

第 2 部分

C/C++程序设计

C / C + + p r o g r a m d e s i g n

本部分主要以 C/C++设计语言为基础，通过大量实际的例子分析各大公司 C/C++面试题目，从技术上分析问题的内涵。为什么要选择 C 系的语言呢？这是因为各大公司的编程语言绝大多数是 C 系语言，虽然 Java 也占很大的比重，可是 C++相对于 Java 来说更有区分度——C++是那种为每一个问题提供若干个答案的语言，远比 Java 灵活，所以面试考题绝大多数以 C/C++为主（或者是两套试题，C++或 Java，面试者可以选择）。

许多面试题看似简单，却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 strcpy 函数就可看出面试者在技术上究竟达到了怎样的水平。我们能真正写好一个 strcpy 函数吗？我们都觉得自己能，可是我们写出的 strcpy 很可能只能拿到 10 分中的 2 分。读者可以从本部分中关于 C++的几个常用考点，看看自己属于什么样的层次。

第 5 章

程序设计基本概念

作为一个求职者或是应届毕业生，公司除了对你的项目经验有所问询之外，最好的考量办法就是你的基本功，包括你的编程风格，你对赋值语句、递增语句、类型转换、数据交换等程序设计基本概念的理解。当然，在考试之前你最好对你所掌握的程序概念知识有所复习，尤其是各种细致的考点要加以重视。以下的考题来自真实的笔试资料，希望读者先不要看答案，自己解答后再与答案加以比对，找出自己的不足。

5.1 赋值语句

面试例题 1：Which of the following statements describe the results of executing the code snippet below in C++?（下列 C++代码的输出结果是什么？）[台湾某著名杀毒软件公司 2010 年 7 月笔试题]

```
C/C++ code
int i = 1;
void main()
{
    int i = i;
}
```

- A. The i within main will have an undefined value. （main()里的 i 是一个未定义值）
- B. The i within main will have a value of 1. （main()里的 i 值为 1）
- C. The compiler will not allow this statement. （编译器不允许这种写法）
- D. The i within main will have a value of 0. （main()里的 i 值为 0）

解析：当面试者看到 `int i=i;` 时，也许第一反应就是怎么有这么诡异的代码？但是在 C++ 中这样做是完全合法的（但显然不合理）。`int i = i`，`i` 变量从声明的那一刻开始就是可见的了，

main()里的 i 不是 1，因为它和 main()外的 i 无关，而是一个未定义值。

答案：A

面试例题 2：What does the following program print? (下面程序的结果是多少?) [中国台湾某著名计算机硬件公司 2005 年 12 月面试题]

```
#include <iostream>
using namespace std;
int main()
{
    int x=2,y,z;
    x *=(y=z=5); cout << x << endl;
    z=3;
}
```

```
x ==(y=z); cout << x << endl;
x =(y==z); cout << x << endl;
x =(y&z); cout << x << endl;
x =(y&&z); cout << x << endl;
y=4;
x=(y|z); cout << x << endl;
x=(y||z); cout << x << endl;
return 0;
}
```

解析：

$x *= (y=z=5)$ 的意思是说 5 赋值给 z， z 再赋值给 y， $x=x*y$ ，所以 x 为 $2*5=10$ 。

$x ==(y=z)$ 的意思是说 z 赋值给 y，然后看 x 和 y 相等否？不管相等不相等，x 并未发生变化，仍然是 10。

$x =(y==z)$ 的意思是说首先看 y 和 z 相等否，相等则返回一个布尔值 1，不等则返回一个布尔值 0。现在 y 和 z 是相等的，都是 3，所以返回的布尔值是 1，再把 1 赋值给 x，所以 x 是 1。

$x =(y&z)$ 的意思是说首先使 y 和 z 按位与。y 是 3，z 也是 3。y 的二进制数位是 0011，z 的二进制数位也是 0011。按位与的结果如下表所示。

y	0	0	1	1
z	0	0	1	1
y&z	0	0	1	1

所以 y&z 的二进制数位仍然是 0011，也就是还是 3。再赋值给 x，所以 x 为 3。

$x =(y&&z)$ 的意思是说首先使 y 和 z 进行与运算。与运算是指如果 y 为真，z 为真，则 $(y&&z)$ 为真，返回一个布尔值 1。这时 y、z 都是 3，所以为真，返回 1，所以 x 为 1。

$x =(y|z)$ 的意思是说首先使 y 和 z 按位或。y 是 4，z 是 3。y 的二进制数位是 0100，z 的二进制数位是 0011。与的结果如下表所示。

y	0	1	0	0
z	0	0	1	1
y&z	0	1	1	1

所以 y&z 的二进制数位是 0111，也就是 7。再赋值给 x，所以 x 为 7。

$x = (y \mid\mid z)$ 的意思是说首先使 y 和 z 进行或运算。或运算是指如果 y 和 z 中有一个为真，则 $(y \mid\mid z)$ 为真，返回一个布尔值 1。这时 y、z 都是真，所以为真，返回 1。所以 x 为 1。

答案：10, 10, 1, 3, 1, 7, 1。

面试例题 3：以下代码结果是多少？[中国某杀毒软件公司 2010 年 3 月笔试题]

```
#include <iostream>
using namespace std;
int func(int x)
{
    int count = 0;
    while(x)
    {
        count++;
        x=x&(x-1);
    }
    return count;
}

int main()
{
    cout << func(9999) << endl;
    return 0;
}
```

- A. 8 B. 9 C. 10 D. 11

解析：本题 func 函数返回值是形参 x 转化成二进制后包含 1 的数量。理解这一点就很容易答出来了。9999 转化为二进制是：

9999: 10011100001111

答案：A

5.2 i++

面试例题 1：下面两段代码的输出结果有什么不同？[中国著名网络企业 XL 公司 2007 年 10 月面试题]

第 1 段：

```
#include<iostream>
using namespace std;
int main()
{
    int a,x;
    for(a=0,x=0;a<=1 && !x++;a++)
    {
        a++;
    }
    cout<<a<<x<<endl;
    return 0;
}
```

第 2 段：

```
#include<iostream>
using namespace std;
```

```

int main()
{
int a,x;
for(a=0,x=0;a<=1 &&!x++;)
{
    a++;
}

cout<<a<<x<<endl;
return 0;
}

```

解析：这两段代码的不同点就在 for 循环那里，前者是 for($a=0, x=0; a \leq 1 \ \&\& !x++; a++$)，后者是 for($a=0, x=0; a \leq 1 \ \&\& !x++;$)。

先说第 1 段代码。

第 1 步：初始化定义 $a=0, x=0$ 。

第 2 步： a 小于等于 1， x 的非为 1，符合循环条件。

第 3 步： $x++$ 后 x 自增为 1。

第 4 步：进入循环体， $a++$ ， a 自增为 1。

第 5 步：执行 for($a=0, x=0; a \leq 1 \ \&\& !x++; a++$) 中的 $a++$ ， a 自增为 2。

第 6 步： a 现在是 2，已经不符合小于等于 1 的条件了，所以 “ $\&\&$ ” 后面的 “ $!x++$ ” 不执行， x 还是 1，不执行循环体。

第 7 步：打印 a 和 b ，分别是 2 和 1。

再说第 2 段代码。

第 1 步：初始化定义 $a=0, x=0$ 。

第 2 步： a 小于等于 1， x 的非为 1，符合循环条件。

第 3 步： $x++$ 后 x 自增为 1。

第 4 步：进入循环体， $a++$ ， a 自增为 1。

第 5 步： a 现在是 1，符合小于等于 1 的条件，所以 “ $\&\&$ ” 后面的 “ $!x++$ ” 被执行， x 现在是 1， x 的非为 0，不符合循环条件，不执行循环体，但 $x++$ 依然执行，自增为 2。

第 6 步：打印 a 和 b ，分别是 1 和 2。

答案：第一段输出结果是 21，第二段输出结果是 12。

面试例题 2：What will be the output of the following C code? (以下代码的输出结果是什么?)
[中国著名通信企业 H 公司 2007 年 7 月面试题]

```

#include <stdio.h>

main()
{

```

```

    int b=3;

    int arr[]={6,7,8,9,10};

```

```
Sint *ptr=arr;
*(ptr++)+=123;
```

```
printf( "%d,%d\n ",*ptr,*(++ptr));
}
```

- A. 8 8 B. 130 8 C. 7 7 D. 7 8

解析：C 中 printf 计算参数时是从右到左压栈的。

几个输出结果分别如下：

printf("%d\n ",*ptr); 此时 ptr 应指向第一个元素 6。

*(ptr++)+=123 应为*ptr=*ptr+123;ptr++, 此时 ptr 应指向第二个元素 7。

printf("%d\n ",*(ptr-1)); 此时输出第一个元素 129，注意此时是经过计算的。

printf("%d\n ",*ptr); 此时输出第二个元素 7，此时 ptr 还是指向第二个元素 7。

printf("%d,%d\n ",*ptr,*(++ptr)); 从右到左运算，第一个是(++ptr)，也就是 ptr++, *ptr=8，此时 ptr 指向第三个元素 8，所以全部为 8。

答案：A

5.3 编程风格

面试例题：We have two pieces of code , which one do you prefer, and tell why. (下面两段程序有两种写法，你青睐哪种，为什么？) [美国某著名计算机嵌入式公司 2005 年 10 月面试题]

A.

```
// a is a variable
```

写法 1：

```
if( 'A'==a ) {
    a++;
}
```

写法 2：

```
if( a=='A' ) {
    a++;
}
```

B.

写法 1：

```
for(i=0;i<8;i++) {
    X= i+Y+J*7;
    printf("%d",x);
}
```

写法 2：

```
S= Y+J*7;
for(i=0;i<8;i++) {
    printf("%d",i+S);
}
```

答案：

- A. 第一种写法'A'==a 比较好一些。这时如果把“==”误写成“=”的话，因为编译器不允许对常量赋值，就可以检查到错误。
- B. 第二种写法好一些，将部分加法运算放到了循环体外，提高了效率。缺点是程序不够简洁。

5.4 类型转换

面试例题 1：下面程序的结果是多少？[中国著名通信企业 S 公司 2007 年 8 月面试题]

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;
int main()
{
float a = 1.0f;
cout << (int)a << endl;
cout << &a << endl;
cout << (int&)a << endl;
cout << boolalpha << ( (int)a == (int&)a )
    << endl;           //输出什么？
float b = 0.0f;
cout << (int)b << endl;
cout << &b << endl;
cout << (int&)b << endl;
cout << boolalpha << ( (int)b == (int&)b )
    << endl;           //输出什么？
return 0;
```

解析：在机器上运行一下，可以得到结果，“`cout << (int&)a << endl;`”输出的是 1065353216，而不是 1。这是因为浮点数在内存里和整数的存储方式不同，`(int&)a` 相当于将该浮点数地址开始的 `sizeof(int)` 个字节当成 int 型的数据输出，因此这取决于 float 型数据在内存中的存储方式，而不是经过`(int&)a` 显示转换的结果（1）。

因为 `float a = 1.0f` 在内存中的表示都是 3f800000，而浮点数和一般整型不一样，所以当`(int&)a` 强制转换时，会把内存值 3f8000000 当作 int 型输出，所以结果自然变为了 1065353216（0x3f800000 的十进制表示）。

答案：false true 或者 0 1。

面试例题 2：下面程序的结果是多少？[中国著名通信企业 S 公司 2007 年 8 月面试题]

```
#include <stdio.h>

int main()
{
    unsigned int a = 0xFFFFFFFF7;
    unsigned char i = (unsigned char)a;
    char* b = (char*)&a;

    printf("%08x, %08x", i,*b);
}
```

解析：在 X86 系列的机器中，数据的存储是“小端存储”，小端存储的意思就是，对于一个跨多个字节的数据，其低位存放在低地址单元，其高位存放在高地址单元。比如一个 int 型的数据 ox12345678，假如存放在 0x00000000, 0x00000001, 0x00000002, 0x00000003 这四个内存单元中，那么 0x00000000 中存放的是低位的 ox78，而 0x00000003 中存放的是高位的 0x12，依此类推。

有了以上的认识，继续分析上面的程序为什么输出 ffffff7：char* b = (char*)&a;这句话到底干了什么事呢？其实说来也简单，&a 可以认为是个指向 unsigned int 类型数据的指针，(char *) &a 则把&a 强制转换成 char *类型的指针，并且这个时候发生了截断！截断后，指针 b 只指向 0xf7 这个数据（为什么 b 指向最低位的 0xf7 而不是最高位的 0xff？想想上面刚刚讲过的“小端存储”，低地址单元存放低位数据），又由于指针 b 是 char *型的，属于有符号数，所以有符号数 0xf7 在 printf () 的作用下输出 ffffff7。

或者我们可以通过汇编代码更直观地看内部的情况：

```
int main()
{
01321380 push    ebp
01321381 mov     ebp,esp
01321383 sub     esp,0E4h
01321389 push    ebx
0132138A push    esi
0132138B push    edi
0132138C lea     edi,[ebp-0E4h]
01321392 mov     ecx,39h
01321397 mov     eax,0CCCCCCCCCh
0132139C rep stos  dword ptr es:[edi]
        unsigned int a = 0xFFFFFFFF65;
0132139E mov     dword ptr [a],0FFFFFFF7h
        unsigned char i = (unsigned char)a;
013213A5 mov     al,byte ptr [a]
013213A8 mov     byte ptr [i],al
        char* b = (char*)&a;
013213AB lea     eax,[a]           //取 a 的地址: 0x0018FD70
```

```

013213AE mov      dword ptr [b],eax //指针b的值为: 0x0018FD70, 该位置放着0xF7;

        printf("%08x, %08x\n", i, *b);
013213B1 mov      eax,dword ptr [b] //把b的值,也就是0x0018FD70放到EAX中;
013213B4 movsx    ecx,byte ptr [eax] //这句话最关键, byte ptr [eax]就是把0xF7取出来,注意命令是
byte ptr。然后movsx指令是按符号扩展,放到ecx中,按符号扩展其实就是将char扩展成int,然后printf中格式说明的
'x'则说明将这个int按16进制输出,也就是fffffff7,而如果将'x'变成'd',按整数输出,那么程序就会输出-9
013213B7 mov      esi,esp           //上一句的byte ptr就反映了我们上面说的 char* b =
(char*)&a 截取的问题
013213B9 push    ecx
013213BA movzx   edx,byte ptr [i] //注意因为i是unsigned char(无符号),所以按0扩展成unsigned
int
013213BE push    edx
013213BF push    offset string "%08x, %08x\n" (1325830h)
013213C4 call    dword ptr [_imp__printf (13282B0h)]
013213CA add     esp,0Ch
013213CD cmp     esi,esp
013213CF call    @ILT+295(__RTC_CheckEsp) (132112Ch)
}

```

答案: 000000f7, ffffff7。

扩展知识

C++定义了一组内置类型对象之间的标准转换，在必要时它们被编译器隐式地应用到对象上。

隐式类型转换发生在下列这些典型情况下。

1. 在混合类型的算术表达式中

在这种情况下最宽的数据类型成为目标转换类型，这也被称为算术转换(Arithmetic Conversion)，例如：

```

int ival = 3;
double dval = 3.14159;

// ival 被提升为 double 类型: 3.0
ival + dval;

```

2. 用一种类型的表达式赋值给另一种类型的对象

在这种情况下目标转换类型是被赋值对象的类型。例如在下面第一个赋值中文字常量0的类型是int。它被转换成int*型的指针表示空地址。在第二个赋值中double型的值被截取成int型的值。

```

// 0 被转换成 int*类型的空指针值
int *pi = 0;
// dval 被截取为 int 值 3
ival = dval;

```

3. 把一个表达式传递给一个函数，调用表达式的类型与形式参数的类型不相同
在这种情况下目标转换类型是形式参数的类型。例如：

```
extern double sqrt( double );
// 2 被提升为 double 类型 2.0
cout << "The square root of 2 is " << sqrt( 2 ) << endl;
```

4. 从一个函数返回一个表达式的类型与返回类型不相同

在这种情况下返回的表达式类型自动转换成函数类型。例如：

```
double difference( int ival1,
                    int ival2 )
{
    // 返回值被提升为 double 类型
    return ival1 - ival2;
}
```

算术转换保证了二元操作符，如加法或乘法的两个操作数被提升为共同的类型，然后再用它表示结果的类型。两个通用的指导原则如下：

- (1) 为防止精度损失，如果必要的话，类型总是被提升为较宽的类型。
- (2) 所有含有小于整型的有序类型的算术表达式在计算之前其类型都会被转换成整型。

规则的定义如上面所述，这些规则定义了一个类型转换层次结构。我们从最宽的类型 long double 开始。

如果一个操作数的类型是 long double，那么另一个操作数无论是什么类型都将被转换成 long double。例如在下面的表达式中，字符常量小写字母 a 将被提升为 long double，它的 ASC 码值为 97，然后再被加到 long double 型的文字常量上：

```
3.14159L + 'a';
```

如果两个操作数都不是 long double 型，那么若其中一个操作数的类型是 double 型，则另一个就将被转换成 double 型。例如：

```
int ival;
float fval;
double dval;
// 在计算加法前 fval 和 ival 都被转换成 double
dval + fval + ival;
```

类似地，如果两个操作数都不是 double 型而其中一个操作数是 float 型，则另一个被转换成 float 型。例如：

```
char cval;
int ival;
float fval;
// 在计算加法前 ival 和 cval 都被转换成 double
cval + fval + ival;
```

否则如果两个操作数都不是3种浮点类型之一，它们一定是某种整值类型。在确定共同的目标提升类型之前，编译器将在所有小于int的整值类型上施加一个被称为整值提升（integral promotion）的过程。

在进行整值提升时类型char、signed char、unsigned char和short int都被提升为类型int。如果机器上的类型空间足够表示所有unsigned short型的值，这通常发生在short用半个字而int用一个字表示的情况下，则unsigned short int也被转换成int，否则它会被提升为unsigned int。wchar_t和枚举类型被提升为能够表示其底层类型（underlying type）所有值的最小整数类型。例如已知如下枚举类型：

```
enum status { bad, ok };
```

相关联的值是0和1。这两个值可以但不是必须存放在char类型的表示中。当这些值实际上被作为char类型来存储时，char代表了枚举的底层类型，然后status的整值提升将它的底层类型转换为int。

在下列表达式中：

```
char cval;
bool found;
enum mumble { m1, m2, m3 } mval;
unsigned long ulong;
cval + ulong; ulong + found;
mval + ulong;
```

在确定两个操作数被提升的公共类型之前，cval found 和 mval 都被提升为int类型。

一旦整值提升执行完毕，类型比较就又一次开始。如果一个操作数是unsigned long型，则第二个也被转换成unsigned long型。在上面的例子中所有被加到ulong上的3个对象都被提升为unsigned long型。如果两个操作数的类型都不是unsigned long而其中一个操作数是long型，则另一个也被转换成long型。例如：

```
char cval;
long lval;
// 在计算加法前cval和1024都被提升为long型
cval + 1024 + lval;
```

long类型的一般转换有一个例外。如果一个操作数是long型而另一个是unsigned int型，那么只有机器上的long型的长度足以存放unsigned int的所有值时（一般来说，在32位操作系统中long型和int型都用一个字长表示，所以不满足这里的假设条件），unsigned int才会被转换为long型，否则两个操作数都被提升为unsigned long型。若两个操作数都不是long型而其中一个是unsigned int型，则另一个也被转换成unsigned int型，否则两个操作数一定都是int型。

尽管算术转换的这些规则带给你的困惑可能多于启发，但是一般的思想是尽可能地保留多类型表达式中涉及的值的精度。这正是通过把不同的类型提升到当前出现的最宽的类型来实现的。

5.5 运算符问题

面试例题 1：下面程序的结果是多少？[中国台湾某著名 CPU 生产公司 2010 年 7 月面试题]

```
#include <iostream>
using namespace std;

int main()
{
    unsigned char a=0xA5;
    unsigned char b=~a>>4+1;
    //cout <<b;
    printf("b=%d\n",b);
    return 0;
}
```

- A. 245 B. 246 C. 250 D. 2

解析：这道题目考查两个知识点：一是类型转换问题；二是算符的优先级问题。

对于第一个问题：`unsigned char b=~a>>4`，在计算这个表达式的时候，编译器会先把 a 和 4 的值转换为 int 类型（即所谓整数提升）后再进行计算，当计算结果出来后，再把结果转换成 `unsigned char` 赋值给 b。

对于第二个问题：因为“~”的优先级高于“>>”和“+”，本题的过程是这样的：先对于 1010 0101 取反 0101 1010；再右移，这里有一个问题，是先右移 4 位再加 1 呢，还是直接右移 5（4+1）位。因为“+”的优先级高于“>>”，所以直接右移 5 位。结果是 0000 0010。

`~a` 操作时，会对 a 进行整型提升，a 是无符号的，提升时左边补 0（一般机器 32 位，char 是 8 位，左边 24 个 1；16 位 int 则左边补 8 个 0），取反后左边为 1，右移就把左边的 1 都移到右边（注意是算术移位），再按照无符号读取，才有 250 这个结果。

答案：C

扩展知识

运算符优先级如下表所示。

优先级	运算符	案例	结合性	优先级	运算符	案例	结合性
1	0 [] -> . . :: ++ --	(a + b) / 4; array[4] = 2; ptr->age = 34; obj.age = 34; Class::age = 2; for(i = 0; i < 10; i++) ... for(i = 10; i > 0; i--) ...	从左向右	7	< <=	if(i < 42) ... if(i <= 42) ... if(i > 42) ... if(i >= 42) ...	从左向右
				8	== !=	if(i == 42) ... if(i != 42) ...	从左向右
				9	&	flags = flags & 42;	从左向右
				10	^	flags = flags ^ 42;	从左向右
				11		flags = flags 42;	从左向右
				12	&&	if(conditionA && conditionB) ...	从左向右
2	! ~ ++ -- - + * & (type) sizeof	if(!done) ... flags = ~flags; for(i = 0; i < 10; ++i) ... for(i = 10; i > 0; --i) ... int i = -1; int i = +1; data = *ptr; address = &obj; int i = (int) floatNum; int size = sizeof(floatNum);	从右向左	13		if(conditionA conditionB) ...	从左向右
				14	? :	int i = (a > b) ? a : b;	从右向左
				15	= += -= *= /= %= &= ^= = <=>	int a = b; a += 3; b -= 4; a *= 5; a /= 2; a %>= 3; flags &= new_flags; flags ^= new_flags; flags = new_flags; flags <<= 2; flags >>= 2;	从右向左
3	->*	ptr->*var = 24; obj.*var = 24;	从左向右				
4	* / %	int i = 2 * 4; float f = 10 / 3; int rem = 4 % 3;	从左向右				
5	+	int i = 2 + 3; int i = 5 - 1;	从左向右				
6	<< >>	int flags = 33 << 1; int flags = 33 >> 1;	从左向右	16	,	for(i = 0, j = 0; i < 10; i++, j++) ...	从左向右

面试例题 2：用一个表达式，判断一个数 X 是否是 2^N 次方($2, 4, 8, 16, \dots$)，不可用循环语句。[中国台湾某著名 CPU 生产公司 2007 年 10 月面试题]

解析：2、4、8、16 这样的数转化成二进制是 10、100、1000、10000。如果 X 减 1 后与 X 做运算，答案若是 0，则 X 是 2^N 次方。

答案：!(X&(X - 1))

面试例题 3：下面代码：

```
int f(int x, int y)
{
    return(x&y)+( (x^y)>>1)
}
```

(729,271)=_____

解析：这道题如果使用笨办法来求解，就都转化成二进制然后按位与。但这样的做法显然不是面试官所期待的。仔细观察一下题目， $x \& y$ 是取相同的位与，这个的结果是 x 和 y 相同位的和的一半， $x \wedge y$ 是取 x 和 y 的不同位，右移相当于除以 2，所以这个函数的功能是取

两个数的平均值。 $(729+271)/2=500$ 。

答案：500

面试例题4：利用位运算实现两个整数的加法运算，请用代码实现。

答案：代码如下：

```
int Add(int a,int b)
{
    if(b == 0) return a;//没有进位的时候完成运算
    int sum,carry;
    sum = a ^ b;//完成第一步没有进位的加法运算
    carry=(a & b) << 1;//完成第二步进位并且左移运算
    return Add(sum,carry);//进行递归，相加
}
```

5.6 a、b 交换与比较

面试例题1：There are two int variables: a and b, don't use “if”, “?:”, “switch” or other judgement statements, find out the biggest one of the two numbers.(有两个变量a和b,不用“if”、“?:”、“switch”或其他判断语句,找出两个数中间比较大的。) [美国某著名网络开发公司2005年面试题]

答案：方案一：

```
int max = ((a+b)+abs(a-b)) / 2
```

方案二：

```
int c = a -b;
char *strs[2] = {"a Large ","b Large "};
c = unsigned(c) >> (sizeof(int) * 8 - 1);
```

上面的情况没有考虑溢出的情况,如果考虑溢出的话需要加额外的判断。

面试例题2：两个整型数,不准用while,if,for,switch,?:等判断语句求出两者最大值。

答案：代码如下,可以采用bool值:

```
bool fun(int a, int b)
{
    return a>b;
}
int max(int a, int b)
{
    bool flag = fun(a, b);
    return flag*a + (1-flag)*b;
}
```

面试例题3：有2数据,写一个交换数据的宏?

解析：如果用异或语句，无须担心超界的问题

这样做的原理是按位异或运算。按位异或运算符“ \wedge ”是双目运算符，其功能是参与运算的两数各对应的二进制位相异，或当对应的二进制位相异时结果为 1。参与运算数仍以补码形式出现。例如 $9 \wedge 5$ 可写成如下算式：

```
00001001^00000101 00001100 (十进制数为12)
main() {
    int a=9;
    a=a^5;
    printf("a=%d\n",a);
}
00001001^00000101 得到 00001100。
00001001^00001100 得到 00000101。
00001100^00000101 得到 00001001。
```

但是如果这个数据是浮点的话，就不好处理了（浮点数不能进行位运算），正确的做法是采用内存交换。

答案：

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

#define swap(a,b) \
{ char tempBuf[10]; memcpy(tempBuf,&a,sizeof(a)); \
memcpy(&a,&b,sizeof(b)); memcpy(&b,tempBuf,sizeof(b)); }

int main()
{
    double a=2,b=3;

    swap(a,b);

    printf("%lf %lf \n",a,b);

    return 0;
}
```

5.7 C 和 C++ 的关系

面试例题 1：在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 `extern "C"`？

答案：C++ 语言支持函数重载，C 语言不支持函数重载。函数被 C++ 编译后在库中的名字与 C 语言的不同。假设某个函数的原型为 `void foo(int x, int y)`。该函数被 C 编译器编译后在库中的名字为 `_foo`，而 C++ 编译器则会产生像 `_foo_int_int` 之类的名字。

C++ 提供了 C 连接交换指定符号 `extern "C"` 解决名字匹配问题。

面试例题 2：头文件中的 ifndef/define/endif 是干什么用的？

答案：头文件中的 ifndef/define/endif 是条件编译的一种，除了头文件被防止重复引用（整体），还可以防止重复定义（变量、宏或者结构）。

面试例题 3：评价一下 C 与 C++的各自特点。如果一个程序既需要大量运算，又要有一个好的用户界面，还需要与其他软件大量交流，应该怎样选择合适的语言？

答案：C 是一种结构化语言，重点在于算法和数据结构。C 程序的设计首先考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出（或实现过程（事务）控制）。而对于 C++，首先考虑的是如何构造一个对象模型，让这个模型能够契合与之对应的问题域，这样就可以通过获取对象的状态信息得到输出或实现过程（事务）控制。

对于大规模数值运算，C/C++和 Java/.NET 之间没有明显的性能差异。不过，如果运算设计向量计算、矩阵运算，可以使用 FORTRAN 或者 MATLAB 编写计算组件（如 COM）。

大规模用户界面相关的软件可以考虑使用.NET 进行开发（Windows 环境下），而且.NET 同 COM 之间的互操作十分容易，同时.NET 对数据库访问的支持也相当好。

5.8 程序设计的其他问题

面试例题 1：下面的 switch 语句输出什么。[日本著名软件企业 F 公司 2013 年 2 月面试题]

```
int n='c';
switch(n++)
{default:printf("error");break;
case 'a':case'A':case 'b':case'B':printf("ab");break;
case 'c':case 'C':printf("c");
case 'd':case 'D':printf("d");}
```

- A. cdd B. cd C. abcd D. cderror

解析：本题考的是 switch 中的“fall through”：如果 case 语句后面不加 break，就依次执行下去。

所以先顺序执行，考虑 n 的初始值，从'c'开始查找输出(default 和 ab 直接略过)，输出 c；没有 break，那么继续输出后面的，输出 d。

答案：B

面试例题 2：上机题目描述：选秀节目打分，分为专家评委和大众评委，score[] 数组里面存储每个评委打的分数，judge_type[] 里存储与 score[] 数组对应的评委类别，judge_type == 1，表示专家评委，judge_type == 2，表示大众评委，n 表示评委总数。打分规则如下：专家评委

和大众评委的分数先分别取一个平均分(平均分取整),然后,总分 = 专家评委平均分 * 0.6 + 大众评委 * 0.4, 总分取整。函数最终返回选手得分。[中国著名通信企业 H 公司 2013 年 2 月面试题]

函数接口 int cal_score(int score[], int judge_type[], int n)

解析: 上机题目都是很简单的,但是考的就是考虑问题全面与否。

答案: 代码如下:

```
int CallScore(int N,int *Score,int *Judge_type)
{
    int ret=0,n=0,m=0;
    double sum1=0,sum2=0; //评分可能出现小数,所以要用双精度
    if(N&&Score&&Judge_type){
        for(int i=0;i<N;++i)
            switch(Judge_type[i]){
                case 1: sum1 += Score[i];++n;break;
                case 2: sum2 += score[i];++m;break;
                default://----舍弃不符要求数据
            }
        if(n)sum1=int(sum1 / n); //考虑到专家人数可能为0,务必确保除数不为0;
        if(m)sum2 = int (sum2/m); //考虑到大众人数可能为0,务必确保除数不为0;
        ret = m?sum1*0.6+sum2*0.4:sum1;//最后总分取整。要把double转化成int
    }
    return ret;
}
```

第 6 章

预处理、const 与 sizeof

预 处理问题、const 问题和 sizeof 问题是 C++ 设计语言中的三大难点，也是各大企业面试中反复出现的问题。就 sizeof 问题而言，我们曾在十几家公司、几十套面试题目中发现它的存在。所以本章把这三大问题单独提出来，并结合详细的分析和解释来阐述各个知识点。

6.1 宏定义

面试例题 1：下面代码输出结果是多少？[美国著名搜索引擎公司 G 2012 年秋季校园招聘题目]

```
#define SUB(x,y) x-y
#define ACCESS_BEFORE(element,offset,value) *SUB(&element, offset) =value
int main(){
    int i; int array[10] = {1,2,3,4,5,6,7,8,9,10};
    ACCESS_BEFORE(array[5], 4, 6);
    for (i=0; i<10; ++i) { printf("%d", array[i]); }
    return (0);
}
```

- A. array: 1 6 3 4 5 6 7 8 9 10
- B. array: 6 2 3 4 5 6 7 8 9 10
- C. 程序可以正确编译，但是运行时会崩溃
- D. 程序语法错误，编译不成功

解析：宏的那句被预处理器替换成：*&array[5]-4 = 6；

由于减号比赋值优先级高，因此先处理减号；由于减号返回一个数而不是合法的值，所以编译报错。

答案：C

扩展知识：请思考#define SUB(x,y) (x-y)的情况。

面试例题 2：用预处理指令#define 声明一个常数，用以表明 1 年中有多少秒（忽略闰年问题）。

[美国某著名计算机嵌入式公司 2005 年面试题]

解析：

通过这道题面试官想考以下几个知识点：

- #define 语法的基本知识（例如，不能以分号结束、括号的使用，等等）。
- 要懂得预处理器将为你计算常数表达式的值，因此，写出你是如何计算一年中有多少秒而不是计算出实际的值，会更有意义。
- 意识到这个表达式将使一个 16 位机的整型数溢出，因此要用到长整型符号 L，告诉编译器这个常数是长整型数。

如果在表达式中用到 UL（表示无符号长整型），那么你就有了一个好的起点。记住，第一印象很重要。

答案：

```
#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

面试例题 3：写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。[美国某著名计算机嵌入式公司 2005 年面试题]

解析：

这个测试是为下面的目的而设的：

- 标识#define 在宏中应用的基本知识。这是很重要的，因为直到嵌入（inline）操作符变为标准 C 的一部分，宏都是方便地产生嵌入代码的唯一方法。对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。
- 三重条件操作符的知识。这个操作符存在 C 语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码，了解这个用法是很重要的。
- 懂得在宏中小心地把参数用括号括起来。

答案：

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))。
```

6.2 const

面试例题 1：Which "const" modifier should be removed （下面哪个 const 应该被移除）？[美国某著名软件开发公司 2013 年面试题]

```
const bufsize=100;
#include <windows.h>
#include <iostream>
```

```

#define BUF_SIZE 30

using namespace std;

class A
{
public:
    A();
    ~A() {};
public:
    inline const[A] BYTE* GetBuffer() const[B] {return m_pBuf; }
    int Pop(void);
private:
    const[C] BYTE * const[D] m_pBuf;
};

A::A():m_pBuf()
{
    BYTE* pBuf = new BYTE[BUF_SIZE];
    if(pBuf == NULL)
        return;

    for(int i = 0; i < BUF_SIZE; i++)
    {
        pBuf[i]=i;
    }
    m_pBuf = pBuf;
}

int main()
{
    A a;
    Const[E] BYTE* pB = a.GetBuffer();
    if(pB != NULL)
    {
        for(int i = 0; i< BUF_SIZE; i++)
        {
            printf("%u", pB[i++]);
        }
    }
    system("pause");
    return 0;
}

```

解析：关于 const 修饰指针的情况，一般分为如下 4 种情况：

```

int b = 500;
const int* a = &b           //情况 1
int const *a = &b          //情况 2
int* const a = &b          //情况 3
const int* const a = &b     //情况 4

```

如何区别呢？

1) 先看情况1。

如果 const 位于星号的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量；如果 const 位于星号的右侧，const 就是修饰指针本身，即指针本身是常量。因此，1 和 2 的情况相同，都是指针所指向的内容为常量（与 const 放在变量声明符中的位置无关），这种情况下不允许对内容进行更改操作。

换句话来说，如果 a 是一名仓库管理员的话，他所进入的仓库，里面的货物(*a)是他没权限允许动的，仓库里面的东西原来是什么就是什么；所以

```
int b = 500;
const int* a = &b;
*a = 600; // 错误
```

但是也有别的办法去改变*a 的值，一个是通过改变 b 的值：

```
int b = 500;
const int* a = &b
b = 600;
cout << * a << endl // 得到 600。
```

还有一种改变*a 办法就是 a 指向别处(管理员换个仓库)：

```
int b = 500, c = 600;
const int* a = &b;
a = &c;
cout << * a << endl // 得到 600
```

对于情况1，可以先不进行初始化。因为虽然指针内容是常量，但指针本身不是常量。

```
const int* a; // 正确
```

2) 情况2与情况1相同。

3) 情况3为指针本身是常量，这种情况下不能对指针本身进行更改操作，而指针所指向的内容不是常量。

举例来说：如果 a 是一名仓库管理员的话，他只能进入指定的某仓库，而不能去别的仓库(所以 a++是错误的)；但这个仓库里面的货物(*a)是可以随便动的，(*a=600 是正确的)。

此外，对于情况3：定义时必须同时初始化。

```
int b = 500, c = 600;
int* const a; // 错误 没有初始化
int* const a = &b; // 正确 必须初始化
*a = 600; // 正确，允许改值
cout << a++ << endl; // 错误
```

4) 对于情况4为指针本身和指向的内容均为常量。那么这个仓库管理员只能去特定的仓库，并且仓库里面所有的货物他都没有权限去改变。

下面再说一下 const 成员函数是什么？

我们定义的类的成员函数中，常常有一些成员函数不改变类的数据成员，也就是说，这些函数是“只读”函数，而有一些函数要修改类数据成员的值。如果把不改变数据成员的函数都加上 const 关键字进行标识，显然，可提高程序的可读性。其实，它还能提高程序的可靠性，已定义成 const 的成员函数，一旦企图修改数据成员的值，则编译器按错误处理。

一些成员函数改变对象，例如：

```
void Point:: SetPt (int x, int y)
{
    xVal=x;
    yVal=y;
}
```

一些成员函数不改变对象。

```
int Point::GetY()
{
    return yVal;
}
```

为了使成员函数的意义更加清楚，我们可在不改变对象的成员函数的函数原型中加上 const，下面是定义 const 成员函数的一个实例：

```
class Point
{
    int xVal, yVal;
public:
    int GetY() const;
};

//关键字 const 必须用同样的方式重复出现在函数实现里，否则编译器会把它看成一个不同的函数：
int Point::GetY() const

{
    return yVal;
}
```

如果 GetY() 试图用任何方式改变 yVal 或调用另一个非 const 成员函数，编译器将给出错误信息。任何不修改成员数据的函数都应该声明为 const 函数，这样有助于提高程序的可读性和可靠性。

如果把 const 放在函数声明前呢？因为这样做意味着函数的返回值是常量，意义就完全不同了。

本题中，选项 A 修饰函数返回值，表示返回的是指针所指向值是常量；选项 B 的 const 这样的函数是常成员函数。常成员函数可以理解为是一个“只读”函数，它既不能更改数据成员的值，也不能调用那些能引起数据成员值变化的成员函数，只能调用 const 成员函数，把

不会修改数据成员的函数 GetBuffer 声明为 const 类型。这大大提高了程序的健壮性。

选项 D 显然不对因为如果存在 const BYTE * const m_pBuf 的情况，势必要进行初始化。

答案：D。

面试例题 2：const 与#define 相比有什么不同？

答案：C++语言可以用 const 定义常量，也可以用#define 定义常量，但是前者比后者有更多的优点：

- const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换中可能会产生意料不到的错误（边际效应）。
- 有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。在 C++程序中只使用 const 常量而不使用宏常量，即 const 常量完全取代宏常量。

扩展知识

常量的引进是在早期的 C++ 版本中，当时标准 C 规范正在制订。那时，常量被看作一个好的思想而被包含在 C 中。但是，C 中的 const 的意思是“一个不能被改变的普通变量”。在 C 中，它总是占用内存，而且它的名字是全局符。C 编译器不能把 const 看成一个编译期间的常量。在 C 中，如果写：

```
const bufsize=100;
char buf[bufsize];
```

尽管看起来好像做了一件合理的事，但这将得到一个错误的结果。因为 bufsize 占用内存的某个地方，所以 C 编译器不知道它在编译时的值。在 C 语言中可以选择这样书写：

```
const bufsize;
```

这样写在 C++ 中是不对的，而 C 编译器则把它作为一个声明，这个声明指明在别的地方有内存分配。因为 C 默认 const 是外部连接的，C++ 默认 const 是内部连接的，这样，如果在 C++ 中想完成与 C 中同样的事情，必须用 extern 把内部连接改成外部连接：

```
extern const bufsize;//declaration only
```

这种方法也可用在 C 语言中。在 C 语言中使用限定符 const 不是很有用，即使是在常数表达式里（必须在编译期间被求出）想使用一个已命名的值，使用 const 也不是很有用的。C 迫使程序员在预处理器里使用#define。

面试例题3：有类如下：

```
Class A_class
{
    void f() const
    {
        ....
    }
}
```

在上面这种情况下，如果要修改类的成员变量，应该怎么办？[美国著名软件企业GS公司2007年12月面试题]

解析：在C++程序中，类里面的数据成员加上mutable后，修饰为const的成员变量，就可以修改它了，代码如下：

```
#include <iostream>
#include <iomanip>
using namespace std;

class C
{
public:
    C(int i):m_Count(i){}
    int incr() const
    //注意这里的const
    {
        return ++m_Count;
    }
    int decr() const
    {
        return --m_Count;
    }
}
```

```
private:
mutable int m_Count;
//可以将这里的mutable去掉再编译试一试
};

int main()
{
    C c1(0),c2(10);
    for(int tmp,i=0;i<10;i++)
    {
        tmp = c1.incr();
        cout<<setw(tmp)<<setfill('
')<<tmp<<endl;
        tmp = c2.decr();
        cout<<setw(tmp)<<setfill('
')<<tmp<<endl;
    }
    return 0;
}
```

答案：在const成员函数中，用mutable修饰成员变量名后，就可以修改类的成员变量了。

6.3 sizeof

面试例题1：What is the output of the following code?（下面代码的输出结果是什么？）[美国某著名计算机软硬件公司2005年、2007年面试题]

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
struct{
short a1;
short a2;
short a3;
}A;
struct{
```

```
long a1;
short a2;
}B;

int main()
{
    char* ss1 = "0123456789";
    char ss2[] = "0123456789";
    char ss3[100] = "0123456789";
    int ss4[100] ;
```

```

char q1[]="abc";
char q2[]="a\n";
char* q3="a\n";
char *str1 = (char *)malloc(100);

void *str2 = (void *) malloc(100);

cout << sizeof(ss1) << " ";
cout << sizeof(ss2) << " ";
cout << sizeof(ss3) << " ";
cout << sizeof(ss4) << " ";

```

```

cout << sizeof(q1) << " ";
cout << sizeof(q2) << " ";
cout << sizeof(q3) << " ";
cout << sizeof(A) << " ";
cout << sizeof(B) << " ";
cout << sizeof(str1) << " ";
cout << sizeof(str2) << " ";

return 0;
}

```

解析：

ss1 是一个字符指针，指针的大小是一个定值，就是 4 字节，所以 sizeof(ss1) 是 4 字节。

ss2 是一个字符数组，这个数组最初未定大小，由具体填充值来定。填充值是“0123456789”。1 个字符所占空间是 1 字节，10 个就是 10 字节，再加上隐含的“\0”，所以一共是 11 字节。

ss3 也是一个字符数组，这个数组开始预分配 100，所以它的大小一共是 100 字节。

ss4 也是一个整型数组，这个数组开始预分配 100，但每个整型变量所占空间是 4，所以它的大小一共是 400 字节。

q1 与 ss2 类似，所以是 4 字节。

q2 里面有一个“\n”，“\n”算作一位，所以它的空间大小是 3 字节。

q3 是一个字符指针，指针的大小是一个定值，就是 4，所以 sizeof(q3) 是 4 字节。

A 和 B 是两个结构体。在默认情况下，为了方便对结构体内元素的访问和管理，当结构体内的元素的长度都小于处理器的位数的时候，便以结构体里面最长的数据元素为对齐单位，也就是说，结构体的长度一定是最长的数据元素的整数倍。如果结构体内存在长度大于处理器位数的元素，那么就以处理器的位数为对齐单位。但是结构体内类型相同的连续元素和数组一样，将在连续的空间内。

结构体 A 中有 3 个 short 类型变量，各自以 2 字节对齐，结构体对齐参数按默认的 8 字节对齐，则 a1、a2、a3 都取 2 字节对齐，sizeof(A) 为 6，其也是 2 的整数倍。B 中 a1 为 4 字节对齐，a2 为 2 字节对齐，结构体默认对齐参数为 8，则 a1 取 4 字节对齐，a2 取 2 字节对齐；结构体大小为 6 字节，6 不为 4 的整数倍，补空字节，增到 8 时，符合所有条件，则 sizeof(B) 为 8。

CPU 的优化规则大致原则是这样的：对于 n 字节的元素 ($n=2,4,8,\dots$)，它的首地址能被 n 整除，才能获得最好的性能。设计编译器的时候可以遵循这个原则：对于每一个变量，可以从当前位置向后找到第一个满足这个条件的地址作为首地址。例子比较特殊，因为即便采

用这个原则，得到的结果也应该为 6 字节（long 的首地址偏移量 0000，short 首地址偏移量 0004，都符合要求）。但是结构体一般会面临数组分配的问题。编译器为了优化这种情况，干脆把它的大小设为 8 字节，这样就没有麻烦了，否则的话，会出现单个结构体的大小为 6 字节，而大小为 n 的结构体数组大小却为 $8 \times (n - 1) + 6$ 的尴尬局面。IBM 出这道题并不是考查理解语言本身和编译器，而是考查应聘者对计算机底层机制的理解和设计程序的原则。也就是说，如果让你设计编译器，你将怎样解决内存对齐的问题。

答案：

4, 11, 100, 400, 4, 3, 4, 6, 8, 4, 4。

扩展知识（内存中的数据对齐）

数据对齐，是指数据所在的内存地址必须是该数据长度的整数倍。DWORD 数据的内存起始地址能被 4 除尽，WORD 数据的内存起始地址能被 2 除尽。x86 CPU 能直接访问对齐的数据，当它试图访问一个未对齐的数据时，会在内部进行一系列的调整。这些调整对于程序来说是透明的，但是会降低运行速度，所以编译器在编译程序时会尽量保证数据对齐。同样一段代码，我们来看看用 VC、Dev C++ 和 LCC 这 3 个不同的编译器编译出来的程序的执行结果：

```
#include <stdio.h>
int main()
{
    int a;
    char b;
```

```
int c;
printf("0x%08x ",&a);
printf("0x%08x ",&b);
printf("0x%08x ",&c);
return 0;
}
```

这是用 VC 编译后的执行结果：

```
0x0012ff7c
0x0012ff7b
0x0012ff80
```

变量在内存中的顺序：b（1字节）—a（4字节）—c（4字节）。

这是用 Dev C++ 编译后的执行结果：

```
0x0022ff7c
0x0022ff7b
0x0022ff74
```

变量在内存中的顺序：c（4字节）—中间相隔 3 字节—b（占 1 字节）—a（4 字节）。

这是用 LCC 编译后的执行结果：

```
0x0012ff6c
0x0012ff6b
0x0012ff64
```

变量在内存中的顺序：同上。

3个编译器都做到了数据对齐，但是后两个编译器显然没VC“聪明”，让一个char占了4字节，浪费内存。

面试例题2：以下代码为32位机器编译，数据是以4字节为对齐单位，这两个类的输出结果为什么不同？[中国著名软件企业JS公司2008年4月面试题]

```
class B
{
private:
    bool m_bTemp;
    int m_nTemp;
    bool m_bTemp2;
};

class C
{
private:
    int m_nTemp;
    bool m_bTemp;
    bool m_bTemp2;
};

cout << sizeof(B) << endl;
cout << sizeof(C) << endl;
```

解析：在访问内存时，如果地址按4字节对齐，则访问效率会高很多。这种现象的原因在于访问内存的硬件电路。一般情况下，地址总线总是按照对齐后的地址来访问的。例如你想得到0x00000001开始的4字节内容，系统首先需要以0x00000000读4字节，从中取得3字节，然后再用0x00000004作为开始地址，获得下一个4字节，再从中得到第一个字节，两次组合出你想得到的内容。但是如果地址一开始就是对齐到0x00000000，则系统只要一次读写即可。

考虑到性能方面，编译器会对结构进行对齐处理。考虑下面的结构：

```
struct aStruct
...
    char cValue;
    int iValue;
```

直观地讲，这个结构的尺寸是`sizeof(char)+sizeof(int)=6`，但是在实际编译下，这个结构尺寸默认是8，因为第二个域iValue会被对齐到第4个字节。

在VC中，我们可以用`pack`预处理指令来禁止对齐调整。例如，下面的代码将使得结构尺寸更加紧凑，不会出现对齐到4字节问题：

```
#pragma pack(1)
struct aStruct...
    char cValue;
    int iValue;
```

对于这个`pack`指令的含义，大家可以查询MSDN。请注意，除非你觉得必须，否则不要轻易做这样的调整，因为这将降低程序的性能。目前比较常见的用法有两种，一是这个结

构需要被直接写入文件；二是这个结构需要通过网络传给其他程序。

注意：字节对齐是编译时决定的，一旦决定则不会再改变，因此即使有对齐的因素在，也不会出现一个结构在运行时尺寸发生变化的情况。

在本题中，第一种类的数据对齐是下面的情况：

```
| bool| ----| ----| ----|
| -----int-----|
| bool| ----| ----| ----|
```

第二种类的数据对齐是下面的情况：

```
| -----int-----|
| bool| bool| ----| ----|
```

所以类的大小分别是 3×4 和 2×4 。

答案：B类输出12字节，C类输出8字节。

面试例题3：求解下面程序的结果。[中国著名通信企业H公司面试题]

```
#include <iostream>
using namespace std;

class A1
{
public:
    int a;
    static int b;

    A1();
    ~A1();
};

class A2
{
public:
    int a;
    char c;
    A2();
    ~A2();
};

class A3
{
public:
    float a;
    char c;
    A3();
    ~A3();
};

class A4
{
public:
    float a;
    int b;
    char c;
    A4();
    ~A4();
};

class A5
{
public:
    double d;
    float a;
    int b;
    char c;
    A5();
    ~A5();
};

int main() {
    cout<<sizeof(A1)<<endl;
    cout<<sizeof(A2)<<endl;
    cout<<sizeof(A3)<<endl;
    cout<<sizeof(A4)<<endl;
    cout<<sizeof(A5)<<endl;
}
```

```
return 0; }
```

解析：因为静态变量是存放在全局数据区的，而 sizeof 计算栈中分配的大小，是不会计算在内的，所以 sizeof(A1) 是 4。

- 为了照顾数据对齐，int 大小为 4，char 大小为 1，所以 sizeof(A2) 是 8。
- 为了照顾数据对齐，float 大小为 4，char 大小为 1，所以 sizeof(A3) 是 8。
- 为了照顾数据对齐，float 大小为 4，int 大小为 4，char 大小为 1，所以 sizeof(A4) 是 12。
- 为了照顾数据对齐，double 大小为 8，float 大小为 4，int 大小为 4，char 大小为 1，所以 sizeof(A5) 是 24。

答案：4, 8, 8, 12, 24。

面试例题 4：说明 sizeof 和 strlen 之间的区别。

解析：

由以下几个例子我们说明 sizeof 和 strlen 之间的区别。

第 1 个例子：

```
char* ss = "0123456789";
```

sizeof(ss) 结果为 4，ss 是指向字符串常量的字符指针。

sizeof(*ss) 结果为 1，*ss 是第一个字符。

第 2 个例子：

```
char ss[] = "0123456789";
```

sizeof(ss) 结果为 11，ss 是数组，计算到 “\0” 位置，因此是 $(10 + 1)$ 。

sizeof(*ss) 结果为 1，*ss 是第一个字符。

第 3 个例子：

```
char ss[100] = "0123456789";
```

sizeof(ss) 结果为 100，ss 表示在内存中预分配的大小， 100×1 。

strlen(ss) 结果为 10，它的内部实现是用一个循环计算字符串的长度，直到 “\0” 为止。

第 4 个例子：

```
int ss[100] = "0123456789";
```

sizeof(ss) 结果为 400，ss 表示在内存中的大小， 100×4 。

strlen(ss) 错误，strlen 的参数只能是 char*，且必须是以 “\0” 结尾的。

第 5 个例子：

```
class X {
```

```
int i;
int j;
char k;
```

cout<<sizeof(X)<<endl; 结果为 12，内存补齐。

cout<<sizeof(x)<<endl; 结果为 12，理由同上。

答案：

通过对 sizeof 与 strlen 的深入理解，得出两者区别如下：

(1) sizeof 操作符的结果类型是 size_t，它在头文件中的 typedef 为 unsigned int 类型。该类型保证能容纳实现所建立的最大对象的字节大小。

(2) sizeof 是运算符，strlen 是函数。

(3) sizeof 可以用类型做参数，strlen 只能用 char* 做参数，且必须是以“\0”结尾的。sizeof 还可以用函数做参数，比如：

```
short f();
printf("%d\n", sizeof(f()));
```

输出的结果是 sizeof(short)，即 2。

(4) 数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针。

(5) 大部分编译程序在编译的时候就把 sizeof 计算过了，是类型或是变量的长度。这就是 sizeof(x) 可以用来定义数组维数的原因：

```
char str[20] = "0123456789";
int a = strlen(str); // a=10;
int b = sizeof(str); // 而 b=20;
```

(6) strlen 的结果要在运行的时候才能计算出来，用来计算字符串的长度，而不是类型占内存的大小。

(7) sizeof 后如果是类型必须加括号，如果是变量名可以不加括号。这是因为 sizeof 是个操作符而不是个函数。

(8) 当使用了一个结构类型或变量时，sizeof 返回实际的大小。当使用一静态的空间数组时，sizeof 返回全部数组的尺寸。sizeof 操作符不能返回被动态分配的数组或外部的数组的尺寸。

(9) 数组作为参数传给函数时传递的是指针而不是数组，传递的是数组的首地址，如 fun(char [8])、fun(char []) 都等价于 fun(char *)。在 C++ 里传递数组永远都是传递指向数组首元素的指针，编译器不知道数组的大小。如果想在函数内知道数组的大小，需要这样做：进入函数后用 memcpy 将数组复制出来，长度由另一个形参传进去。代码如下：

```
fun (unsigned char *pl, int len)
{
```

```

unsigned char* buf = new unsigned
    char[len+1]
memcpy(buf, p1, len);
}

```

(10) 计算结构变量的大小就必须讨论数据对齐问题。为了使 CPU 存取的速度最快（这同 CPU 取数操作有关，详细的介绍可以参考一些计算机原理方面的书），C++在处理数据时经常把结构变量中的成员的大小按照 4 或 8 的倍数计算，这就叫数据对齐（data alignment）。这样做可能会浪费一些内存，但在理论上 CPU 速度快了。当然，这样的设置会在读写一些别的应用程序生成的数据文件或交换数据时带来不便。MS VC++中的对齐设定，有时候 sizeof 得到的与实际不等。一般在 VC++中加上#pragma pack(n)的设定即可。或者如果要按字节存储，而不进行数据对齐，可以在 Options 对话框中修改 Advanced Compiler 选项卡中的“Data Alignment”为按字节对齐。

(11) sizeof 操作符不能用于函数类型、不完全类型或位字段。不完全类型指具有未知存储大小数据的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、void 类型等。

面试例题 5：说明 sizeof 的使用场合。

答案：

(1) sizeof 操作符的一个主要用途是与存储分配和 I/O 系统那样的例程进行通信。例如：

```

void *malloc (size_t size),
size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)

```

(2) 用它可以看看某种类型的对象在内存中所占的单元字节。例如：

```

void * memset (void * s, int c, sizeof(s))

```

(3) 在动态分配一对象时，可以让系统知道要分配多少内存。

(4) 便于一些类型的扩充。在 Windows 中有很多结构类型就有一个专用的字段用来存放该类型的字节大小。

(5) 由于操作数的字节数在实现时可能出现变化，建议在涉及操作数字节大小时用 sizeof 替代常量计算。

(6) 如果操作数是函数中的数组形参或函数类型的形参，sizeof 给出其指针的大小。

面试例题 6：How many bytes will be occupied for the variable (definition: int **a[3][4])? (这个数组占据多大空间？) [中国某著名计算机金融软件公司 2005 年面试题]

- A. 64 B. 12 C. 48 D. 128

解析： sizeof 问题， $3 \times 4 \times 4 = 48$ 。

答案：C

面试例题 7：Find the defects in each of the following programs, and explain why it is incorrect. (找出下面程序的错误，并解释它为什么是错的。) [中国台湾某著名杀毒软件公司 2005 年、2007 年面试题]

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
    //To output "TrendMicroSoftUSCN"

    string strArr1[]=
        {"Trend", "Micro", "Soft"};
    string *pStrArr1=new string[2];
    pStrArr1[0]="US";
}
```

```
pStrArr1[1] = "CN";

for(int i=0; i<sizeof(strArr1)
    /sizeof(string); i++)
    cout<<strArr1[i];
for(int j=0; j<sizeof(pStrArr1)
    /sizeof(string); j++)
    cout<<pStrArr1[j];
}

return 0;
```

解析： sizeof 问题。本题在 2005、2007 两年的面试题中都出现过。

程序运行后输出：TrendMicroSoftUS。这是因为 sizeof(pStrArr1) 运算得出的结果是指针 pStrArr1 的大小，即 4。这样就不能正确地输出“USCN”。而字符串 strArr1 是由 3 段构成的，所以 sizeof(strArr1) 大小是 12。

首先要明确 sizeof 不是函数，也不是一元运算符，它是个类似宏定义的特殊关键字，sizeof ()。括号内的内容在编译过程中是不被编译的，而是被替代类型，如 int a=8; sizeof (a)。在编译过程中，不管 a 的值是什么，只是被替换成类型 sizeof (int)，结果为 4。如果 sizeof (a=6) 呢？也是一样地转换成 a 的类型。但是要注意，因为 a=6 是不被编译的，所以执行完 sizeof (a=6) 后，a 的值还是 8，是不变的。

请记住以下几个结论：

(1) unsigned 影响的只是最高位 bit 的意义（正/负），数据长度是不会被改变的，所以：

```
sizeof(unsigned int) == sizeof(int)
```

(2) 自定义类型的 sizeof 取值等同于它的类型原形。如：

```
typedef short WORD; sizeof(short) == sizeof(WORD)
```

(3) 对函数使用 sizeof，在编译阶段会被函数返回值的类型取代。如：int f1() {return 0;}。

```
cout << sizeof(f1()) << endl; // f1() 返回值为 int，因此被认为是 int
```

(4) 只要是指针，大小就是 4。如：

```
cout << sizeof(string*) << endl; // 4
```

(5) 数组的大小是各维数的乘积 × 数组元素的大小。如：

```
char a[] = "abcdef ";
int b[20] = {3, 4};
char c[2][3] = { "aa ", "bb "};
// 7, 注意还有空格及 '\0'
```

```
cout << sizeof(a) << endl;
cout << sizeof(b) << endl; // 20×4
cout << sizeof(c) << endl; // 6
```

数组 a 的大小在定义时未指定，编译时给它分配的空间是按照初始化的值确定的，也就是 7，包括 ‘\0’。

所以上面的问题就解决得差不多了，`sizeof(string)=4`。

有的面试者说正确答案是将：

```
for(i=0;i < sizeof(p)/sizeof(string);i++)
```

改为：

```
for(i=0;i < sizeof(p)*2/sizeof(string);i++)
```

这样修改的结果能得到正确的输出结果，但却不是正确答案。`sizeof (p))` 只是指针大小为 4，要想求出数组 p 指向数组的成员个数，应该为 `sizeof(*p)*2/sizeof(string)`。这是因为指针 p 指向数组，则*p 就是指向数组中的成员了，成员的类型是 string 型，那么 `sizeof(*p)` 为 4，乘以 2 才是整个数组的大小。`sizeof (p)` 的大小只是碰巧与 `sizeof (*p)` 大小一致罢了。

答案：

正确的程序如下：

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc,char *argv[])
{
    string strArr1[] = {"Trend",
                        "Micro","Soft"};
    string *pStrArr1 = new string[2];
    pStrArr1[0] = "US";
    pStrArr1[1] = "CN";
    cout << sizeof(strArr1) << endl;
```

```
cout << sizeof(string) << endl;
for(int i =0; i < sizeof(strArr1)/
    sizeof(string);i++)
    cout << strArr1[i];
    cout << endl;
for(int j =0; j< sizeof(*pStrArr1)*2/
    sizeof(string);j++)
    cout << pStrArr1[j];

return 0;
}
```

面试例题 8：写出下面 sizeof 的答案。[德国某著名软件咨询企业 2005 年面试题]

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
    Base() { cout<<"Base-ctor"<<endl; }
    ~Base() { cout<<"Base-dtor"<<endl; }
    virtual void f(int) { cout<<"Base::
        f(int)"<<endl; }
    virtual void f(double) { cout<<"Base::
        f(double)"<<endl; }
```

```
virtual void g(int i = 10) {cout<<"Base::
    g()"<<i<<endl; }
void g2(int i = 10) {cout<<"Base::
    g2()"<<i<<endl; }
};

class Derived: public Base
{
public:
    Derived() { cout<<"Derived-ctor"
        <<endl; }
    ~Derived() { cout<<"Derived-dtor"
```

```

        <<endl;
    void f(complex<double>) { cout
        <<"Derived::f(complex)"<<endl;
    virtual void g(int i = 20) {cout
        <<"Derived::g()"<<i<<endl;
    }

int main()
{
    Base b;
    Derived d;
}

```

```

Base* pb = new Derived;
//从下面的4个答案中选出1个正确的答案
cout<<sizeof(Base)<<"tt"<<endl;
//A.4  B.32  C.20
//D Platform-dependent
cout<<sizeof(Derived)<<"bb"<<endl;
//A.4  B.8  C.36  D.Platform-dependent
}

```

解析：求类 base 的大小。因为类 base 只有一个指针，所以类 base 的大小是 4。Derive 大小与 base 类似，所以也是 4。

答案：A, A。

面试例题 9：以下代码的输出结果是多少？[中国某著名计算机金融软件公司 2006 年面试题]

```

char var[10];
int test(char var[])
{

```

```

    return sizeof(var)
};


```

- A. 10 B. 9 C. 11 D. 4

解析：因为 var[] 等价于 *var，已经退化成一个指针了，所以大小是 4。

答案：D。

面试例题 10：以下代码的输出结果是多少？[美国某著名防毒软件公司 2006 年面试题]

```

class B
{
    float f;
    char p;
}

```

```

int adf[3];
};

cout << ""<< sizeof(B);

```

解析：float f 占 4 个字节，char p 占 1 字节，int adf[3] 占 12 字节，总共是 17 字节。根据内存对齐原则，要选择 4 的倍数，是 20 字节。

答案：20

面试例题 11：一个空类占多少空间？多重继承的空类呢？[英国某著名计算机图形图像公司面试题]

解析：我们用程序来实现一个空类和一个多重继承的空类。看看它们的大小是多少。代码如下：

```

#include <iostream>
#include <memory.h>
#include <assert.h>

using namespace std;
class A
{
}

```

```

};

class A2
{
};

class B : public A

```

```

    {
    class A
    {
    };
    class C : public virtual B
    {
    };
    class D : public A,public A2
    {
    };
    int main(int argc,char *argv[])
    {

```

```

        cout << "sizeof(A) : " << sizeof(A)
            << endl;
        cout << "sizeof(B) : " << sizeof(B)
            << endl;
        cout << "sizeof(C) : " << sizeof(C)
            << endl;
        cout << "sizeof(D) : " << sizeof(D)
            << endl;
        return 0;
    }
}

```

以上答案分别是：1，1，4，1。这说明空类所占空间为1，单一继承的空类空间也为1，多重继承的空类空间还是1。但是虚继承涉及虚表（虚指针），所以 sizeof(C)的大小为4。

答案：一个空类所占空间为1，多重继承的空类所占空间还是1。

6.4 内联函数和宏定义

面试例题：内联函数和宏的差别是什么？

答案：内联函数和普通函数相比可以加快程序运行的速度，因为不需要中断调用，在编译的时候内联函数可以直接被镶嵌到目标代码中。而宏只是一个简单的替换。

内联函数要做参数类型检查，这是内联函数跟宏相比的优势。

inline 是指嵌入代码，就是在调用函数的地方不是跳转，而是把代码直接写到那里去。对于短小的代码来说 inline 增加空间消耗换来的是效率提高，这方面和宏是一模一样的，但是 inline 在和宏相比没有付出任何额外代价的情况下更安全。至于是否需要 inline 函数，就需要根据实际情况来取舍了。

inline 一般只用于如下情况：

- (1) 一个函数不断被重复调用。
- (2) 函数只有简单的几行，且函数内不包含 for、while、switch 语句。

一般来说，我们写小程序没有必要定义成 inline，但是如果要完成一个工程项目，当一个简单函数被调用多次时，则应该考虑用 inline。

宏在 C 语言里极其重要，而在 C++里用得就少多了。关于宏的第一规则是绝不应该去使用它，除非你不得不这样做。几乎每个宏都表明了程序设计语言里、程序里或者程序员的一个缺陷，因为它将在编译器看到程序的正文之前重新摆布这些正文。宏也是许多程序设计工具的主要麻烦。所以，如果你使用了宏，就应该准备只能从各种工具（如排错系统、交叉引用系统、轮廓程序等）中得到较少的服务。

宏是在代码处不加任何验证的简单替代，而内联函数是将代码直接插入调用处，而减少了普通函数调用时的资源消耗。

宏不是函数，只是在编译前（编译预处理阶段）将程序中有关字符串替换成宏体。

关键字 `inline` 必须与函数定义体放在一起才能使函数成为内联，仅将 `inline` 放在函数声明前面不起任何作用。如下风格的函数 `Foo` 不能成为内联函数：

```
inline void Foo(int x, int y); // inline 仅与函数声明放在一起
void Foo(int x, int y){}
```

而如下风格的函数 `Foo` 则成为内联函数：

```
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
```

所以说，`inline` 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。内联能提高函数的执行效率，至于为什么不把所有的函数都定义成内联函数？如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

以下情况不宜使用内联：1 如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。2 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 `inline` 不应该出现在函数的声明中）。

第 7 章

指针与引用

指针是 C 系语言的特色。指针是 C++ 提供的一种颇具特色的数据类型，允许直接获取和操纵数据地址，实现动态存储分配。

指针是 C 和 C++ 的精华所在，也是 C 和 C++ 中一个十分重要的概念。一个数据对象的内存地址称为该数据对象的指针。指针可以表示各种数据对象，如简单变量、数组、数组元素、结构体，甚至函数。换句话说，指针具有不同的类型，可以指向不同的数据存储体。

指针问题，包括常量指针、数组指针、函数指针、this 指针、指针传值、指向指针的指针等，这些问题也是各大公司的常备考点。本章不对指针基本知识做回顾和分析（请参考 C++ 其他经典著作），而是通过对各公司面试题目进行全面、仔细的解析帮助读者解决其中的难点。

7.1 指针基本问题

面试例题 1：指针和引用的差别？

答案：

(1) 非空区别。在任何情况下都不能使用指向空值的引用。一个引用必须总是指向某些对象。因此如果你使用一个变量并让它指向一个对象，但是该变量在某些时候也可能不指向任何对象，这时你应该把变量声明为指针，因为这样你可以赋空值给该变量。相反，如果变量肯定指向一个对象，例如你的设计不允许变量为空，这时你就可以把变量声明为引用。不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针要高。

(2) 合法性区别。在使用引用之前不需要测试它的合法性。相反，指针则应该总是被测试，防止其为空。

(3) 可修改区别。指针与引用的另一个重要的区别是指针可以被重新赋值以指向另一个不同的对象。但是引用则总是指向在初始化时被指定的对象，以后不能改变，但是指定的对象其内容可以改变。

(4) 应用区别。总的来说，在以下情况下应该使用指针：一是考虑到存在不指向任何对象的可能（在这种情况下，能够设置指针为空），二是需要能够在不同的时刻指向不同的对象（在这种情况下，你能改变指针的指向）。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向，那么应该使用引用。

面试例题2：Please check out which of the following statements are wrong? (看下面的程序哪里有错?) [中国台湾某著名计算机硬件公司 2005 年 12 月面试题]

```
#include <iostream>
using namespace std;
int main()
{
    int iv; //1
    int iv2=1024; //2
    int iv3=999; //3
    int &reiv; //4
    int &reiv2 = iv; //5
    int &reiv3 = iv; //6
    int *pi; //7
    *pi = 5; //8
    pi=&iv3; //9
    const double di; //10
    const double maxWage =10.0; //11
    const double minWage =0.5;
    const double *pc =&maxWage; //12
    cout << pi;
    return 0;
}
```

答案：

- 1 正确，很正常地声明了一个整型变量。
- 2 正确，很正常地声明了一个整型变量，同时初始化这个变量。
- 3 正确，理由同上。
- 4 错误，声明了一个引用，但引用不能为空，必须同时初始化。
- 5 正确，声明了一个引用 reiv2，同时初始化了，也就是 reiv2 是 iv 的别名。
- 6 正确，理由同上。
- 7 正确，声明了一个整数指针，但是并没有定义这个指针所指向的地址。
- 8 错误，整数指针 pi 并没有指向实际的地址。在这种情况下就给它赋值是错误的，因为赋的值不知道该放到哪里去，从而造成错误。
- 9 正确，整数指针 pi 指向 iv3 实际的地址。
- 10 错误，const 常量赋值时，必须同时初始化。
- 11 正确，const 常量赋值并同时初始化。
- 12 正确，const 常量指针赋值并同时初始化。

7.2 传递动态内存

面试例题 1：下面 5 个函数哪个能够成功进行两个数的交换？[中国某互联网公司 2009 年 12 月笔试题]

```
#include <iostream>
using namespace std;

void swap1(int p, int q)
{
    int temp;
    temp=p;
    p=q;
    q=temp;
}

void swap2(int *p, int *q)
{
    int *temp;
    *temp=*p;
    *p=*q;
    *q=*temp;
}

void swap3(int *p, int *q)
{
    int *temp;
    temp=p;
    p=q;
    q=temp;
}
```

```
void swap4(int *p, int *q)
{
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}

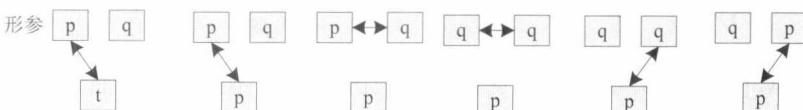
void swap5(int &p, int &q)
{
    int temp;
    temp=p;
    p=q;
    q=temp;
}

int main()
{
    int a=1,b=2;
    //swap1(a,b);
    //swap2(&a,&b);
    //swap3(&a,&b);
    //swap4(&a,&b);
    //swap5(a,b);
    cout << a << " " << b << endl;
    return 0;
}
```

解析：这道题考察函数参数传递、值传递、指针传递（地址传递）、引用传递。

swap1 传的是值的副本，在函数体内被修改了形参 p、q（实际参数 a、b 的一个复制），p、q 的值确实交换了，但是它们是局部变量，不会影响到主函数中的 a 和 b。当函数 swap1 生命周期结束时，p、q 所在栈也就被删除了，如下图所示。

实参 a b



swap2 传的是一个地址进去，在函数体内的形参*p、*q 是指向实际参数 a、b 地址的两个