CS1100 - Lecture 15

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

In the last class, we were discussing about number systems and binary representation of decimal numbers. The representation of decimal numbers in binary format inside a computer uses a fixed number of bytes. Depending on the value to be represented there are different variants of integer type available in C. Each of these variants differ in the number of bytes used to store a variable.

From the discussion in previous lecture, recall that signed numbers are represented using 2's complement representation in computers. Further, the largest signed integer that can be represented using 8 bits is 01111111_2 which is equal to $-2^7 \times 0 + 2^6 \times 1 + ... + 2^0 \times 1 = 2^7 - 1 = 127_{10}$. Also, the smallest signed integer that can be represented using 8 bits is 10000000_2 which is equal to $-2^7 \times 1 + 2^6 \times 0 + ... + 2^0 \times 0 = -128_{10}$. In general, if 'k' bits are used to store a signed integer, the largest possible number that can be represented is $2^{k-1} - 1$ and the smallest integer that can be represented is -2^{k-1}.

Some variants of integers available in C corresponding to different lengths of representation are the following.

- `int`
- `short int`
- `long int`
- `long long int`

Apart from these, the keyword `unsigned` can be also used with any of the above types to denote that the number should be considered as an unsigned number, i.e., its left most bit should be considered to have positive weight, as against negative weight for signed numbers. If the number is unsigned, the largest number that can be represented using k bits is $2^k - 1$.

The following program displays the size of different types of integers available in C and the maximum and minimum values that can be represented using type `int` and the maximum unsigned value that can be represented using type `unsigned int`.

```
#include<stdio.h>
#include<limits.h>
int main()
{
   printf("an integer occupies %d bytes \n",sizeof(int));
   printf("a short integer occupies %d bytes \n",sizeof(short int));
   printf("a long long integer occupies %d bytes \n",sizeof(long long int));
   printf("maximum integer is %d \n",INT_MAX);
   printf("minimum integer is %d \n",INT_MIN);
   printf("maximum unsigned integer is %u \n",UINT_MAX);

   return 0;
}
```

When this program was executed on my computer, the output obtained is as given below.

```
an integer occupies 4 bytes
a short integer occupies 2 bytes
a long long integer occupies 8 bytes
maximum integer is 2147483647
minimum integer is -2147483648
maximum unsigned integer is 4294967295
```

Note that, the output may change from one system to another. The general rule is that a `short int` variable is at least of length 16-bits and at most the length of `int`. Data type `int` has length at least 16-bits. Data type `long int` is of length at least 32-bits and its length cannot be less than that of `int`. Data type `long long int` is of length at least 64-bits and its length cannot be less than that of `long int`.

Overflow errors

If we use a particular type of integer to represent a variable, say for example the declaration *int a*, and we try to represent a number by this variable *a* which is beyond the range of values that can be represented by *int*, the result will be incorrect. Similarly, suppose we have two integer variables *int x,y* such that values of `x` and `y` are within permissible limits of values that can be represented by `int`, but the value of `x+y` is outside the range of `int`, then the expression `x+y` will produce an incorrect result. While doing arithmetic operations, a programmer has to be aware of such pitfalls and declare variables of the correct type depending on the values that may get assigned to the variable. If such overflow errors occur, the compiler will not be able to give any compilation error, warning or runtime error message, because the error is logical.

Consider a program with two integers `x, y`, where `x` and `y` store large integer values, such that their sum is outside the range of an integer. We will show different ways of computing the sum of `x` and `y`. Some of these methods will produce overflow errors and there are some ways to avoid an overflow error using appropriate data types.

Attempt 1:

In the following program, the result of the addition is stored to an integer which causes an overflow which in turn produces a wrong output.

```
#include<stdio.h>
#include<limits.h>
int main()
{
   int x, y;
   int z;

   x=1234567890;
   y=x;
   z=x+y;
   printf("%d+%d is %d!!!\n", x, y, z);

   return 0;
}
```

Output

```
1234567890+1234567890 is -1825831516!!!
```

Attempt 2:

In the program given below, the type of the variable which stores the result is changed to `long long int`. This program also produces a wrong output because in the instruction `z=x+y;`, the computation $x + y$ is performed first, before the resultant value is assigned to `z`. Thus the program still has an overflow error.

```
#include<stdio.h>
#include<limits.h>
int main()
{
   int x, y;
   long long int z;

   x=1234567890;
   y=x;
   z=x+y;
   printf("%d+%d is %lld!!!\n", x, y, z);

   return 0;
}
```

Output

```
1234567890+1234567890 is -1825831516!!!
```

Attempt 3:
In the program given below, the result of the computation `x+y` is type casted to `long long int` in an effort to avoid overflow. But this still doesn't work, because the evaluation of `x+y` is performed first which causes an overflow. Afterwards, this wrong result is getting type casted to `long long int`, which will be still incorrect.

```
#include<stdio.h>
#include<limits.h>
int main()
{
   int x, y;
   long long int z;

   x=1234567890;
   y=x;
   z=(long long int)(x+y);
   printf("%d+%d is %lld!!!\n", x, y, z);

   return 0;
}
```

Output

```
1234567890+1234567890 is -1825831516!!!
```

Attempt 4:
In the program given below, the value of the first variable `x` is type casted to `long long int` and it is added with the value of `y`. In this addition, the operands are of different lengths. So the value of the smaller length operand is internally converted to the longer type `long long int` while doing the addition. The resultant value is considered to be of type `long long int`. This will be a correct sum and it is assigned to `z` which is also of type `long long int`. Thus, there is no overflow in this program and the output is correct.

```
#include<stdio.h>
#include<limits.h>
int main()
{
   int x, y;
   long long int z;

   x=1234567890;
   y=x;
   z=(long long int)x+y;
   printf("%d+%d is %lld!!!\n", x, y, z);

   return 0;
}
```

Output

```
1234567890+1234567890 is 2469135780!!!
```

Attempt 5:
The following program handles the chances of overflow in a different way. For computing the sum of `x` and `y`, the value of `x` is first assigned to `z` which is of type `long long int`. Now, for computation of `z+y` in the instruction `z=z+y;`, since operands are of different lengths, their values are internally converted to the bigger type, which is `long long int` and then the addition is performed and the result obtained will be of type `long long int`. So far, the result is correct. When the result of evaluation of `z+y` is assigned to `z`, it will not cause any overflow, because `z` is `long long int`.

```
#include<stdio.h>
#include<limits.h>
int main()
{

   int x, y;
   long long int z;

   x=1234567890;
   y=x;
   z=x;
   z=z+y;
   printf("%d+%d is %lld!!!\n", x, y, z);

   return 0;
}
```

Output

```
1234567890+1234567890 is 2469135780!!!
```