Integer Data Types

Representation

Integers are represented in computers as a sequence of bits in 2's complement form. There are various built in data types provided with the C/C++ compilers which has different storage capability. The higher the storage capability of a type, the larger the sequence of bits it can store and higher is the range of integers it can support.

Ranges of standard data types

Data Type	Number of Bits	Range
Char	8	-2^7 to 2^7-1
Unsigned Char	8	0 to 2^8-1
Short	16	-2^{15} to $2^{15}-1$
Unsigned Short	16	0 to $2^{16}-1$
Int	32	-2^{31} to $2^{31}-1$
Unsigned Int	32	0 to $2^{32}-1$
Long	32	-2^{31} to $2^{31}-1$
Unsigned Long	32	0 to $2^{32}-1$

Value Overflow

What will happen if you try to store a value in a variable which is too big or too small for it to contain?

Run the following code and you will get your answer:

```
int i = 120;
printf("Expected value: 120\tThe value we get: %d\n", i);
i += 60;
printf("Expected value: 180\tThe value we get: %d\n", i);
i = -120;
printf("Expected value: -120\tThe value we get: %d\n", i);
i -= 60;
printf("Expected value: -180\tThe value we get: %d\n", i);
```

So never think that an addition operation will always give you a greater value or a subtraction operation will always give you a lesser value.

* *What actually happens in this case is the addition/subtraction operation produces carry bits which the variable is not capable of storing. Hence it is ignored and the value we are left with is only last 8 bits of the expected value. Remember that the value is considered in 2's complement form and if the value has 1 as its MSB it is a negative number.*

Use of sizeof()

In different compilers the size of the same data type can be different. For example in some compilers such as Turbo C/C++ integer data type is 16 bits and in some others it is 32 bits. In GNU C/C++ compiler which is used by UVA integer data type is 32 bits.

To know about the size of a data type in the t compiler in which your program is running one can use sizeof().

```
sizeof <expression>
Returns the size (in bytes) of the given expression

sizeof <type>
Returns the size (in bytes) of the given data type

Example:

   1. sizeof(long)
   2. long array[100];
      sizeof(array) will return (100*4)

   * Thus sizeof() can be used in case of memset(), memcpy()
     functions as follows:
         memset(array, 0, sizeof(array))
```

Use of memset()

Sometimes we need to set the values of all the elements of an array. For this purpose memset() is a very good function. For example to set all the values of an array to zero we can always use:

```
memset(arrayName, 0, sizeof(arrayName))
```

This is a much faster way than setting all the values by running a loop. (as the function directly works on a block of memory rather than an array element)

But be careful about the second parameter of the array. It actually says what to put in each BYTE of the array. (not in each element). For example if arrayName is an array of integers then using memset() with the value as 1 will not set the all the values to 1. Rather it will set all the bytes to 1. So each integer will actually contain

```
0000 0001 0000 0001 0000 0001 0000 0001 = 16843009
```

Memory Limit

We will limit our discussion to UVA site problems. In UVA the default memory limit for problems are 32 MB (unless otherwise stated in the problem description). So what will be the size of biggest integer array that we can use. We already know that the compiler used in UVA is GNU C/C++ where the size of integer is 32 bits or 4 bytes.

```
32MB = 32,000,000 Bytes = 4 * 8,000,000
```

Hence we can say that the biggest integer array possible is 8,000,000. But we have to keep some for other variables and use of the program.

Bitwise operation

The lowest data type available to us is bool / char which is 8 bits or 1 byte in length. But in many occasions we need to only store TURE / FALSE or 1 / 0 in a field. In these cases we can use bitwise operations to save memory and also to make solutions faster. But as we have 8 bits as the lowest available memory unit we will have to access individual bits by using manual bitwise operatives such as AND (&), OR (|), NOT (!).

Say you want to declare an array of size X which will contain only TRUE / FALSE values. (e.g. for the adjacency matrix for representing a graph). But if X is beyond the provided memory limit you can solve the problem by using an array of size X/8 bytes by the use of bitwise operations.

Some basic bitwise operations:

1. Set the k^{th} bit in variable 'a':
   `a |=` 2^k
2. Clear the k^{th} bit in variable 'a':
   `a &= (!(`2^k`))`
3. Change the k^{th} bit in variable 'a':
   `a ^=` 2^k

* *There is also a specialized container class in STL* `bit_vector` *which has the same interface as a normal vector container but is optimized for space efficiency. A* `vector` *always requires at least one byte per element, but a* `bit_vector` *only requires one bit per element.*

64-bit integers

General Discussion

In the Turbo C/C++ environment we don't see any option for 64 bit integers. But in MSVC++ and GNU C/C++ you can use 64 bit integers.

Data Type		**Number of Bits**	**Range**
In GNU C/C++	**In MSVC++**		
long long	__int64	64	-2^{63} to $2^{63}-1$
unsigned long long	unsigned __int64	64	0 to $2^{64}-1$

Normally we don't compile our code in GNU C/C++. But as the UVA uses this compiler we have to make our code compatible. The following is a good way of doing it.

```
#ifdef ONLINE_JUDGE
typedef long long i64;
#else
typedef __int64 i64;
#endif

i64 intName;
```

You can easily use "i64" to represent 64-bit integers in the following part of your code.

* *In this case we have used "Preprocessor Directives". The online-judge of UVA defines the ONLINE_JUDGE while compiling your code. You can use it to know whether your code is running in your own machine or in the judge machine.*

Print or scan 64-bit integers:

In GNU C/C++	**In MSVC++**

`long long value;`
 	
`//To Take Input`
`scanf("%lld", &value);`
 	
`//To Print Output`
`printf("%lld", value);`
	`__int64 value;`
	
	`//To Take Input`
	`scanf("%I64d", &value);`
	
	`//To Print Output`
	`printf("%I64d", value);`

`unsigned long long value;`
 	
`//To Take Input`
`scanf("%llu", &value);`
 	
`//To Print Output`
`printf("%llu", value);`
	`unsigned __int64 value;`
	
	`//To Take Input`
	`scanf("%I64u", &value);`
	
	`//To Print Output`
	`printf("%I64u", value);`

Floating Point Errors

Representation

The most common way of representing real numbers are defined by IEEE 754 standard. In this representation the real numbers are defined by three parts.

1. The sign bit (s)
2. The exponent (e)
3. The mantissa (m)

Ignoring some details we can say that real numbers are represented in the format:

$$(-1)^s \times m \times 2^e$$

Single precision (float) and Double precision (double) have the following capacity

Type	Number of bits	sign	exponent	mantissa
float	32	1	8	23
double	64	1	11	52

Special Values

1. *Zero*
   Zero is denoted by a special value where the mantissa and exponent both fields contain all zeroes. But the sign bit may contain one. Hence we have to face the problem with negative zeroes. Moreover as the sign bits differ -0.0 and +0.0 are not the same. If this negative comes in the output then your program may cause a Wrong Answer. One way to get rid of this is to add a very small value (epsilon = 1e-7) to your result.

2. *Infinities*
   An exponent of all ones and a mantissa of all zeroes represent infinity. The sign bit defines whether it is $-\infty$ or $+\infty$

3. *Not a Number*
   NaN represents non-real numbers. For example sqrt(-1.0) will result in such a number. This kind of numbers is represented by exponent of all ones and a non-zero mantissa.

4. *Subnormal Numbers*
   Exponent is all zero and the mantissa is non-zero. This represents a number very close to zero.

Bitwise Comparison of floating point numbers

The bitwise comparison of the floating point numbers will give us the result that we expect comparing the numbers as a whole. Notice that in floating point numbers the sign bit is the MSB. Hence negative and positive numbers can be compared easily by the MSB. Now we can consider the rest as two positive numbers. In the bit sequence next comes the exponent and surely the number with the greater exponent is greater no matter what the mantissa values are. And last of all we are left with comparing the mantissa values when both the exponent and sign bits are same.

Round Off Error and Equality Check

For the double precision floating point numbers we can see that we have $2^{54} > 10^{16}$. Hence *we can exactly represent any integer with 15 decimal digits accurately using the "double" data type*. If the target number is beyond the limit then a floating point number produces a value as close as possible to the target.

It is a very bad practice to check the equality of two floating point numbers using the normal "= ="operator.

```
for(double r=0.0; r != 1.0; r+=0.1)
     printf("*");
```

The above code should print 1.0 / 0.1 = 10 dots. But it actually goes into an infinite loop. The reason is that decimal 0.1 is equivalent to binary 0.0(0011). So it cannot be perfectly represented and the number 1.0 is never actually reached.

One common way to check the equality of two floating point numbers

Incorrect way	Better way

`double fVal;`
 	
`if(fVal == 100.0)`
`{`
`    ……..`
`    ……..`
`}`
	`const double eps = 1e-7;`
	`//Here 'eps' represents a very small value`
	
	`double fVal;`
	
	`if(fabs(fVal-100.0) < eps)`
	`{`
	`    ……..`
	`    ……..`
	`}`

Here we are using a constant ‘eps’ value irrespective the value of the numbers being compared. It is better to use a value dependent on the values that we are comparing.

```
           for (double r=0.0; r<1e22; r+=1.0)
                printf(".");
```

The above code does not use any equal operators and after iterating for 10^{22} times we expect it to terminate as r is increased by an integer value 1. But yet again this will not terminate. 10^{22} is a large number that a double variable cannot store accurately. When you add 1 to 10^{22} then 1 is considered such a small number that the resultant is rounded off to the original number. Hence after a certain value the loop variable ‘r’ never increases and hence the loop is never terminated. The best way of checking equality of two numbers is as follows:

```
bool equals(double a, double b)
{
     if(fabs(a-b) < eps)
          return true;

     double v1 = b*(1-1e-10), v2 = b*(1+1e-10);

     return (a > min(v1, v2)) && (a < max(v1, v2));
}
```

Rounding Off to an integer value

Let we have a double value ‘a’ and an integer ‘b’:

```
     1. b = (int) ceil(a); //b is the nearest integer >= a
     2. b = (int) floor(a); //b is the nearest integer <= a
```

Unfortunately because different problem-setters in UVA use different way of floating point arithmetic you may face some strange Wrong Answers. For example if we want to print the double variable ‘a’ up to 3 decimal spaces sometimes you have to use the following method

```
                a = a + 0.0005 + eps
                printf(“%.3lf”, a);
```

But sometimes only `printf(“%.3lf”, a);` will work fine.

Further Reading

1. Two's Complement : From Wikipedia
2. Secure Coding in C++ : Integers by Robert C. Seacord
3. Bitwise Operators and Bit Masks by Vipan Singla
4. Hacker's Delight by by Henry S. Warren : Basics
5. IEEE Floating Point Standard : From Wikipedia
6. The Aggregate Magic Algorithms
7. Topcoder Tutorial : Integers and Reals : Part 1
8. Topcoder Tutorial : Integers and Reals : Part 2

Created by: ***Quazi Sarfaraz Hossain***