strlen()的区别

用这两个函数计算字符串str的长度，sizeof(str)的计算结果会将字符串末尾的'\0'也算进长度当中，而strlen(str)不会计算末尾的'\0'。即便是在字符串中间出现'\0'字符，sizeof(str)仍然会将其计算进总长度当中去，而strlen(str)遇到'\0'就会停止计数。

7.=和==的区别

在C语言中，=是赋值运算符，==是等于运算符，两者有本质上的区别。经常在条件判断条件中出现将两者混淆的错误。

```
    int i = 0;
    scanf_s("%d", &i);
    if (i = 0) { i = i + 1; }
    printf("%d", i);
```

程序原意是想让用户输入数值，若输入零让其加一并输出结果，但是事实上这个程序是错误的，因为判断条件中用的是赋值符号，语句的意思是将零赋值给i，也就是判断条件恒假，程序永远无法进入后续的执行语句。如果我们想用"相等"的逻辑，一定要使用==符号。

在需要从键盘读入一个字符时，建议不使用scanf标准输入函数，而使用专门的函数getchar()，这个函数的作用是从键盘中获取一个字符。还需注意，在编写函数的时候，除了一些专门用于输入输出的函数，凡事要通过scanf()输入的值，一律定义到形参当中，一般的函数中尽量不要出现标准输入输出函数。

8.分治策略

将一个规模比较大的问题，分解成许多规模较小的相同问题。问题不变，规模变小。经常和递归搭配使用。

9.递归

一个函数直接或者间接地调用自己，称这个函数为递归函数。具体分为"递推"和"回归"两个步骤，由终止条件控制，即逐层递推，然后递归终止条件满足，终止递推，逐层回归。分治和递归是两个孪生兄弟，经常同时出现在一个算法设计当中，产生许多高效的代码。

10.位运算

> 异或　^（相异为真，相同为假）
> 位与　&
> 位或　|
> 位反　~
> 左移　<<
> 右移　>>
> 区分位运算和逻辑运算
> 逻辑符号 && || ！
> 位运算　& | ~

位运算是将两个操作数转换为二进制数的形式之后，再对每一位进行与、或、非的逻辑判断，最终得到一个一串二进制数。逻辑符号最终的计算结果为1或者0，而位运算符计算出来的结果还是一个数。注意，**位运算的对象均是整型**，如char、int、short、long int、long long等。浮点数类型均不能进行位运算。

11.状态方案

和分治策略一样，是一种比较常用的编程思想。通过设置不同的状态将程序局部化，然后针对不同的状态来进行相应的逻辑判断，可以将一些复杂的逻辑判断简化。

数组和指针专题

1.数组名和指针的联系

当数组名作为函数参数传递时，数组名都会退化成一个指向数组首元素地址的指针。也就是说，声明函数时fun(int ar[10] , int n)和fun(int ar[] , int n)等这一类写法，编译器都会解释为fun(int* ar , int n)的形式。在这种情况下，ar等价于&ar[0]。由于数组名和指针之间的关系，当我们需要取得数组下标为i的元素时，ar[i]、*(ar+i)、i[ar]都可以成功访问数组对应下标的元素。

2.指针加一

指针加一相当于该指针指向元素的数据类型字节数乘以一。类似的，如果有以下指针变量type* p ; p = p + n , 相当于 p + sizeof(type) * n。

3.数据截断演示

```
int main() {
    int a = 0x12345678;
    int* ip = &a;
    short* sp = (short*)&a;
    char* cp = (char*)&a;
    printf("0x%08x\n", *ip);    //0x12345678
    printf("0x%08x\n", *sp);    //0x5678
    printf("0x%08x\n", *cp);    //0x78

    return 0;
}
```

变量a为十六进制数，我们将三个不同类型的指针都指向a的地址，然后输出这三个指针所指向元素的值。从结果可以看出，因为a本身是int型的变量，占用内存4个字节，所以对于int类型的指针ip来说，对ip解引用之后的值也是4个字节，所以将a完整地输出了出来；对于short类型的指针sp来说，short类型只占2个字节，所以对sp解引用后也占用2个字节，所以只会输出a的低2字节；对于char类型的指针cp来说，char类型只占1个字节，所以对cp解引用后也占用1个字节，所以只会输出a的低1字节。

4.探究指针

指针之间不能相加，但是可以相减。同类型的指针之间相减的结果是它们之间该类型元素的个数，这种性质一般被用于数组元素数量的计算当中。

```
int my_strlen(const char* str) {
    assert(str != nullptr);
    const char* sp = str;
    while (*sp != '\0') {
        sp++;
    }
    return sp - str;
}
```

该程序通过指针相减计算字符串的长度，基指针str始终指向字符串的开头，sp指针向后移动，寻找结束符即字符串的末尾，然后将两个指针相减，结果就是两个指针之间字符的个数，并通过返回值返回。

常量指针和指针常量。在介绍这两个概念之前，首先要明确，无论是常量，还是变量，它们的读取是不会受限制的，const修饰符的作用是用来改变变量的写入限制的。const int * p或者是int const * p是常量指针，其中const修饰的是* 运算符，可以理解为“封印”了指针的“指向”能力，也就是说该指针所指向元素的值不能被修改，但是可以重新给该指针另一个元素的地址，被读取的能力不受影响；int * const p是指针常量，现在指针变量本身是常量，所以不能改变该指针当中存储的地址，但是可以改变该指针所指向元素的值，被读取的能力不受影响。

补充：const int * const p是以上两种形式的结合，如果定义成这种形式的话，指针将不能被重新赋值成另一个变量的地址，也不能修改指针此时指向元素的值，该指针只能被读取。结合下面代码加深理解：

```
int main() {
    int a = 10, b = 20;
    const int* p1 = &a;
    *p1 = -100;     //error
    p1 = &b;        //ok

    int* const p2 = &a;
    *p2 = -100;     //ok
    p2 = &b;        //error

    const int* const p3 = &a;
    *p3 = -100;     //error
    p3 = &b;        //error

    return 0;
}
```

无类型void。无类型出现在函数声明的返回值处时void fun(/ * 一些定义 * /)，说明该函数不存在返回值；出现在函数声明处的形参处时/* 数据类型 */ fun(void)，说明该函数不接收任何参数；出现在强制类型转换中时(void)，后续的表达式（或变量）不能参与到之后其他任何运算当中，应用在断言机制当中，该函数的返回值被强制类型转换成了无类型，所以断言不能作为任何运算的操作数，是独立的一条语句。

无类型不能定义变量，但是无类型指针可以。无类型指针主要应用于C语言泛型编程当中，写法是void * point。这种指针的特点是可以存储任何数据类型的数据的地址，但是不能将无类型指针赋值给其他任何数据类型的指针。若想要解除这一限制，使用强制类型转换，将无类型指针转换成想要转换的类型再赋值。无类型指针还有一个缺陷，就是不能进行加法运算。原因是编译器无法判断无类型指针的数据类型，进而无法判断指针加一加的是多少字节。结合下面的代码可以加深理解：

```
void fun1(int a,int b)
{
    /*返回值是无类型的函数不能有返回值*/
}
int fun2(void)
{
    /*形参数无类型的函数不接收任何参数*/
}

int main() {
    const int n = 10;
    int a = 10;
    double b = 3.5;
    char c = 'a';
```

```
    int ar[n] = {};

    void* vp = &a;          //现在vp中存储变量a的地址
    int* ip = (int*)vp;     //使用强制类型转换将vp转换成int*类型，下同

    vp = &b;                //现在vp中存储变量b的地址
    double* dp = (double*)vp;

    vp = &c;                //现在vp中存储变量c的地址
    char* cp = (char*)vp;

    vp = ar;                //现在vp中存储数组ar的首地址

    const void* cvp = &n;
    //对于常量n需要在无类型指针前加上const标识符
    const int* cip = (const int*)cvp;
    //强制类型转换时也需要加上const标识符

    return 0;
}
```

结构体专题

1.结构体所占用字节数的计算

先确定实际对齐单位，其由以下三个因素决定：

> (1) CPU周期
>
> VS2019中默认8字节对齐
>
> (2) 结构体最大成员(基本数据类型变量)
>
> (3) 预编译指令#pragma pack(n)手动设置，其中n只能填1 2 4 8 16
>
> 上面三者取最小的,就是实际对齐单位(这里的"实际对齐单位"是为了方便区分随便取的概念)

除结构体的第一个成员外，其他所有的成员的地址相对于结构体地址(即它首个成员的地址)的偏移量必须为实际对齐单位或自身大小的整数倍(取两者中小的那个)。

结构体的整体大小必须为实际对齐单位的整数倍。

通过这三个步骤可以正确计算结构体所占字节数。计算下列情况下结构体所占字节数：

```
struct Student {
    char Stu_id[20];
    char Stu_name[20];
    char Stu_sex[6];
    int Stu_age;
};
```

首先确定实际对齐单位，没有手动设定对齐长度，选择int类型的长度为实际对齐长度为4。在为结构体第一个成员分配了空间之后，对于第二个、第三个char类型成员来说，偏移量是1，截止当前系统为结构体分配了20 + 20 + 6 = 46个字节数；对于第四个int类型成员来说，偏移量是4，所以系统会额外分配2个字节的空间，用以满足第四个成员的起始地址要求。所以，目前结构体已经占用了46 + 2 (额外分配) + 4 = 52字节的空间。判断这个大小是否是实际对齐长度的整数倍，52是4的整数倍，所以student结构体所占用的字节数是52。

```
struct sdate {
    int year;
    int month;
    int day;
};
struct Student {
    char s_id[10];
    char s_name[8];
    struct sdate birthday;//嵌套结构体
    double grade;
};
```

对于Student结构体来说，实际对齐单位为8，系统首先为s_id和s_namef分配18个字节的空间，然后进入内嵌结构体sdate中；sdate结构体的实际对齐单位为4，所以起始地址必须为4的整数倍，所以系统在18个字节的基础上又多分配了2个字节用以满足这个要求；系统为sdate结构体分配了12个字节的空间，截至目前一共分配了18 + 2 + 12 = 32个字节的空间，由于grade占8个字节，32又是8的整数倍，所以直接分配8个字节存放grade；一共32 +8 = 40个字节，又因为40是实际对单位8的整数倍，所以最终Student结构体所占用的字节数是40。

- **第三种情况**

```
struct Student {
    const char* name;
    int age;
};
struct Student s1;//sizeof(s1)?
struct Student s2 = { "yhping",20 };//sizeof(s2)?
```

观察Student结构体，是由一个char类型指针和一个int类型变量构成，对于s1，由于并没有初始化，所以占4 + 4 = 8字节；对于s2，"yhping"这个字符串并不在栈区分配长度，而是在数据区分配相应的长度，并由指针指向它；整型变量20存储在栈区当中。所以s2的长度仍然是一个指针和一个int变量的长度4 + 4 = 8字节。

2.结构体的深入理解

结构体是一种数据类型，可以将结构体比作“图纸”，将结构体变量比作“零件”，根据一张图纸生产出来的零件的特性都是一样的。由于结构体是创建变量的模板，不占用内存空间，所以我们编写结构体的过程叫做“设计结构体”而不是“定义结构体”。结构体变量包含了实实在在的数据，需要存储空间，所以编写结构体变量的过程叫做“定义结构体变量”。

结构体设计的形式不同，实现的原理也不同。如下面的程序段：

```
struct StudentA {
    char s_name[10];
    char s_id[10];
    char s_sex[8];
};
struct StudentB {
    const char* s_name;
    const char* s_id;
    const char* s_sex;
};
```

实际上，这两个结构体的功能几乎没有什么区别。在主函数中我们可以这样定义：

```
int main() {
    struct StudentA s1 = { "hdc","19001","man" };
    struct StudentB s2 = { "hdc","19001","man" };
    return 0;
}
```

但是，在定义结构体变量时，系统调用内存的情况是不一样的。由于结构体A在声明学生的姓名、学号、性别时采用数组的形式，所以对于结构体变量s1，系统会直接在栈区分配相应的内存空间；相对的，结构体B采用了指针的形式，所以对于结构体变量s1来说，系统只会在栈区（.stack）中开辟三个指针所需的空间，而在数据区（.data）分配相应的字符串所需的空间。

易错点提醒，比如有下面一段程序：

```
struct Student {
    char s_name[20];
    int s_age;
};

int main() {
    struct Student s1 = { "hdc",12 };
    char name[20];
    name = s1.s_name;           //error
    strcpy(name, s1.s_name);     //right
}
```

name = s1.s_name是错误的，因为name是一个数组名，在这里退化成存储数组的首元素地址的指针，是指针常量，不能被修改而指向别的数据。要想将s1.s_name中的字符串拷贝进name数组中，可以用系统提供的拷贝函数strcpy()，该函数包含在<string.h>头文件当中，接收两个地址，第一个地址作为目的操作数，第二个地址作为源操作数。

3.联合体（union）

同结构体一样，也是一种用户自定义类型，但是区别在于，结构体中每一个数据成员都拥有独立的存储空间，数据成员之间没有相互的影响，而且拥有一套字节对齐的方法；而对于联合体，联合体的长度是联合体当中最长数据成员的长度，联合体中的各个成员共用同一片内存，所以改变联合体中任意成员的值，其他联合体成员的值也会受到影响。

联合体举例：

```
union Node {
    unsigned char ch[2];
    unsigned short sx;
};
int main() {
    union Node x;
    x.sx = 0x6162;
    printf("%c \n", x.ch[0]);//b
    printf("%c \n", x.ch[1]);//a
}
```

联合体Node长度为2个字节，由ch数组和sx变量共用。首先初始化sx为十六进制的6162，由于是联合体，ch数组中也对应存储了数据，然后输出ch数组中的值。由于PC机采用小端存放，十六进制数62作为低位存储在ch[0]的位置，十六进制数61作为高位存储在ch[1]的位置，所以输出结果是62、61对应的字符'b'和'a'。

易错点提醒，请看下面的情况：

```
union Node {
    char a;
    char b;
    short sx;
};
int main() {
    union Node x;
    x.sx = 0x0001;
    if (x.b == 0) {
        printf("小端存放\n");
    }
    else {
        printf("大端存放\n");
    }
    return 0;
}
```

因为联合体长度为2个字节，由三个成员共用。由于对sx的初始化，联合体的第一个字节里存放的是十六进制数01。所以无论是成员a，还是成员b，所用的都是联合体开始的第一个字节，所以程序会输出"大端存放"。

4.哑元

简单来说，哑元就是一个无名的结构体。如下面的例子：

```
struct {
    char ch[2];
    int age;
}x;
struct {
    char ch[2];
    int age;
}y;
```

声明了两个哑元，并分别定义了两个无名结构体变量。虽然这两个哑元的成员变量一样，但并不是相同的数据类型，所以x和y不能相互赋值。

结构体和联合体的结合：

```
struct Node {
    char ch;
    union {
        int a;
        float f;
    };
};
int main() {
    struct Node s1;//8
    s1.ch = 'a';
    s1.a = 10;

    return 0;
}
```

结构体中有一个char类型的变量，还内嵌一个联合体的哑元，该哑元中含有两个成员，共用4个字节的空间，所以结构体变量s1占据8个字节的空间。虽然变量a和f被包含在一个联合体的哑元当中，但是它们仍然是属于结构体Node的数据成员，所以访问方式都是通过点运算符访问，只不过其中有两个成员共用了一片内存空间。

```
struct Node {
    char ch;
    union AF {
        int a;
        float f;
    };
};
int main() {
    struct Node s1;//1
    s1.ch = 'a';
    Node::AF af;//4
    af.a = 10;

    return 0;
}
```

结构体中有一个char类型的变量，接下来内嵌了一个联合体AF的声明，**不占用内存空间**，所以s1只占用1个字节的的空间。要想访问变量a和f，需要通过作用域解析运算符，定义一个联合体变量af，这个变量的长度为4个字节，然后再通过点运算符访问联合体中的数据成员。

```
struct Node {
    char ch;
    union {
        int a;
        float f;
    }AF;
};
int main() {
    struct Node s1;//8
    s1.ch = 'a';
    s1.AF.a = 10;

    return 0;
}
```

结构体中有一个char类型的变量，内嵌了一个联合体的哑元，并且定义了一个无名结构体变量AF。现在结构体中相当于有两个数据成员，一个是普通变量ch，另一个是联合体变量AF，所以s1占用8个字节。访问普通变量时直接采用点运算符，访问a和f时需要先通过点运算符访问AF，再通过AF访问联合体里的各个变量。

```
struct Node {
    char ch;
    typedef union {
        int a;
        float f;
    }AF;
};
int main() {
    struct Node s1;//1
    s1.ch = 'a';
```

```
    Node::AF af;//4
    af.a = 10;

    return 0;
}
```

结构体中有一个char类型的变量，内嵌了一个联合体的哑元，并且定义个一个无名结构体变量AF，但是前面加上了typedef关键字，该关键字的功能是将合法的变量声明更改成类型名。所以AF从一个无名结构体变量变成了一个类型名，类型的声明不需要占用空间，结构体变量s1占用1字节的空间。要想访问变量a和f，需要通过作用域解析运算符，定义一个联合体变量af，这个变量的长度为4个字节，然后再通过点运算符访问联合体中的数据成员。

数据类型专题

1.计算机中的数据按照补码形式存储，补码是原码按位取反加一得到。之所以采用补码是为了配合cpu的特性，因为cpu只能进行加法、按位取反、位移这三种逻辑。

2.C语言是一种强类型语言，变量的类型一旦确定就不能再被更改。

3.在C语言中，类型有自己的长度。不加任何前缀修饰时，默认是**signed**（有符号）类型，最高位代表符号位；当前缀关键字**unsigned**（无符号类型时），最高位就代表数值位。下面的程序展示了有符号和无符号之间的差别：

```
int main() {
    for (char i = 0;i < 128;i++) {
        printf("%4d", i);
    }
    return 0;
}
```

由于char类型的值始终介于-128~127之间，所以循环将无限执行。

```
int main() {
    for (unsigned char i = 0;i < 128;i++) {
        printf("%4d", i);
    }
    return 0;
}
```

由于将char定义成了无符号类型，所以它可以表示128,所以循环将会在循环128次后退出。

4.无符号和有符号的区别更多体现在类型转换方面。C语言的类型转换中，长度短的类型转换成长度长的类型时，需要进行位扩充，对于有符号数来说，扩充的应该是其符号位，之所以要扩充符号位，是因为要保证扩充后的数值正负和大小与原来保持不变；而对于无符号数来说，扩充的均为零。比如下面的程序段：

```c
int main() {
    char c = -5;
    unsigned int a = 10;
    if (c > a) {
        printf("%d > %d", c, a);
    }
    else {
        printf("%d < %d", c, a);
    }
    return 0;
}
```

程序结果输出-5 > 10，这显然是不符合逻辑的，但是事实上这是类型转换造成的结果。变量c的二进制表示为1111 1011（-5的补码形式），在进行c > a时进行了隐式类型转换，将c的类型转换成了unsigned int类型，所以要进行位扩充。由于c是char类型，扩充符号位，所以扩充后用十六进制表示0xfffffffd，而a用十六进制表示为0x0000000a，对于无类型来说最高位不是符号位，0xfffffffd远远大于0x0000000a，所以最后输出时打印出-5 > 10的情况。由于有隐式类型转换的存在，所以在编程的时候，不要将有符号数和无符号数混用，这样会造成一些严重的问题，比如下面的程序：

```c
int main() {
    for (int i = -4;i < sizeof(int);i++) {
        printf("%d ", i);
    }
    return 0;
}
```

程序将什么也不打印。因为sizeof(int)的返回值是一个无符号整型，所以i将会被进行隐式类型转换。i的十六进制表示为0xfffffffc，如果这是一个有符号数，那么它将是一个负数，但是此时i与一个无类型数进行比较，从int隐式转换成了unsigned int，所以i就变成了一个非常大的数，远远大于sizeof(int)的值，所以循环条件不满足，循环体未执行。

类型转换例题

```c
int main() {
    char c = 128;
    unsigned char uc = 128;
    unsigned short us = 0;

    us = c + uc;
    printf("%x \n", us);
    //  c = 1000 0000 -> 1111 1111 1000 0000
    //uc = 1000 0000 -> 0000 0000 1000 0000
    //                       us = 0000 0000 0000 0000

    us = (unsigned char)c + uc;
    printf("%x \n", us);
    //  c = 1000 0000 -> 0000 0000 1000 0000
    //uc = 1000 0000 -> 0000 0000 1000 0000
    //                       us = 0000 0001 0000 0000

    us = c + (char)uc;
    printf("%x \n", us);
    //  c = 1000 0000 -> 1111 1111 1000 0000
    //uc = 1000 0000 -> 1111 1111 1000 0000
    //                       us = 1111 1111 0000 0000
```

```
    us = (unsigned short)c + uc;
    printf("%x \n", us);
    //  c = 1000 0000 -> 1111 1111 1000 0000
    //uc = 1000 0000 -> 0000 0000 1000 0000
    //                       us = 0000 0000 0000 0000

    return 0;
}
```

需要注意，类型转换规则遵守的是数据原本类型的转换规则，而不是转换后类型的转换规则。

文件专题

文件的概念：一般指存储在外部介质上数据的集合，比如txt，tmp，jpg，exe等，C语言中将输入输出设备也抽象承一种“文件”。我们可以通过数据流向文件中写入或者读出数据。

流的概念：I/O设备型号众多而且标准不一，要想访问它们十分麻烦，所以我们将这些种类繁多的设备统一抽象成了“标准I/O设备”，程序绕过具体的设备，而与“标准I/O设备”进行交互，这样就可以做到不依赖与任何具体I/O设备的统一操作接口。我们将抽象出来的“标准I/O设备”或者“标准文件”称为“流”。对于将任意I/O设备转换为“标准I/O设备”的过程是由系统自动完成，不需要程序员处理。可以认为，任意输入的源端或者任意输出的终端均对应一个“流”。

文件包含三个要素：文件路径，文件名称，文件后缀。

预定义的三个标准流：stdin（标准输入流），stdout（标准输出流），stderr（标准错误流）。

文件类型FILE：对象类型，包含I/O流所需的全部信息。

文件操作的三个步骤：打开文件，读写文件，关闭文件。下面分别介绍这三个过程。

1.打开文件

要想对文件进行操作，首先要将文件打开，使用fopen函数达到这个目标。但是在VS2019中原来的fopen函数禁用，改用更安全的fopen_s函数。该函数接受三个参数，分别是文件指针、文件名、访问类型。如果打开失败，返回errno_t（int）值，用户可以通过返回的值来得知错误类型。访问类型常用的有以下几种：

读取方式	含义
r	只读。文件必须存在才可以读取。
w	只写。文件如果已经存在，则清除原文件重新写入；如果文件不存在， 则创建新文件写入。
a	末尾只写。文件如果已经存在，则在原文件的末尾继续写入；如果文件不存在，则创建新文件写入。
rb	二进制文件只读。功能同“r”，以二进制模式打开。
wb	二进制文件只写。功能同“w”，以二进制模式打开。

2.关闭文件

对文件操作完成之后应当关闭文件，使用fclose函数完成。该函数接受一个参数即文件指针，和动态内存分配中的free函数相似，在调用完fclose函数后，应当将文件指针置为空值，原理与调用free函数之后将指向堆区内存的指针置为空值相似。

3.读写文件

无格式输入输出

```
int main() {
    //putchar
    //puts
    const int n = 20;
    char ch = 'a';
    char stra[n] = { "hdc hello" };
    putchar(ch);// printf("%c", ch);
    putchar('\n');
    puts(stra);// prtintf("%s", stra);
    puts("\n");
    return 0;
}
```

putchar函数将char类型数据送进标准输出流中，puts函数将字符串送进标准输出流中。

```
int main() {
    //gets_s
    const int n = 100;
    char stra[n] = {};
    //scanf_s("%s", stra, n);//容纳n-1个有效字符，将空格视为结束符
    //printf("%s", stra);
    gets_s(stra, n);//容纳n-1个有效字符,将换行符视为结束符
    printf("%s", stra);
    return 0;
}
```

gets_s函数从标准输入流中获取字符串，在VS2019中不允许使用不安全的gets函数。

```
int main() {
    //getchar
    char ch = '\0';
    //scanf_s("%c", &ch);//stdin
    ch = getchar();//stdin

    return 0;
}
```

getchar函数从标准输入流中获取一个char类型数据，需要注意此函数存在返回值，需要使用变量接收。

```
int main(int argc, char* argv[]) {
    //fgetc
    //fputc
    char ch = '\0';
    if (argc < 3) {
        printf("copy file error \n");
        exit(EXIT_FAILURE);
```

```
    }
    FILE* fr, * fw;
    errno_t rx = fopen_s(&fr, argv[1], "r");
    errno_t wx = fopen_s(&fw, argv[2], "w");
    if (fr == nullptr || fw == nullptr) {
        printf("open file error \n");
        exit(EXIT_FAILURE);
    }
    while (!feof(fr)) {
        ch = fgetc(fr);
        fputc(ch, fw);
        putchar(ch);
    }
    fclose(fr);
    fr = nullptr;
    fclose(fw);
    fw = nullptr;
}
```

将一个文件中的内容拷贝到另外一个文件当中,以读方式打开一个文件，通过fgetc函数获取其字符，然后以写方式打开另一个文件，通过fputc函数将刚才获得的字符送进去，当读文件指针到达文件末尾的时候，拷贝结束。

```
int main() {
    //fgets
    //fputs
    FILE* fpr = nullptr;
    FILE* fpw = nullptr;
    errno_t ersa = fopen_s(&fpr, "Test7_10.cpp", "r");
    if (fpr == nullptr) {
        printf("fopen file error %d", ersa);
        exit(1);
    }
    errno_t ersb = fopen_s(&fpw, "D:\\VS2019程序代码
\\Test7_10\\Test7_12_copy.cpp", "w");
    if (fpw == nullptr) {
        printf("fopen file error %d", ersb);
        exit(1);
    }
    char buff[10];
    while (!feof(fpr)) {
        fgets(buff, 10, fpr);
        fputs(buff, fpw);
    }
    fclose(fpr);
    fpr = nullptr;
    fclose(fpw);
    fpw = nullptr;

}
```

使用fgets函数读取Test7_10.cpp中的内容进入缓冲区，然后使用fputs函数将缓冲区的数据写入另一个文件。当读文件指针到达文件末尾的时候，拷贝结束。

文本文件的写入和读取
下面的实例是通过SaveData函数创建并保存了一个txt文本文件，然后通过LoadData函数读取文件中的数值并打印至屏幕上。

```
void LoadData() {
    int ar[10];
    FILE* fp = nullptr;
    errno_t ers = fopen_s(&fp, "hdc.txt", "r");
    if (fp == nullptr) {
        printf("open file error %d", ers);
        exit(1);
    }
    for (int i = 0;i < 10;i++) {
        fscanf_s(fp, "%d", &ar[i]);//从文件流读取数据
        printf("%d ", ar[i]);
    }
    fclose(fp);
    fp = nullptr;
}
void SaveData() {
    int ar[] = { 12,23,34,45,56,67,78,89,90,100 };
    int n = sizeof(ar) / sizeof(ar[0]);
    FILE* fp = nullptr;
    errno_t ers = fopen_s(&fp, "hdc.txt", "w");
    if (fp == nullptr) {
        printf("open file error %d", ers);
        exit(1);
    }
    for (int i = 0;i < n;i++) {
        fprintf_s(fp, "%d ", ar[i]);//将数据写入到文件流
    }
    fclose(fp);
    fp = nullptr;
}
int main() {
    SaveData();
    LoadData();
    return 0;
}
```

第二个实例与第一个类似，区别是在文件的开头保存了文件中数据的个数，这样做的话在读取文件的时候，读取了文件中第一个值之后就可以知道文件的数据个数。

```
void LoadData() {
    int ar[100];
    FILE* fp = nullptr;
    errno_t ers = fopen_s(&fp, "data.txt", "r");
    if (fp == nullptr) {
        printf("fopen file error %d", ers);
        exit(1);
    }
    int n = 0;
    fscanf_s(fp, "%d", &n);
    for (int i = 0;i < n;i++) {
        fscanf_s(fp, "%d", &ar[i]);
        printf("%d ", ar[i]);
```

```
    }
    fclose(fp);
    fp = nullptr;
}
void SaveData() {
    int ar[] = { 12,23,34,45,56,67,78,89,90,100,90,89,78,67,56,45,34,23 };
    int n = sizeof(ar) / sizeof(ar[0]);
    FILE* fp = nullptr;
    errno_t ers = fopen_s(&fp, "data.txt", "w");
    if (fp == nullptr) {
        printf("fopen file error %d", ers);
        exit(1);
    }
    fprintf_s(fp, "%d\n", n);
    for (int i = 0;i < n;i++) {
        fprintf_s(fp, "%d ", ar[i]);
    }
    fclose(fp);
    fp = nullptr;
}

int main() {
    SaveData();
    LoadData();
    return 0;
}
```

二进制文件的写入和读取

相对于文本文件，二进制文件的读写分别采用fread和fwrite函数实现，其中fwrite接受四个参数，指向缓冲区的指针、写入数据的类型，写入数据的个数以及文件指针；fread函数接受四个参数，指向缓冲区的指针、读取数据的类型、读取数据的个数以及文件指针。

```
void LoadData() {
    FILE* fp = nullptr;
    errno_t ers = fopen_s(&fp, "data.bin", "rb");
    if (fp == nullptr) {
        printf("fopen file error %d", ers);
        exit(1);
    }
    int n = 0;
    fread(&n,sizeof(int), 1, fp);
    int* ar = (int*)malloc(sizeof(int) * n);
    if (ar == nullptr)exit(1);
    fread(ar,sizeof(int), n, fp);
    for (int i = 0;i < n;i++) {
        printf("%5d", ar[i]);
    }
    free(ar);
    ar = nullptr;
    fclose(fp);
    fp = nullptr;
}
void SaveData() {
    int ar[] = { 12,23,34,45,56,67,78,89,90,100,90,89,78,67,56,45,34,23 };
    int n = sizeof(ar) / sizeof(ar[0]);
    FILE* fp = nullptr;
```

```
    errno_t ers = fopen_s(&fp, "data.bin", "wb");
    if (fp == nullptr) {
        printf("fopen file error %d", ers);
        exit(1);
    }
    fwrite(&n, sizeof(int), 1, fp);
    fwrite(ar, sizeof(int), n, fp);
    fclose(fp);
    fp = nullptr;
}
int main() {
    SaveData();
    LoadData();
    return 0;
}
```

4.文件位置相关函数介绍

ftell：接受文件指针，返回当前文件位置指示值。
fgetpos：与ftell函数功能相似，功能是获取文件位置指示器。文件指针的位置是由专门的fpos_t类型变量给出。
下面的实例会在每次读取一个数据后打印文件指针的位置，可以看出，由于使用文本文件读写，读入的数据都被当作字符类型存储，这样的结果是随着数据的读取，文件位置的增加是没有规律的。

```
void LoadData() {
    int val;
    const int n = 10;
    FILE* fr = nullptr;
    errno_t rx = fopen_s(&fr, "hdc.txt", "r");
    if (fr == nullptr) {
        printf("open file error\n");
        exit(EXIT_FAILURE);
    }
    fpos_t ps;//int pos = ftell(fr);
    fgetpos(fr, &ps);
    while (!feof(fr)) {
        fscanf_s(fr, "%d", &val);
        printf("val = %d ", val);
        fgetpos(fr, &ps);//pos = ftell(fr);
        printf("ps = %lld \n", ps);//printf("pos = %d \n", pos);
    }
    fclose(fr);
    fr = nullptr;
}
void SaveData() {
    FILE* fw = nullptr;
    const int n = 10;
    int ar[n] = { 12,23,34,45,56,1,234,2345,6543,231 };
    errno_t wx = fopen_s(&fw, "hdc.txt", "w");
    if (fw == nullptr) {
        printf("open file error\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0;i < n;i++) {
        fprintf(fw, "%d ", ar[i]);
    }
```

```
    fclose(fw);
    fw = nullptr;
}
int main(){
    SaveData();
    LoadData();
    return 0;
}
```

第二个实例指出，如果以二进制文件写入，那么数据在内存中是以什么形式存储的，那么在文件中也是以什么形式存储。所以，随着读取数据的进行，文件位置的增加是均匀的。

```
void LoadData() {
    FILE* fr = nullptr;
    const int n = 10;
    int val;
    errno_t rx = fopen_s(&fr, "hdc.bin", "rb");
    if (fr == nullptr) {
        printf("open file error\n");
        exit(EXIT_FAILURE);
    }
    int pos = ftell(fr);
    for (int i = 0;i < n;i++) {
        fread(&val, sizeof(int), 1, fr);
        pos = ftell(fr);
        printf("val = %d pos = %d\n", val, pos);
    }
    fclose(fr);
    fr = nullptr;
}
void SaveData() {
    FILE* fw = nullptr;
    const int n = 10;
    int ar[n] = { 12,23,34,45,56,1,234,2345,6543,231 };
    errno_t wx = fopen_s(&fw, "hdc.bin", "wb");
    if (fw == nullptr) {
        printf("open file error\n");
        exit(EXIT_FAILURE);
    }
    fwrite(ar, sizeof(int), n, fw);
    fclose(fw);
    fw = nullptr;
}
int main(){
    SaveData();
    LoadData();
    return 0;
}
```

fseek：将文件位置指示符移动到指定位置。
fsetpos：将文件位置指示器移动到指定位置。
rewind：将文件位置指示器移动到文件首。
以下实例通过fseek函数将文件位置指示符移动到文件尾部，然后使用ftell可以得到文件的总字节数，向堆区动态申请对应空间，然后通过rewind函数将文件位置指示器移动到文件首，最后使用fread函数读取文件并打印值屏幕上。

```
int main() {
    FILE* fr = nullptr;
    errno_t err = fopen_s(&fr, "Test7_19.cpp", "rb");
    if (fr == nullptr) {
        printf("open file error\n");
        exit(EXIT_FAILURE);
    }
    //fseek(fr, n, SEEK_END);//设置到末尾，向前偏移n字节
    //fseek(fr, n, SEEK_SET);//设置到开头，向后偏移n字节
    //fseek(fr, n, SEEK_CUR);//设置到当前位置，向后偏移n字节
    fseek(fr, 0, SEEK_END);
    int len = ftell(fr);
    char* buff = (char*)malloc(sizeof(char) * len + 1);//预留一个结束符空间
    if (buff == nullptr)exit(1);
    rewind(fr);//fseek(fr, 0, SEEK_SET);
    fread(buff, sizeof(char), len, fr);
    buff[len] = '\0';
    printf("%s", buff);
    fclose(fr);
    fr = nullptr;
    return 0;
}
```

二级指针专题

什么是二级指针

简单来说，二级指针存储一级指针的地址，指向一级指针。

```
double a = 0, b = 0, c = 0, d = 0;
double* p0 = &a;
double* p1 = &b;
double* p2 = &c;
double* p3 = &d;
double** s = &p0;
```

如果a、b、c、d的地址是连续分配的，指针p0、p1、p2、p3的地址也是连续分配的，那么下面三种情况：

```
s + 1;      //&p1 移动4个字节 类型是double**
*s + 1;     //&b 移动8个字节 类型是double*
**s + 1;    //a + 1 = 1 类型是double
```

我们知道，指针加一的作用是加上其指向内容的数据类型的长度。
s开始指向的是一级指针p0，p0是指针，长度是4个字节，那么s+1就会移动4个字节，所以s指向了p1。
*s是s解引用的形式，也就是p0指针本身，而p0指针指向变量a，变量a的长度是8个字节，那么 *s+1就会移动8个字节，所以 *s（等同于p0）指向了b。
s 是 *s解引用的形式，也就是变量a，那么 s+1就是a变量本身加一，所以变量a由0变为1。

二级指针的应用：二维数组

数组指针和指针数组

```
int main() {
    int a = 0, b = 1, c = 3, d = 4;
    int* pr[4] = { &a,&b,&c,&d };   //指针数组
    int(*s)[4] = nullptr;           //数组指针
    int ar[4] = { 1,2,3,4 };
    s = &ar;

    return 0;
}
```

由程序可知，指针数组pr就是四个int类型变量构成的一个数组，性质和其他类型的数组一样。
数组指针通俗来讲是指向数组的指针，将数组看成一个整体，这个指针指向的是整个数组。也就是说，定义成*int(*s)[4]*，s就只能指向元素个数为4，类型为int的数组，不能指向其他类型的数组，或是元素数量不是4的数组。

数组地址和数组首元素地址

```
int main() {
    int ar[4] = { 1,2,3,4 };

    int* p0 = ar;
    int* p1 = &ar[0];

    int(*s)[4] = &ar;

    return 0;
}
```

我们知道，数组名就是数组首元素的地址，所以p0和p1是完全等价的，都存储数组首元素地址；通过指针数组定义方式的s指向的整个数组，存储数组的地址，虽然从结果上来看，数组首元素的地址和数组的地址相同，也就是说p0、p1、s中存储的地址都是相同的，但是区别在于它们加一的能力不同。p0、p1加一是相对于数组首元素地址的，加一是加数组中元素类型的长度；s是相对于数组地址的，加一是加数组的长度。
数组地址可以转化为数组首元素地址。s此时指向ar数组（s=&ar），那么*s*就指向数组首元素（s=ar）。

一维数组和二维数组

```
int main() {
    int ar[4] = { 1,2,3,4 };
    int size1 = sizeof(ar); //16字节
    int* p1 = ar;           //数组首元素的地址
    int(*s1)[4] = &ar;      //数组的地址

    int br[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12 };
    int size2 = sizeof(br); //48字节
    int(*p2)[4] = br;       //数组首元素的地址
    int(*s2)[3][4] = &br;   //数组的地址

    return 0;
}
```

如果说一维数组是一个一个变量组成的集合的话，那么二维数组就是一个一个一维数组组成的集合。br[3][4]可以看成是由三个一维数组构成，每个一维数组有四个变量。这样我们把二维数组每一个元素都视为一维数组，那么二维数组的首元素地址就是第一个一维数组的地址。我们如何定义一个指向一维数组的指针呢？就是利用指针数组！所以代码int(*p2)[4] = br是指向二维数组首元素的地址就容易理解了。如果要指向整个二维数组，采用int(*s2)[3][4] = &br的形式，这说明s2指向这样一个数组，这个数组是三行四列的二维数组，且数组的类型是int型。

数组下标形式和指针形式的转化

```
int main() {
    int ar[4] = { 1,2,3,4 };
    printf("%d\n", ar[2]);          //3
    printf("%d\n", *(ar + 2));      //3

    int br[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12 };
    printf("%d\n", br[2][3]);           //12
    printf("%d\n", *(*(br + 2)+3));     //12

    return 0;
}
```

br作为二维数组首元素的地址，br+2首先移动指针指向第三个元素，也就是第三个一维数组的首元素地址处，解引用(br+2)将其转换成指向这个一维数组首元素的指针，(br+2)+3移动指针指向一维数组中第四个元素，解引用 *(*(br + 2)+3)就是这个元素的值。

二维数组的实例

```
int main() {
    int ar[5][2] = { 1,2,3,4,5,6,7,8,9,10 };
    int(*p)[2] = &ar[1];
    int* s = ar[1];

    printf("%d \n", p[1][3]);   //8
    printf("%d \n", s[3]);      //6

    return 0;
}
```

结合前面的知识，该程序的输出结果容易得出。

动态内存管理专题

什么是动态内存管理

所谓动态内存管理，就是用户调用动态内存管理函数，在系统的堆区（.heap）申请一块内存，使用结束后将这块内存再归还给系统的过程。

栈区和堆区的区别

栈区：栈区在函数被调用时分配，用于存放函数的参数值，局部变量等值。在windows中栈的默认大小是1M，在VS2019中可以设置栈的大小。在Linux中栈的默认大小是10M，在使用gcc编译时可以设置栈区的大小。

堆区：程序运行时可以在堆区动态地申请一定大小的内存，并在用完之后归还给堆区。在Linux系统中堆区的大小接近3G。

禁用C99标准

在C99标准中，我们可以动态开辟一个数组。这种方式在栈区开辟空间，但是栈区本身可利用的空间就十分有限，如果想要开辟的数组很大，那么程序就会因为栈溢出而崩溃。与其在空间较小的栈区分配内存，不如直接在空间较大的堆区分配内存更加合适，所以在VS2019中直接规定禁止使用C99标准。

动态内存管理的优势

使用动态内存管理有两个好处。第一，使用传统数组开辟空间，往往会产生“大开小用”的情况。比如需要95个数据，但是定义数组时会定义大小是100个，这样造成了一些空间的浪费。动态内存申请就不会有这种现象，用户的需求是多少，它就会提供多少。第二，栈区的空间小，而在堆区的空间大，我们需要分配大量内存时，使用动态内存申请是最佳的选择。

动态内存管理函数

C语言中动态内存管理中有四个函数：**malloc**，**calloc**，**realloc**，**free**，都需要引用**stdlib.h**或**malloc.h**文件。

malloc函数

malloc向堆区申请一块指定大小的连续内存空间，函数格式如下：

```
void* malloc(size_t size);//typedef unsigned int size_t
```

分配size字节的未初始化内存。如果分配成功，分会指向新分配内存的指针；如果失败，返回空指针。由于并不是每一次申请malloc函数都会成功，所以malloc分配内存之后一定要判空。

free函数

free用来释放从malloc，realloc，calloc成功获取到的动态内存分配的空间。函数格式如下：

```
void free(void* ptr);
```

如果ptr是一个空指针，那么调用free函数将不执行任何操作。换句话说，无论在什么情况下，你都可以调动free函数释放一个空指针。

calloc函数

calloc与malloc功能相似，分配并使用零初始化连续内存空间。函数格式如下：

```
void* calloc(size_t num,size_t size);
```

为num个对象（元素）分配内存，并初始化所有分配存储中的字节为零。

realloc函数

realloc函数重新分配已有的内存块，可以将其扩大或者缩小。函数格式如下：

```
void* realloc(void* ptr,size_t new_size);
```

其中ptr指向原有内存，并重新分配new_size字节的空间大小。如果在扩大内存空间的时候出现错误（比如堆区空间不足），则返回空指针，分配失败。

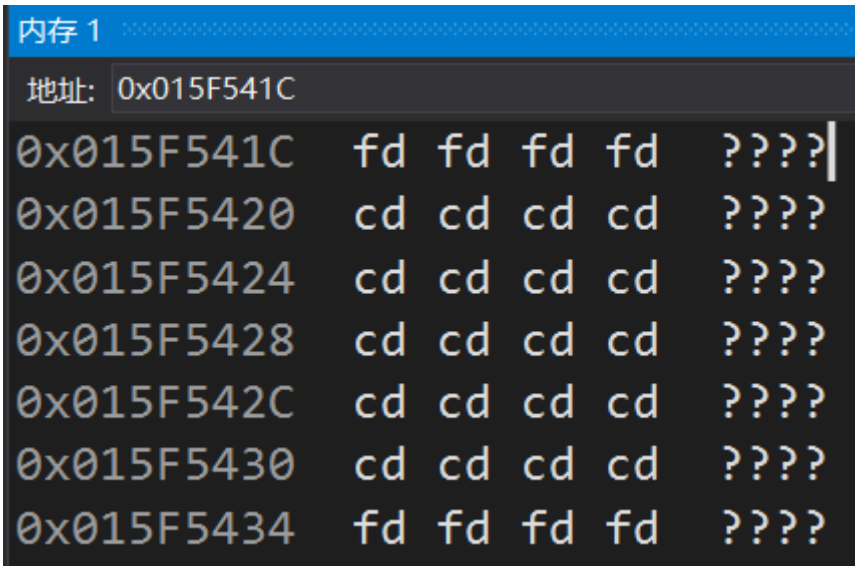
动态内存的使用

malloc调用实例

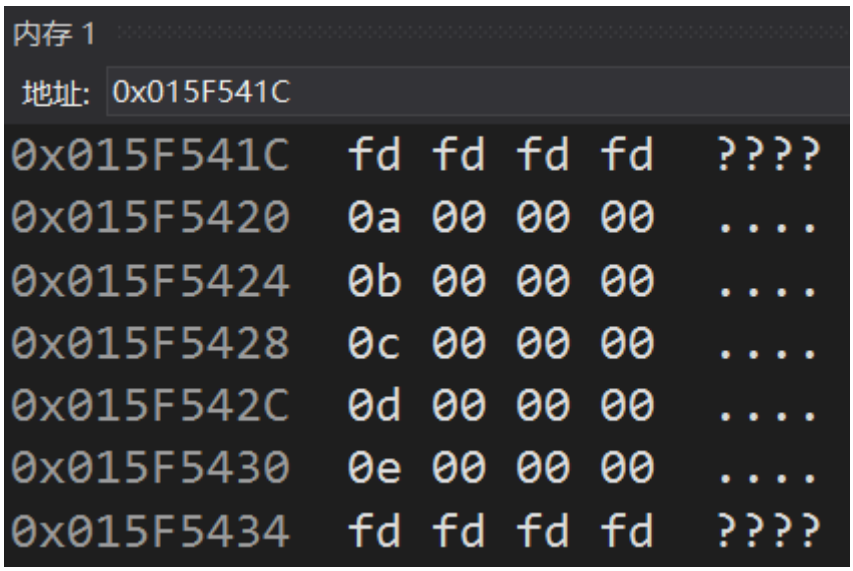
```
#include<stdio.h>
#include<stdlib.h>//malloc,free

int main() {
    int n = 0;
    int i = 0;
    scanf_s("%d", &n);//5
    int* p = (int*)malloc(sizeof(int) * n);
    if (p == nullptr)exit(EXIT_FAILURE);
    for (i = 0;i < n;i++) {
        p[i] = i + 10;
    }
    free(p);//why?
    p = nullptr;//why?
    return 0;
}
```

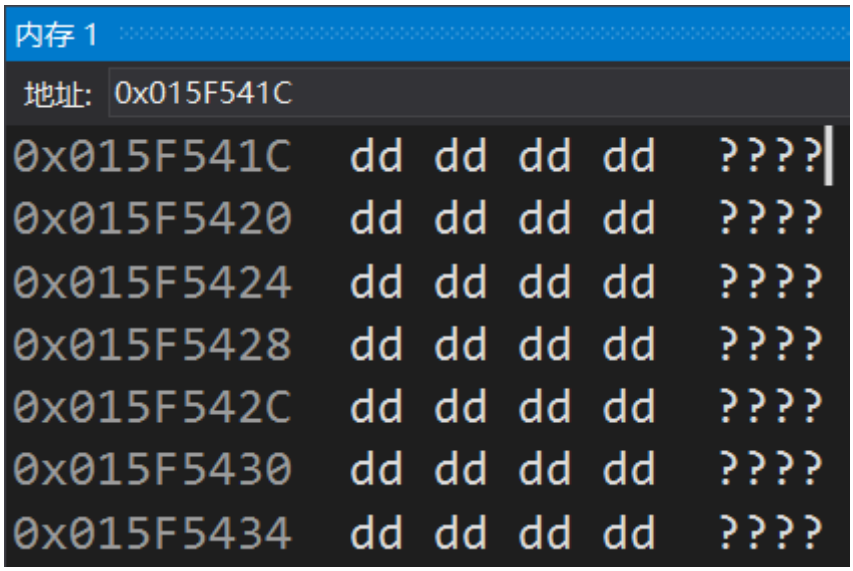
键入n的大小，malloc分配相应的字节数，分配成功，返回连续空间的首地址，由指针p接收。注意，连续空间的未初始化，并由十六进制数“cd”填充，图中“fd”填充在连续空间的上下两端，这分别代表上越界标记和下越标记，系统在调用free函数调用时，会查看越界标识是否被更改从而判断用户使用这块内存时是否产生了越界。



可以从图中看到，申请的五个整型空间已经被赋值。



调用free函数后，将堆区的内存归还给系统，使用“dd”将这片内存覆盖，以便下次动态内存申请时再次使用。



空悬指针

从刚才的调用实例中我们可以看到，在调用free函数之后，我们还要将p赋值成空指针，这样做是十分有必要的。free函数释放掉了指针p所指向堆区的内存，指针p仍然指向堆区的那片内存，但是此时内存已经被认为是释放掉了，这时指针p就被称为空悬指针。

空悬指针的危害

虽然不处理空悬指针，在主函数或者空悬指针所在的函数调用结束后，系统将会释放该函数的栈帧，指针将不复存在，但是在这之前，不处理空悬指针，将有可能产生严重的问题。

如果有一个空悬指针，仍然可以对其所指向的空间进行赋值，也可以将结果打印到屏幕上，但是不能对其再一次调用free函数，因为此时系统认为这片内存已经处于释放状态了，不能对已经释放过的内存在进行释放；还有一个最严重的问题，如果此时用户再一次向堆区申请分配内存，那么新分配的内存有可能和之前空悬指针指向的内存相同。一旦出现这种情况，改变空悬指针指向内存中的值，或者对空悬指针调用free函数，将会影响到真正指向这片内存的新指针，从而产生一系列各式各样的错误。所以，为了避免这种情况的发生，在调用free函数后，一定要将指针赋值成空指针。

内存的释放

free函数是释放堆区内存的，我们自然知道应该释放多少字节的内存，因为一开始malloc函数调用时就已经给出了要分配的字节数。但是系统是如何知道要释放多少空间的呢？其实在malloc函数调用时，并不是我们申请多少系统就给我们分配多少，通过刚才的调用实例也可以知道，除了真正存放数据的空间，还有越界标记存在。不止越界标记，在windows中还有28个字节左右的头部信息，用户分配内存的大小就存储在这些头部信息中。系统调用free函数时，会查看这些头部信息，从而就知道了用户申请内存的大小。除了头部信息，还有状态标记，用以标识这块内存的释放情况等等。所以可以知道，除开用户真正使用的内存，系统还将分配很多额外的内存来存储其他信息。

内存泄漏

内存泄漏有两种情况。一种是**堆区空间不足**造成的内存泄漏。顾名思义，就是一次性申请了太多空间，总而导致系统无法提供这么多的内存导致的内存泄漏。

还有一种是**通过malloc申请了一片内存，但是由于某些原因丢失了指向这片内存的指针**，这也会造成内存泄漏。因为在free函数调用时，要通过指针偏移一个固定值读取头部信息中用户开辟空间的大小，如果此时指针并没有在连续内存的起始处，那么偏移的位置就会出错，导致系统读取不到用户开辟内存的大小，从而导致释放时出现异常，程序崩溃。

动态内存申请构建二维数组

一般动态开辟二维数组的方法

对于一个m行n列的二维数组，首先动态开辟有m个元素的指针数组，返回值用一个二级指针接收。然后对每一个指针数组中的成员分别动态开辟n个元素的整型空间（假定类型是整形），完成了对二维数组的开辟。然后与二维数组的操作一样对其初始化。

```
int** Get2Array(int row, int col) {
    int** arr = (int**)malloc(sizeof(int*) * row);
    if (arr == nullptr)exit(1);
    for (int i = 0;i < row;i++) {
        arr[i] = (int*)malloc(sizeof(int) * col);
        if (arr[i] == nullptr)exit(1);
    }
    for (int r = 0;r < row;r++) {
        for (int c = 0;c < col;c++) {
            arr[r][c] = r + c;
        }
    }
    return arr;
}
```

接下来就可以像对二维数组一样控制内存，比如打印输出：

```
void Print_2Ar(int** arr, int row, int col) {
    for (int r = 0;r < row;r++) {
        for (int c = 0;c < col;c++) {
            printf("%-5d", arr[r][c]);
        }
        printf("\n");
    }
}
```

释放时采用逐级释放，首先将指针数组里的指针指向的堆内存释放掉，然后在释放二级指针。

```
void Free_2Ar(int** arr,int row) {
    for (int i = 0;i < row;i++) {
        free(arr[i]);
        arr[i] = nullptr;
    }
    free(arr);
    arr = nullptr;
}
```

测试用例如下：

```
int main() {
    int row, col;
    scanf_s("%d %d", &row, &col);
    int** arr = Get2Array(row, col);
    Print_2Ar(arr, row, col);
    Free_2Ar(arr, row);
    return 0;
}
```

利用结构体创建二维数组

基本思路是通过结构体创建一维数组，然后模拟二维数组。整体解决方案如下：

```
typedef int ElemType;
struct Array2 {
    ElemType* data;
    int row;
    int col;
};

void Init_2Ar(Array2& ar,int row,int col) {//初始化二维数组
    ar.row = row;
    ar.col = col;
    ar.data = (ElemType*)malloc(sizeof(ElemType) * row * col);
    if (ar.data == nullptr)exit(1);
    for (int i = 0;i < row * col;i++) {
        ar.data[i] = 0;
    }
}

void Print_2Ar(Array2& ar, int row, int col) {//输出二维数组
    for (int i = 0;i < row * col;i++) {
        if (i % col == 0)printf("\n");
        printf("%-5d", ar.data[i]);
    }
    printf("\n");
}

int GetItem(const Array2& ar, int r, int c) {//获取元素
    assert(r < ar.row && c < ar.col);
    return ar.data[r * ar.col + c];
}

void SetItem(Array2& ar, int r, int c, ElemType val) {//设置元素
    assert(r < ar.row && c < ar.col);
```

```
    ar.data[r * ar.col + c] = val;
}

void Destroy_2Ar(Array2& ar) {//销毁二维数组
    free(ar.data);
    ar.data = nullptr;
    ar.row = ar.col = 0;
}

/*
ElemType& Item(Array2& ar, int r, int c) {
    assert(r < ar.row && c < ar.col);
    return ar.data[r * ar.col + c];
}
*/
```

利用结构体创建n维数组

开辟n维数组的思路和二维数组相似，开辟一维数组，模拟n维数组的情况。不过在这个时候，需要调整结构体成员的构成：设置数组记录n维每一维的数据，记录下当前的维度。初始化函数和销毁函数如下所示：

```
typedef int ElemType;
struct Arrayn {
    ElemType* data;
    int* index;
    int n;
};

void Init_Arrayn(Arrayn& ar, int* info, int dimen) {//初始化n维数组
    assert(info != nullptr);
    ar.n = dimen;
    ar.index = (int*)malloc(sizeof(int) * dimen);
    if (ar.index == nullptr)exit(1);
    for (int i = 0;i < dimen;i++) {
        ar.index[i] = info[i];
    }
    int size = 1;
    for (int i = 0;i < dimen;i++) {
        size *= ar.index[i];
    }
    ar.data = (ElemType*)calloc(size, sizeof(ElemType));
    if (ar.data == nullptr)exit(1);
}

void Destroy_Arrayn(Arrayn& ar) {//销毁n维数组
    free(ar.index);
    ar.index = nullptr;
    free(ar.data);
    ar.data = nullptr;
    ar.n = 0;
}
```

调用时只需要传入结构体变量，每一维的数据和维度数即可，如下调用实例：

```
int main() {
    Arrayn ar;
    int info[3] = { 3,4,5 };
    Init_Arrayn(ar, info, 3);
    Destroy_Arrayn(ar);
    return 0;
}
```

柔性数组

什么是柔性数组

比如存在下面结构体：

```
#define MAXSIZE 1024
struct A_struct {
    int num;
    int size;
    char data[MAXSIZE];
};
```

由于结构体中数组的大小是定长的，所以数据的长度不足规定长度时会造成一定的浪费，这是无法避免会产生的问题。如果想要一个可变长度的数组，应该怎么办呢？我们自然可以采用动态内存的方式：

```
struct B_struct {
    int num;
    int size;
    char* data;
};
```

每当需要多少内存，我们就通过data指针向堆区申请对应的内存空间即可。这个指针只占4个字节的空间，我们几乎没有造成浪费。下面是一种使用方式：

```
struct B_struct* bp = (struct B_struct*)malloc(sizeof(struct B_struct));
if (bp == nullptr)exit(1);
bp->data = (char*)malloc(sizeof(char) * 100);
```

这样我们就可以存储100个字节的数据。还有下面的一种方式：

```
struct C_struct {
    int num;
    int size;
    char data[];//char data[0];
};
```

在C语言中结构体的**最后一个**元素可以是一个长度可变的数组，这种数组称为柔性数组。注意，数组名只是一个待使用的标识符，不占用实际的存储空间，所以以上结构体的大小只有8个字节。我们可以对这个数组的大小进行动态分配，如下所示：

```
struct C_struct* cp = (struct C_struct*)malloc(sizeof(struct C_struct) + 100);
if (cp == nullptr)exit(1);
```

这里指定了这个数组的长度就是100个字节。

柔性数组的特点

相较于定长数组，可变长度的数组节省空间；对于动态内存分配方式来说，采用柔性数组方式可以减少动态内存申请的次数，简化了操作。我们可以看到，采用柔性数组的时候，只需要使用结构体指针向堆区申请一次空间即可；而动态内存方式先要向堆区申请结构体的空间，然后使用结构体中的指针再一次申请空间才可以存放数据。

需要注意，柔性数组的使用仅限于结构体，且只能在结构体的最后声明。