# 面试学习笔记

## [预编译](https://so.csdn.net/so/search?q=%E9%A2%84%E7%BC%96%E8%AF%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_39849232/article/details/_blank)

预编译又称为预处理

程序的编译过程可以分为预处理、编译、汇编、链接

预处理，

根据源代码中的预编译指令修改源代码

处理以#开头的指令

#include包含的文件

#define宏定义

条件编译

[编译器](https://so.csdn.net/so/search?q=%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_39849232/article/details/_blank)进行的操作

可以放在程序中的任何位置

## [头文件](https://so.csdn.net/so/search?q=%E5%A4%B4%E6%96%87%E4%BB%B6&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Joker_N/article/details/_blank)

作用

通过头文件调用库功能

源码不方便向用户公布，只提供头文件和二进制的库

头文件加强类型安全检查

在接口被实现或使用时与头文件中的不一致，编译器就会报错，大大减轻了调试改错的负担。

重定义错误

不要在头文件中定义全局变量或函数。如果定义了全局变量，确保该头文件只被包含一次。如果定义了函数，将该函数声明为inline。

## inline 函数

替代 C 中复杂易错、不易维护的[宏函数](https://so.csdn.net/so/search?q=%E5%AE%8F%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/K346K346/article/details/_blank)

inline 通过消除调用开销来提升性能。

编译器在编译阶段完成对 inline 函数的处理

即对 inline 函数的调用替换为函数的本体

解决频繁调用的小函数大量消耗栈空间（栈内存）的问题

inline关键字是一种"用于实现的关键字",而不是"用于声明的关键字", 换句话说inline只有放在函数的定义前,才有可能使该函数成为内联函数,放在函数声明前无效.

在c++的类中定义的成员函数默认就是内联函数

但前提是在类中定义,因为我们正常的定义类的方式是在头文件中声明类的成员变量和成员函数,在源文件中对类的成员函数进行定义，此时成员函数的定义发生在类外，所以自然就不再是内联函数了,如果还要让其成为内联函数，就需要我们手动添加inline.

我为什么要说可能使该函数成为内联函数

inline是一种推荐型关键字,他并不保证该函数一定会成为内联函数,而是编译器在编译的时候获取有inline建议的函数对其规模,内容,方法等要素进行判断,如果满足编译器要求该函数会成为内联函数,反之不会成为内联函数,因此成不成内联函数完全由编译器说了算.

宏替换只是单纯的替换并不安全

内联函数在使用的时候,编译器会对内联函数调用的正确性进行判断如果不正确则该操作就不会发生，因此其更加安全

## 常量与指针

char \* const p

char const \* p

const char \*p

上述三个有什么区别？

char\* const p p为指针常量，地址不变，地址内容可变

const char \*p p为指针变量，地址可变，指向字符常量，地址内容不变

char const \*p 与const char \*p等价

const 修饰谁谁为常量

1.关键看const 修饰谁。

2.由于没有const \*的运算，若出现const\*的形式，则const实际上是修饰前面的。

## sizeof

sizeof是C语言和C++的关键字、运算符

sizeof不是函数

其作用是取得一个对象(数据类型或者数据对象)的长度(即占用内存的大小，以byte为单位)

sizeof不能求得void类型的长度；

不能用sizeof(void)，这将导致编译错误：illegalsizeof operand。事实上你根本就无法声明void类型的变量，不信你就试试void a;这样的语句，编译器同样会报错：illegal use of type 'void'。或许你要问为什么，很好，学东西不能只知其然，还要知其所以然。我们知道声明变量的一个重要作用就是告诉编译器该变量需要多少存储空间。然而，void是“空类型”，什么是空类型呢，你可以理解成不知道存储空间大小的类型。既然编译器无法确定void类型的变量的存储大小，那么它自然不让你声明这样的变量。当然了，声明void类型的指针是可以的！这就是特性2的内容。

sizeof能求得void类型的指针的长度；

可以申明void类型的指针，也就是说编译器可以确定void类型的指针所占用的存储空间。事实上确实如此，目前，几乎所有平台上的所有版本的编译器都把指针的大小看做4byte，不信你试试sizeof(int\*);sizeof(void\*);sizeof(double\*)；sizeof(Person\*)；等等，它们都等于4！为什么呢？问得好，我将尽全力对此作出解释：其实指针也是变量，只不过这个变量很特殊，它是存放其他变量的地址的变量。又由于目前32位计算机平台上的程序段的寻址范围都是4GB，寻址的最小单元是byte，4GB等于232Byte，这么多的内存其地址如果编码呢，只需要用32个bit就行了，而32bit = 32/8 = 4byte，也就是说只需要4byte就能存储这些内存的地址了。因此对任何类型的指针变量进行sizeof运算其结果就是4！

sizeof能求得静态分配内存的数组的长度

Int a[10];int n = sizeof(a);假设sizeof(int)等于4，则n= 10\*4=40；特别要注意：charch[]=”abc”;sizeof(ch);结果为4，注意字符串数组末尾有’\0’！通常我们可以利用sizeof来计算数组中包含的元素个数，其做法是：int n = sizeof(a)/sizeof(a[0]);

非常需要注意的是对函数的形参数组使用sizeof的情况。举例来说，假设有如下的函数：

void fun(int array[10])

{

int n = sizeof(array);

}

你会觉得在fun内，n的值为多少呢？如果你回答40的话，那么我很遗憾的告诉你，你又错了。这里n等于4，事实上，不管形参是int的型数组，还是float型数组，或者其他任何用户自定义类型的数组，也不管数组包含多少个元素，这里的n都是4！为什么呢？原因是在函数参数传递时，数组被转化成指针了，或许你要问为什么要转化成指针，原因可以在很多书上找到，我简单说一下：假如直接传递整个数组的话，那么必然涉及到数组元素的拷贝(实参到形参的拷贝)，当数组非常大时，这会导致函数执行效率极低！而只传递数组的地址(即指针)那么只需要拷贝4byte。

sizeof不能求得动态分配的内存的大小

假如有如下语句：int\*a = new int[10];int n = sizeof(a);那么n的值是多少呢？是40吗？答案是否定的！其实n等于4，因为a是指针，在特性2中讲过：在32位平台下，所有指针的大小都是4byte！切记，这里的a与特性3中的a并不一样！很多人(甚至一些老师)都认为数组名就是指针，其实不然，二者有很多区别的，要知详情，请看《c专家编程》。通过特性3和特性4，我们看到了数组和指针有着千丝万缕的关系，这些关系也是导致程序潜在错误的一大因素，关于指针与数组的关系问题我将在《C/C++刁钻问题各个击破之指针与数组的秘密》一文中进行详细介绍。

特性3指出sizeof能求静态分配的数组的大小，而特性4说明sizeof不能求的动态分配的内存的大小。于是有人认为sizeof是编译时进行求值的，并给出理由：语句int array[sizeof(int)\*10];能编译通过，而很多书上都说过数组大小是编译时就确定下来的，既然前面的语句能编译通过，所以认为sizeof是编译时进行求值的。经过进一步测试我发现这个结论有些武断！至少是有些不严谨！因为在实现了c99标准的编译器(如DEV C++)中可以定义动态数组，即：语句：int num;cin>>num; int arrary[num];是对的(注意在vc6.0中是错的)。因此我就在DEV C++中对刚才的array利用语句int n=sizeof(array);cout<<n<<endl来求大小，结果编译通过，运行时输入num的值10之后，输出n等于40！在这里很明显num的值是运行时才输入的，因此sizeof不可能在编译时就求得array的大小！这样一来sizeof又变成是运行时求值的了。

那么到底sizeof是编译时求值还是运行时求值呢？最开初c标准规定sizeof只能编译时求值，后来c99又补充规定sizeof可以运行时求值。但值得注意的是，即便是在实现了c99标准的DEV C++中仍然不能用sizeof求得动态分配的内存的大小！

sizeof不能对不完整的数组求长度

在阐述该特性之前，我们假设有两个源文件：file1.cpp和file2.cpp，其中file1.cpp中有如下的定义：

int arrayA[10] = {1,2,3,4,5,6,7,8,9,10};

int arrayB[10] = {11,12,13,14,15,16,17,18,19,20};

file2.cpp包含如下几个语句：

externarrayA[];

externarrayB[10];

cout<<sizeof(arrayA)<<endl;            //编译出错！！

cout<<sizeof(arrayB)<<endl;

在file2.cpp中第三条语句编译出错，而第条语句正确，并且能输出40！为什么呢？原因就是sizeof(arrayA)试图求不完整数组的大小。这里的不完整的数组是指数组大小没有确定的数组！sizeof运算符的功能就是求某种对象的大小，然而声明：extern int arrayA[]只是告诉编译器arrayA是一个整型数组，但是并没告诉编译器它包含多少个元素，因此对file2.cpp中的sizeof来说它无法求出arrayA的大小，所以编译器干脆不让你通过编译。

那为什么sizeof(arrayB)又可以得到arraryB的大小呢?关键就在于在file2.cpp中其声明时使用externint arrayB[10]明确地告诉编译器arrayB是一个包含10个元素的整型数组，因此大小是确定的。

到此本特性讲解差不多要结束了。其实本问题还能引申出连接和编译等知识点，但是目前我暂时还没自信对这两个知识点进行详细的，彻底的讲解，因此不便在此班门弄斧，不久的将来我会在本系列中加上相关问题的阐述。

当表达式作为sizeof的操作数时，它返回表达式的计算结果的类型大小，但是它不对表达式求值

为了说明这个问题，我们来看如下的程序语句：

char ch = 1;

int num=1;

int n1 =sizeof(ch+num);

int n2 = sizeof(ch =ch+num);

假设char占用1byte，int占用4byte，那么执行上面的程序之后，n1，n2，ch的值是多少呢？我相信有不少人会认为n1与n2相等，也有不少人认为ch等于2，事实这些人都错了。事实上n1等于4，n2等于1，ch等于1，为什么呢？请看分析：

由于默认类型转换的原因，表达式ch+num的计算结果的类型是int，因此n1的值为4！而表达式ch=ch+num；的结果的类型是char，记住虽然在计算ch+num时，结果为int，但是当把结果赋值给ch时又进行了类型转换，因此表达式的最终类型还是char，所以n2等于1。n1,n2的值分别为4和1，其原因正是因为sizeof返回的是表达式计算结果的类型大小，而不是表达式中占用最大内存的变量的类型大小！

对于n2=sizeof(ch=ch+num);乍一看该程序貌似实现了让ch加上num并赋值给ch的功能，事实并非如此！由于sizeof只关心类型大小，所以它自然不应该对表达式求值，否则有画蛇添足之嫌了。正是因为这点，这里告诫各位，尽量不要在sizeof中直接对表达式求大小，以免出现错误，你可以将sizeof(ch = ch+num)；改写成 ch = ch +num;sizeof(ch);虽然多了一条语句，看似冗余了，其实好处多多：首先更加清晰明了，其次不会出现ch等于1这样的错误(假设程序的逻辑本身就是要执行ch = ch +num;)。

sizeof可以对函数调用求大小，并且求得的大小等于返回类型的大小，但是不执行函数体！

假设有如下函数（是一个写得很不好的函数，但是能很好的说明需要阐述的问题）：

int fun(int& num,const int& inc)

{  
         float div = 2.0;

         double ret =0;

         num = num+inc;

         ret = num/div;

         return ret;

}那么语句：

int a = 3;

         int b = 5;

         cout<<sizeof(fun(a,b))<<endl;

         cout<<a<<endl;输出多少呢？不同的人会给出不同的答案，我将对sizeof(fun(a,b))的值和a的值分别进行讨论：

首先sizeof(fun(a,b))的值：其正确是4，因为用sizeof求函数调用的大小时，它得到的是函数返回类型的大小，而fun(a,b)的返回类型是int，sizeof(int)等于4。很多人把函数的返回类型和返回值的类型弄混淆了，认为sizeof(fun(a,b))的值是8,因为函数返回值是ret,而ret被定义成double，sizeof(doube)等于8。注意，虽然函数返回值类型是double，但是在函数返回时，将该值进行了类型转换(这里的转换不安全)。也有人错误的认为sizeof(fun(a,b))的值是12，它们的理由是：fun内部定义了两个局部变量，一个是float一个是double，而sizeof(float)+sizeof(doube)= 4+8=12。这样的答案看似很合理，其实他们是错误地认为这里的sizeof是在求函数内部的变量的大小了。这当然是错误的。

接下来看a的值：其正确答案是3！还记得特性6吗？这里很类似，sizeof的操作对象是函数调用时，它不执行函数体！为此，建议大家不要把函数体放在sizeof后面的括号里，这样容易让人误以为函数执行了，其实它根本没执行。

既然对函数条用使用sizeof得到的是函数返回类型的大小，那么很自然能得出这样的结论：不能对返回类型为void的函数使用sizeof求其大小！原因请参考特性1。同理，对返回类型是任何类型的指针的函数调用使用sizeof求得的大小都为4，原因请参考特性2。

最后我们来看看这样的语句：cout<<sizeof(fun)；其答案是多少呢？其实它得不到答案，原因是编译就通不过！最开始，我以为能输出答案4，因为我认为fun是函数名，而我知道函数名就是函数的地址，地址就是指针，于是我认为sizeof(fun)其实就是对一个指针求大小，根据特性2，任何指针的大小都是4。可是当我去验证时，编译器根本不让我通过！这个是为什么呢？我一时半会想不到，所以还请朋友们补充！

sizeof求得的结构体(及其对象)的大小并不等于各个数据成员对象的大小之和

结构体的大小跟结构体成员对齐有密切关系，而并非简单地等于各个成员的大小之和！比如对如下结构体两个结构体A、B使用sizeof的结果分别是：16,24。可以看出sizeof(B)并不等于sizeof(int)+sizeof(double)+sizeof(int)=16。

struct A{  
         int num1;

         int num2;

         double num3;

};

struct B{  
         int num1;

         double num3;

         int num2;

};

如果您不了解结构体的成员对齐，你会感到非常惊讶：结构体A和B中包含的成员都一样，只不过顺序不同而已，为什么其大小不一样呢？要解释这个问题，就要了解结构体成员对齐的规则，由于结构体成员对齐非常复杂，我将用专题——C/C++刁钻问题各个击破之位域和成员对齐——进行讲解，这里我只简单地介绍其规则：

1、  结构体的大小等于结构体内最大成员大小的整数倍

2、  结构体内的成员的首地址相对于结构体首地址的偏移量是其类型大小的整数倍，比如说double型成员相对于结构体的首地址的地址偏移量应该是8的倍数。

3、  为了满足规则1和2编译器会在结构体成员之后进行字节填充！

基于上面三个规则我们来看看为什么sizeof(B)等于24：首先假设结构体的首地址为0，第一个成员num1的首地址是0(满足规则2，前面无须字节填充，事实上结构体绝对不会在第一个数据成员前面进行字节填充)，它的类型是int，因此它占用地址空间0——3。第二个成员num3是double类型，它占用8个字节，由于之前的num1只占用了4个字节，为了满足规则2，需要使用规则3在num1后面填充4个字节（4——7），使得num3的起始地址偏移量为8，因此num3占用的地址空间是：8——15。第三个成员num2是int型，其大小为4，由于num1和num3一共占用了16个字节，此时无须任何填充就能满足规则2。因此num2占用的地址空间是16——19。那么是不是结构体的总大小就是0——19共20个字节呢？请注意，别忘了规则1！由于结构体内最大成员是double占用8个字节，因此最后还需要在num2后面填充4个字节，使得结构体总体大小为24。

按照上面的三个规则和分析过程，你可以很容易地知道为什么sizeof(A)等于16。特别需要说明的是，我这里给出了三个结论性的规则，而没有阐述为什么要这样。你或许有很多疑问：为什么要结构体成员对齐，为什么要定义规则1等。如果你有这样的疑问，并尝试去弄清楚的话，那么我敢断言，不久的将来你必定会有大成就，至少在学习c++上是这样。前面说过，我会再写一篇专题：C/C++刁钻问题各个击破之位域和成员对齐来详细回答这些问题，如果你急于要弄明白，那么你可以参考其他资料，比如说《高质量c++程序设计指南》。

最后再提醒一点，在进行设计时，最好仔细安排结构体中各个成员的顺序，因为你已经看到了上面的结构体B与结构体A包含的成员相同，只不过顺序略有差异，最终就导致了B比A多消耗了50%的空间，假如在工程中需要定义该结构体的数组，多消耗的空降将是巨大的。即使将来内存降价为白菜价格，你也不要忽视这个问题，勤俭节约是中国人民的优良传统，我们应该继承和保持！

sizeof不能用于求结构体的位域成员的大小，但是可以求得包含位域成员的结构体的大小

首先解释一下什么是位域：类型的大小都是以字节(byte)为基本单位的，比如sizeof(char)为1byte，sizeof(int)为4byte等。我们知道某个类型的大小确定了该类型所能定义的变量的范围，比如sizeof(char)为1byte，而1byte等于8bit，所以char类型的变量范围是-128——127，或者0——255(unsigned char)，总之它只能定义28=256个数！然而，要命的是bool类型只取值true和false，按理所只用1bit(即1/8byte)就够了，但事实上sizeof(bool)等于1。因此我们可以认为bool变量浪费了87.5%的存储空间！这在某些存储空间有限的设备(比如嵌入式设备)上是不合适的，为此需要提供一种能对变量的存储空间精打细算的机制，这就是位域。简单来说，在结构体的成员变量后面跟上的一个冒号+一个整数，就代表位域，请看如下的结构体：

Struct A

{

         Bool b:1;

         char ch1:4;

         char ch2:4;

}item; 其中b,ch1,ch2都是位域成员，而i是普通成员。该结构体的试图让bool类型的变量b只占用1个bit，让ch1和ch2分别只占用4个bit，以此来达到对内存精打细算的功能(事实上使用位域对内存精打细算有时候能成功，有时候却未必，我将《C/C++刁钻问题各个击破之位域和成员对齐》进行论述)。另外需要特别注意的是：c语言规定位域只能用于int，signed int或者unsigned int类型，C++又补充了char和long类型！你不能这样使用位域：float f:8;这是不能通过编译的。并且位域变量不能在函数或者全局区定义，只能在结构体，自定义类，联合(union)中使用！

基于上面的结构体，语句sizeof(item.b)和sizeof(item.ch1)等对位域成员求大小的语句均不能通过编译。其原因能再本篇的概论中找到：sizeof以byte为单位返回操作数的大小！

那么爱学好问的你可能要问，sizeof(A)能否通过编译呢？如何能，其结果又是多少呢？这是两给非常好的问题，事实上我之前没有看到任何关于这方面的论述(可能是我看的资料不足)，我正是在看到sizeof(item.b)不能通过编译时想到了这两个问题，然后通过验证得出了后面的结论：对包含位域的结构体是可以使用sizeof求其大小的，但其求值规则比较复杂，不仅涉及到成员对齐，还与具体编译环境有关！在这里你只需要知道可以对包含位域的结构体使用sizeof求其大小，对于sizeof是根据什么规则来求这个大小的问题，我将会在专题：《C/C++刁钻问题各个击破之位域和成员对齐》中进行详细阐述。

以下代码中的两个sizeof用法有问题吗？

void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母

{

for( size\_t i=0; i <sizeof(str)/sizeof(str[0]); ++i )

if( 'a' <=str[i] && str[i] <='z' )

str[i] -= ('a'-'A' );

}

char str[] = "aBcDe";

cout < < "str字符长度为: " < < sizeof(str)/sizeof(str[0]) < < endl;

UpperCase( str );

cout < < str < < endl;

答案：函数内的sizeof有问题。根据语法，sizeof如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的str是一个静态定义的数组，因此其大小为6，因为还有'\0'，函数内的str实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此sizeof作用于上只将其当指针看，一个指针为4个字节，因此返回4。

sizeof（函数名）

sizeof是计算数据类型所占的字节数的运算符，sizeof的结果等于对象或者类型所占的内存字节数。sizeof（）中，（）后为“空结构体”（不含数据成员）的大小不为0，而是1。试想一个“不占空间”的变量如何被取地址、两个不同的“空结构体”变量又如何得以区分呢于是，“空结构体”变量也得被存储，这样编译器也就只能为其分配一个字节的空间用于占位了。

## 动态内存分配

随时申请随时释放

一个很长的链表需要不断地malloc和free内存，它的意义并不单纯在于创建一个很大的数组，也在于它可以极大地提高内存的利用率。即在复制新链表的同时，可以手动将之前的旧链表删除，把内存释放出来去存储其他东西，但单纯的本地数组变量却做不到，它只能在一个程序结束时一股脑地全部释放掉，而在此之前它是一直存在着的。可以将想象一下，当你在一个程序中像链表一样不断地创建本地数组，而本地数组却只能在结束时自动释放，本地数组不断堆积，最终会让程序因为内存空间不足而无法执行。

## 一个32位的机器,该机器的指针是多少位

答案：

指针是多少位只要看地址总线的位数就行了。80386以后的机子都是32的数据总线。所以指针的位数就是4个字节了。

## 6程序题

main()

{

int a[5]={1,2,3,4,5};

int \*ptr=(int \*)(&a+1);

printf("%d,%d",\*(a+1),\*(ptr-1));

}

答案：2。

5 \*(a+1）就是a[1]，\*(ptr-1)就是a[4],执行结果是2，5 &a+1不是首地址+1，系统会认为加一个a数组的偏移，是偏移了一个数组的大小（本例是5个int） int \*ptr=(int \*)(&a+1); 则ptr实际是&(a[5]),也就是a+5 原因如下： &a是数组指针，其类型为 int (\*)[5]; 而指针加1要根据指针类型加上一定的值，不同类型的指针+1之后增加的大小不同 a是长度为5的int数组指针，所以要加 5\*sizeof(int) 所以ptr实际是a[5] 但是prt与(&a+1)类型是不一样的(这点很重要) 所以prt-1只会减去sizeof(int\*) a,&a的地址是一样的，但意思不一样，a是数组首地址，也就是a[0]的地址，&a是对象（数组）首地址，a+1是数组下一元素的地址，即a[1],&a+1是下一个对象的地址，即a[5].