Basic Assignment and Arithmetic Operators

Assignment Operator '='

```
int count ;

count = 10 ;
```

- The first line declares the variable `count`.
- In the second line an assignment operator (=) is used to store 10 in the location of `count`.

- In C language '`count = 10`' is called an expression (with side effect) where '=' is called the assignment operator (not equality).
- Value of the expression is 10.
- The semicolon ';' converts the expression to a statement.

Let the next statement be

```
count = 2*count + 5;
```

- In this statement the variable `count` is used on two sides of the assignment operator. There are two constants 2 and 5, and three operators '=' (assignment), '*' (multiplication) and '+' (addition).
- `count = 2*count + 5` is an expression (excluding the semicolon).

The informal semantics (action, meaning) of this expression is the following:

- The content (r-value) of `count` is multiplied by 2 and 5 is added to the result ($10 \times 2 + 5$ is 25).
- The final value, 25 is stored in the location (l-value) of `count`.
- The value of the expression is 25.

int count;

count ***int***

count = 10 ;

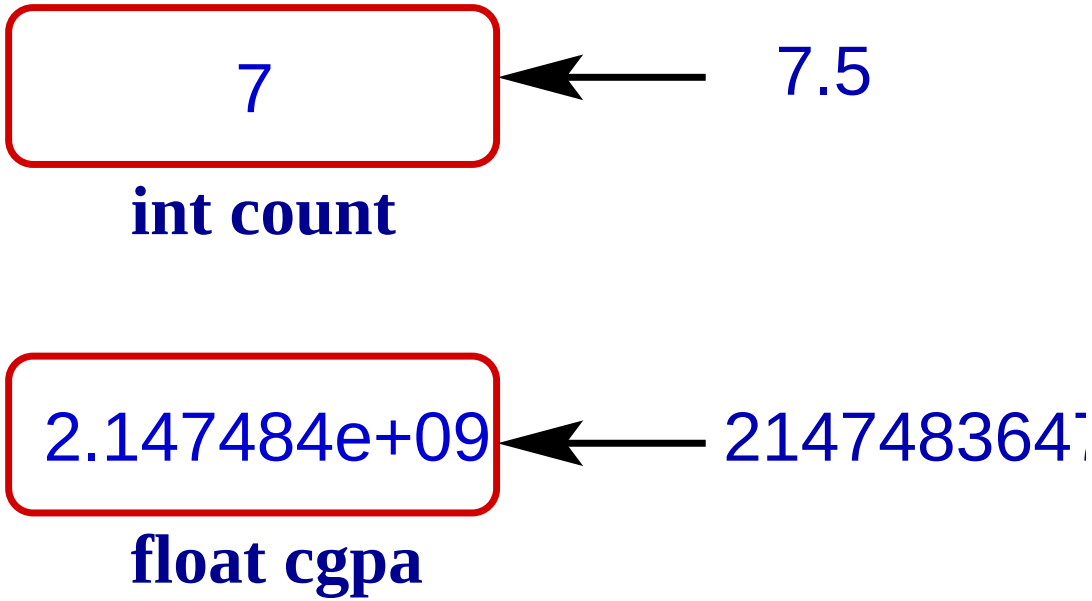
count

count = 2*count + 5;

count

Assignment from One Type to Another

- A `float` data can be assigned to an `int` variable, but there may be loss of precession.
- Similarly an `int` data can also be assigned to a variable of type `float`; but again there may be loss information.
- This process is called type casting.


7
7.5
int count
2.147484e+09
214748364
float cgpa

```
#include <stdio.h>
int main() // temp1.c
{
    int count = (int) 7.5 ;
    float cgpa = (float) 2147483647;
    printf("count: %d\n", count);
    printf("cgpa: %e\n", cgpa);
    return 0 ;
}
```

Output

```
$ cc -Wall temp.c
$ ./a.out
count:  7
cgpa:  2.147484e+09
```

Note

- The assignment of a floating-point data to an int variable (or the opposite) is not a simple operation due to the difference in the internal representations.
- In the first case the fractional part is removed and 7 is stored in 32-bit integer representation (2's complement form).
- If the data was 0.75, the integral part zero will be stored.

Note

- In the second case 2147483647 is converted to floating-point form (IEEE 754 single precession format). In this format a lesser number of bits are available for storing the significant digits (23 bits), so there will be a loss precession.

Five Basic Arithmetic Operators

+, −, *, /, %

- The first four operators have their usual meaning - addition, subtraction, multiplication and division.
- The last operator '%' extracts the remainder[a]. It may be called the mod operator.

[a] $a\%b$ - the first operand a should be non-negative integer and the second operand b should be a positive integer.

```
#include <stdio.h>
int main() // temp2.c
{
    printf("0%%10 = %d\n", 0%10);
    printf("4%%10 = %d\n", 4%10);
    printf("10%%4 = %d\n", 10%4);
    printf("-10%%4 = %d\n", -10%4);
    return 0 ;
}
```

Output

```
$ cc -Wall temp2.c
$ ./a.out
0%10 = 0
4%10 = 4
10%4 = 2
-10%4 = -2
```

The mod operator (%) does not extracts the remainder correctly for negative operands.

Operator Overloading

The first four operators $+, -, *, /$ can be used for `int`, `float` and `char` data[a]. But the fifth operator cannot be used on float data.

[a] The actual operations of addition, subtraction etc. on int and float data are very different due to the difference in their representations.

Mixed Mode Operations

- Mixed mode operations among `int`, `float` and `char` data are permitted.
- If one operand is of type `float` and the other one is of type `int`, the `int` data will be converted to the closest `float` representation before performing the operation.

```
#include <stdio.h>
int main() // temp3.c
{
    int n = 4 ;
    float a = 2.5 ;
    char c = 'a' ; // ASCII value 97
    printf("%d*%f = %f\n", n, a, n*a);
    printf("%d*%f+%c = %f\n",n,a,c,n*a+c);
    return 0 ;
}
```

Output

```
$ cc -Wall temp3.c
$ ./a.out
4*2.500000 = 10.000000
4*2.500000+a = 107.000000
```

Computer Arithmetic!

$$\frac{1}{3} \times 30.0 = 0.0$$

One should be careful about the division operation on int data.

```
#include <stdio.h>
int main() // temp4.c
{
    printf("1/3*30.0=%f\n", 1/3*30.0);
    return 0 ;
}
```

Output

```
$ cc -Wall temp4.c
$ ./a.out
1/3*10.0=0.000000
```

```
#include <stdio.h>
int main() // temp4a.c
{
    printf("10.0*1/3=%f\n", 10.0*1/3);
    printf("10.0*(1/3)=%f\n", 10.0*(1/3));
    return 0 ;
}
```

Output

```
$ cc -Wall temp4a.c
$ a.out
10.0*1/3=3.333333
10.0*(1/3)=0.000000
```

Computer Arithmetic!

$$2147483647 + 1 = -2147483648$$

Addition may give wrong result due to range overflow.

```
#include <stdio.h>
int main() // temp14.c
{
    int n = 2147483647;
    printf("n+1: %d\n", n+1) ;
    return 0 ;
}
```

Output

```
$ cc -Wall temp14.c
$ ./a.out
n+1:  -2147483648
```

Computer Arithmetic!

$$10^5 + 10^{-5} = 10^5$$

There will be loss of precession in floating point arithmetic.

```
#include <stdio.h>
int main() // temp8.c
{
    float a = 1.0e-40, b = 1.0e+5, c;
    c = a+b ;
    printf("%e + %e = %e\n", a, b, c);
    if(b == a+b) printf("Equal\n");
    else printf("not Equal\n");
    return 0 ;
}
```

Output

```
$ cc -Wall temp8.c
$ a.out
9.999946e-41 + 1.000000e+05 = 1.000000e+05
Equal
$
```

Computer Arithmetic!

$$a + (b + c) \neq (a + b) + c$$

Law of associativity may not hold.

```
#include <stdio.h>
int main() // temp8a.c
{
    float a=0.3e-14, b=0.3e-14, c=1.0e+5;
    if(a+(b+c) == (a+b)+c) printf("Equal\n");
    else printf("not Equal\n");
    return 0 ;
}
```

Output

```
$ cc -Wall temp8a.c
$ a.out
not Equal
$
```

Precedence and Associativity

- All these four operators have left-to-right associativity.
- $*, /, \%$ have the same precedence and it is higher than $+, -$ which also have the same precedence.

'=' is Right Associative

```
int count = 10, n ;

n = count = 2*count + 5 ;
```

The variable `n` gets the updated value of `count` i.e. 25.

Precedence of =

The precedence of assignment operator(s) is lower than every other operator except the comma ‘,’ operator.

Computer Arithmetic!

$1.3 \neq$ `float a` $= 1.3;$

```
#include <stdio.h>
int main() // temp5.c
{
    float a = 1.3 ;

    if(a == 1.3) printf("Equal\n") ;
    else printf("Not equal\n") ;
    return 0 ;
}
```

Output

```
$ cc -Wall temp5.c
$ ./a.out
Not equal
```

Error

- Division of `int` data by zero gives error at run time[a].
- But the division of `float` or `double` data by zero does not generate any run time error. The result is inf[b].

[a] GCC error message is funny.

[b] This value can be used.

```
#include <stdio.h>
int main() // temp9.c
{
    int n = 10, m ;
    printf("Enter an integer: ");
    scanf("%d", &m);
    printf("n/m: %d\n", n/m);
    return 0 ;
}
```

Output

```
$ cc -Wall temp9.c
$ ./a.out
Enter an integer:  0
Floating point exception
```

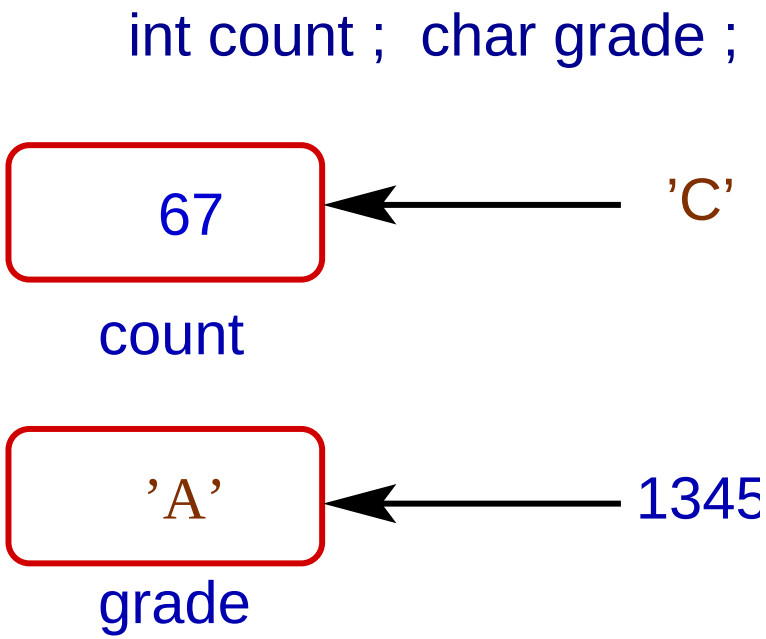
```
#include <stdio.h>
#include <math.h>
int main() // temp10.c
{
    float n = 10.0, m, r ;
    printf("Enter a number: ");
    scanf("%f", &m);
    printf("n/m: %f\n", r = n/m);
    printf("atan(%f) = %f\n", r, atan(r));
    return 0 ;
}
```

Output

```
$ cc -Wall temp10.c -lm
$ a.out
Enter a number:  0
n/m:  inf
atan(inf) = 1.570796
```

Integer ↔ Character

- If a `char` data is assigned to an `int` type variable, the ASCII value of the data is stored in the location.
- But if an `int` data (32-bit size) is assigned to a `char` type variable (8-bit size), the least significant 8-bits of the data are stored in the location.


int count ; char grade ;
67
'C'
count
'A'
1345
grade

```
#include <stdio.h>
int main() // temp6.c
{
    int count = 'C' ;
    char grade = 1345 ;
    printf("count: %d, grade: %c\n",
            count, grade) ;
    return 0 ;
}
```

Output

```
$ cc -Wall temp6.c
temp6.c:  In function ‘main’:
temp6.c:5:  warning:  overflow in
implicit constant conversion
$ ./a.out
count:  67, grade:  A
```

Note

- The ASCII code for 'C' is 67 and that is stored in the location for `count`.
- The internal representation of 1345 is 0000 0000 0000 0000 0000 0101 0100 0001. The decimal value of the least significant byte (8-bit) 0100 0001 is 65, the ASCII code for 'A'.

Pre and Post Increments

```
int count = 10, total = 10 ;

++count ;

total++ ;
```

- Both `++count` and `total++` are expressions with increment operators. The first one is pre-increment and the second one is post-increment.
- At the end of execution of the corresponding statements, the value of each location is 11.
- But the value of the expression `++count` is `11` and that of the `total++` is `10`.

Pre and Post Decrements

```
int count = 10, total = 10 ;

--count ;

total-- ;
```

Similarly we have pre and post decrement operators.

More Assignment Operators

```
int count = 10, total = 10 ;

count += 5*total ;
```

The meaning of the expression
`count += 5*total` is equivalent to
`count = count + 5*total`.

Unary '+' and '−'

The unary '−' and '+' have their usual meaning with higher precedence than $*, /, \%$.