C++的 64 位整数[转]+gyy 整理

（1）

在做 ACM 题时，经常都会遇到一些比较大的整数。而常用的内置整数类型常常显得太小了：

long 和 int 范围是[-2^31,2^31)，-2147483648~2147483647。而 unsigned 范围是[0,2^32)，即 0~4294967295。

也就是说，常规的 32 位整数只能够处理 40 亿以下的数。

那遇到比 40 亿要大的数怎么办呢？这时就要用到 C++的 64 位扩展了。不同的编译器对 64 位整数的扩展有所不同。

基于 ACM 的需要，下面仅介绍 VC6.0 与 g++编译器的扩展。

VC6.0 的 64 位整数分别叫做__int64 与 unsigned __int64 (注意都是两个下划线)，其范围分别是[-2^63, 2^63)与[0,2^64)，

即-9223372036854775808~9223372036854775807

与 0~18446744073709551615(约 1800 亿亿)。

对 64 位整数的运算与 32 位整数基本相同，都支持四则运算与位运算等。当进行 64 位与 32 位的混合运算时，32 位整数会被隐式转换成 64 位整数。

```
#include <iostream>
using namespace std;
int main( )
{   cout<<"__int64 整型类型数据在内存中占"
          <<sizeof(__int64)<<"个字节"<<endl;
    cout<<"unsigned __int64 整型类型数据在内存中占"
          <<sizeof(unsigned __int64)<<"个字节"<<endl;
    return 0;
}
```

```
_int64 整型类型数据在内存中占8个字节
unsigned _int64 整型类型数据在内存中占8个字节
```

但是，VC 的输入输出与__int64 的兼容就不是很好，如果你写下这样一段代码：

```
#include <iostream>
#include <cstdio>
using namespace std;
int main( )
{
  __int64 a;
  cin >> a;
  cout << a;
  return 0;
}
```

编译时，会抱错。

在函数体第 2 行会收到“error C2679: binary '>>' : no operator defined which takes a right-hand operand of type '__int64' (or there is no acceptable conversion)”的错误；

在第 3 行会收到“error C2593: 'operator <<' is ambiguous”的错误。

那是不是就不能进行输入输出呢？当然不是，你可以使用 C 的写法：

```
#include <iostream>
#include <cstdio>
using namespace std;
int main( )
{    __int64 a;
     scanf("%I64d",&a);
    printf("%I64d",a);
     return 0;
}
```

编译无错，就可以正确输入输出了。

当使用 unsigned __int64 时，把"I64d"改为"I64u"就可以了。

OJ 通常使用 g++编译器。其 64 位扩展方式与 VC 有所不同，它们分别叫做 long long 与 unsigned long long。处理规模与除输入输出外的使用方法同上。对于输入输出，它的扩展比 VC 好。既可以使用

```
long long a;
cin>>a;
cout<<a;
```

也可以使用

```
scanf("%lld",&a);
printf("%lld",a);
```

使用无符号数时，将"%lld"改成"%llu"即可。

最后我补充一点：作为一个特例，如果你使用的是 Dev-C++的 g++编译器，它使用的是"%I64d"而非"%lld"。

（2）

C/C++的 64 位整型

在 C/C++中，64 为整型一直是一种没有确定规范的数据类型。现今主流的编译器中，对 64 为整型的支持也是标准不一，形态各异。

一般来说，64 位整型的定义方式有 long long 和__int64 两种(VC 还支持_int64)，而输出到标准输出方式有 printf("%lld",a)，printf("%I64d",a)，和 cout << a 三种方式。

本文讨论的是五种常用的 C/C++编译器对 64 位整型的支持，这五种编译器分别是 gcc(mingw32)，g++(mingw32)，gcc(linux i386)，g++(linux i386)，Microsoft Visual C++ 6.0。可惜的是，没有一种定义和输出方式组合，同时兼容这五种编译器。为彻底弄清不同编译器对 64 位整型，写了程序对它们进行了评测，结果如下表。

变量定义	输出方式	gcc(mingw32)	g++(mingw32)	gcc(linux i386)	g++(linux i386)	MicrosoftVisual C++ 6.0
long long	"%lld"	错误	错误	正确	正确	无法编译
long long	"%I64d"	正确	正确	错误	错误	无法编译
int64	"lld"	错误	错误	无法编译	无法编译	错误
int64	"%I64d"	正确	正确	无法编译	无法编译	正确
long long	cout	非 C++	正确	非 C++	正确	无法编译
__int64	cout	非 C++	正确	非 C++	无法编译	无法编译
long long	printint64()	正确	正确	正确	正确	无法编译

上表中，正确指编译通过，运行完全正确；错误指编译虽然通过，但运行结果有误；无法编译指编译器根本不能编译完成。观察上表，我们可以发现以下几点：

1. long long 定义方式可以用于 gcc/g++，不受平台限制，但不能用于 VC6.0。
2. __int64 是 Win32 平台编译器 64 位长整型的定义方式，不能用于 Linux。
3. "%lld"用于 Linux i386 平台编译器，"%I64d"用于 Win32 平台编译器。
4. cout 只能用于 C++编译，在 VC6.0 中，cout 不支持 64 位长整型。

表中最后一行输出方式中的 printint64(),可以看出，它的兼容性要好于其他所有的输出方式，它是一段这样的代码：

```
void  printint64(long  long  a)
{
      if  (a<=100000000)
            printf("%d\n",a);
```

```
    else
    {
        printf("%d",a/100000000);
        printf("%08d\n",a%100000000);
    }
}
```

这种写法的本质是把较大的 64 位整型拆分为两个 32 位整型，然后依次输出，低位的部分要补 0。

(3)

64 位整数全解(增补板)

64 位整形引起的混乱主要在两方面，一是数据类型的声明，二是输入输出。

首先是如果我们在自己机器上写程序的话，情况分类如下：

(1) 在 win 下的 VC6.0 里面，声明数据类型的时候应该写作

```
__int64 a;
```

输入输出的时候用 %I64d

```
scanf(" %I64d" ,&a);
printf(" %I64d" ,a);
```

(2) 在 linux 下的 gcc/g++里面，数据类型声明写作

```
long long a;
```

输入输出时候用 %lld

(3) 在 win 下的其它 IDE 里面[包括高版本 Visual Studio]，数据类型声明用上面两种均可

输入输出用 %I64d

以下是对这种混乱情况的解释，如无兴趣可以跳过

首先要说的是，和 Java 等语言不同，C/C++本身并没有规定各数据类型的位数，只是限定了一个大小关系，也就是规定从所占的 bit 数来说，short <= int <= long <= long long。至于具体哪种类型占用多少位，是由你所用的开发平台的

编译器决定的。在现在的 PC 上一个通常的标准是，int 和 long 同为 32 位，long long 为 64 位。但是如果换到其它平台(如 ARM)上，这个数字可能会有不同，类型所占的大小可以用 sizeof()运算符查看。

long long 是 C99 标准中新引进的数据类型,在古老的 VC6.0 中并没有这个类型，所以在 VC6.0 中用”long long”会发生编译错误。为了表示 64 位整数，VC6 里采用的是微软自己搞出来的一个数据类型,叫做__int64,所以如果你是在 VC6.0 下编译的话，应该用__int64 定义 64 位整型。新版的 Visual Studio 已经支持 long long 了。GCC 是支持 long long 的，我们在 win 系统中使用的其它 IDE 如 Dev-Cpp, Code::Blocks 等等大多是采用的 MinGW 编译环境,它是与 GCC 兼容的，所以也支持 long long（另外为了与 MS 兼容，也支持__int64）。如果是在纯的 linux 下，就只能使用 long long 了。

关于使用 printf 的输入输出，这里就有一个更囧的情况。实际上只要记住，主要的区分在于操作系统:如果在 win 系统下,那么无论什么编译器,一律用%I64d;如果在 linux 系统，一律用%lld。这是因为 MS 提供的 msvcrt.dll 库里使用的就是%I64d 的方式，尽管 Dev-Cpp 等在语法上支持标准，但也不得不使用 MS 提供的 dll 库来完成 IO，所以就造成了这种情况。

那么对 ACMer 来说，最为关心的就是在各个 OJ 上交题应分别使用哪种方式了。其实方式只有有限的几种：

如果服务器是 linux 系统，那么定义用 long long，IO 用%lld

如果服务器是 win 系统，那么声明要针对编译器而定：

+ 如果用 MS 系列编译器，声明用__int64 [现在新版的 Visual Studio 也支持 long long 了]
+ 如果用 MinGW 环境，声明用 long long
+ 无论什么编译器，IO 一律%I64d

下面把各大 OJ 情况列表如下：

(1) TOJ : Linux 系统

(2) ZOJ : Linux 系统

(3) POJ : Win 系统，语言如选择 C/C++，则用 MS 编译器[支持两种声明]，如选择 GCC/G++，则为 MinGW

(4) UVa ： Linux 系统

(5) Ural: Win 系统，MS 编译器[支持两种声明]

(6) SPOJ: Linux 系统

(7) SGU ： Win 系统，MS 编译器[支持两种声明]

如果有不太清楚的情况可以先看看各 OJ 上的 FAQ，通常会有说明。

另外，为了避免混乱，当数据量不大时，用 cin, cout 进行输入输出也是一种选择