3. OPERATORS & CONVERSIONS

3.1 Arithmetic Operators in C

In C, the binary arithmetic operators are + (addition), - (subtraction), * (multiplication), / (division), and the modulus operator % (remainder). As expected, all except % operate on integer, character, and real operands. The arithmetic operators associate (are evaluated) left to right, following the precedence rules given below:

high * / %

low + -

That means that

a * b + c * d

is evaluated as through it were written as

(a * b) + (c * d)

Ex : evaluate the following expressions.

(i) (A + B) / (C + D)

(ii) A + B / (C + D)

(iii) (A + B) / C + D

(iv) A * B / C * D

3.2 Conversion Characters

In C, for the ***printf*** and ***scanf*** functions following **conversion characters** are used:

Type	Reading with scanf	Printing with printf
int	%d	%d
short	%hd	%d
long	%ld	%ld
unsigned short	%hu	%u
unsigned int	%u	%u
unsigned long	%lu	%lu
float	%f	%f
double	%lf	%lf
long double	%Lf	%Lf
float in scientific notation	%e	%e
hexadecimal	%x	%x

octal	%o	%o
character	%c	%c
string	%s	%s

Eg. : write a program to print sum of two numbers.

```
# include  < stdio.h >

void main(void)
{
        int a, b, c;
        a = 5;
        b = 3;
        c = a + b;
        printf ("Sum of the two numbers = %d \n", c );
}
```

In general, *printf*( ) writes anything in the formatting control string (between the quotation marks) as is. One exception is a %, which it takes as the start of a description of how to write a value. *printf*( ) regards the % and the letter after it as a **directive**. *printf*( ) expects the % to be followed by a letter specifying the **value's type**. %d indicates a decimal integer, %f a floating point value.

Ex : Of course, using % to indicate formatting codes causes a problem: How do we write a percent sign?

Normally, %d uses just enough space to print the entire value, and %f prints floating point values with exactly **six digits** after the decimal point and however many digits are necessary preceding it. We change these defaults by providing an optional field width.

We can re-write the above program as:

```
# include  < stdio.h >

void main(void)
{
        int a = 5, b = 3 ;
        printf ("Sum of the two numbers = %d \n", a+b );
}
```

or

```
# include  < stdio.h >

void main(void)
{
        int a = 5, b = 3 ;
        printf ("Sum of the two numbers  %d and %d = %d \n", a , b, a+b );
}
```

As with **printf**, **scanf**'s arguments are a formatting control string, enclosed in quotation marks, and a list of variables.

The following program inputs the values of a and b rather than having them assigned in the source:

```
# include  <stdio.h>

void main(void)
{
        int a, b;
        printf ("Input two numbers : \n");
        scanf ("%d%d", &a, &b);
        printf ("Sum of the two numbers = %d \n, a+b );
}
```

The **&** operator gives the **address** of a variable in storage. Thus **scanf** is supplied with the address of the memory associated with variable a and can write the input integer value into that location. This procedure is necessary because arguments in C routines are **call-by-value**.

Ex : Write a program to convert the distance of a **marathon** in miles and yards to kilometers. In English units a marathon is defined to be 26 miles and 385 yards. To convert miles into kilometers multiply by **conversion factor** 1.609. To convert yards into miles divide by 1760.

Eg :

```
        # include < stdio.h >

        void main (void)
        {
                char C1 , C2 , C3;
                int  i ;
                float x ;
                double y ;

                printf ("\n %s \n %s" , " Input the character " , \
                        " An int , a float , and a double " ) ;
                scanf ("%c%c%c%d%f%lf " , &C1 , &C2 , &C3 , &i , &x , &y ) ;

                printf (" \n Here is the data that you typed in : \n ") ;

                printf (" %3c%3c%3c%5d%17e%17e\n\n " , C1 , C2 , C3 , i , x , y ) ;
        }
```

3.3 Relational and Logical Operators

We use relational operators to test whether a particular **relationship** holds between two values. C has relational operators to compare values for

equality	==
inequality	!=
greater than	>
less than	<
greater than or equal	>=
less than or equal	<=

A relational operator returns either **nonzero** (true) if the specified relation between the operands holds or **zero** (false) if it doesn't.

We often need to combine relational tests in a single expression. For example, to verify that a value falls between two other values, we need to test whether it is less than the first value and greater than the second. We do so using the **logical operators**:

logical AND	&&
logical OR	||
logical NOT (negation)	!

These return **nonzero** (true) or **zero** (false) according to the expression. Logical AND and OR interpret nonzero operands as true and zero operands as false. Negation (NOT) returns one if its operand is zero, and zero otherwise.

The relational and logical operator precedence is:

high	!
	> >= < <=
	== !=
	&&
low	||

When the precedence is the same, evaluation proceeds left to right.

Thus a == b && c != d is (a == b) && (c != d)

and a == b && c == d || x == y is ((a == b) && (c == d)) || (x == y)

Relational operators have lower precedence than arithmetic operators, so we rarely need to **parenthesize** when we combine them with the relational operators. The expression

x < min || x > max

is evaluated as

(x < min) || (x > max)

But all of these operators have precedence, which is higher than the precedence *assignment* (=), so we often have to use parentheses to guarantee that comparisons are done after any assignments, as in

((c = getchar ( )) != ' \n ')

to achieve the desired result of assignment to C and then comparison with ' \n '.

The same statement written without the parenthesis,

(c = getchar ( ) != ' \n ')

is evaluated as

(c = (getchar ( ) != ' \n '))

That is, as though we were assigning the result of comparing the newly read character with '\n' (newline). **In general, it's a good idea to parenthesize relational operators, since they are in the middle of the precedence hierarchy**.

Normally both operands of an operator are evaluated and then the operator applied to the resulting values. But it's possible that one of the operands of a logical operator might never be evaluated. The reason is that the operator's value can often be determined only after its first operand has been evaluated - eliminating the need to evaluate the other operand. When evaluating the logical AND in the statement

a == b && c != d

the clause a == b is evaluated first. If it is false, the value of the entire logical expression if false, regardless of how the second clause would evaluate, so the second clause is not evaluated. **If there are multiple clauses connected by logical AND, the first false one terminates the evaluation**. In this example, only when a == b evaluates to true is the second clause, the c!=d, evaluated.

In logical OR the first true clause terminates evaluation of all succeeding clauses. In the following statement, **x is not compared with *max* when x is less than *min*.**

```
x < min || x > max
```

3.4 Increment and Decrement Operators

C provides **two special sets of shorthand operators** for the common operations of incrementing and decrementing by one: **++** add one to its operand, **--** subtracts one from its operand. There are two forms of these operators, *prefix* (preceding its operand) and *postfix* (following its operand). As either a *prefix* or a *postfix* operator, ++ adds one to its operand. But when we use it as a *postfix* operator, ++ first evaluates the operand, providing its value to the rest of the statement or expression, and then adds one to it. And when we use it as a *prefix* operator, it does the additions first and makes the new value of the variable available to the expression. *prefix* and *postfix* -- behave similarly to ++ except that they decrement their operand.

Notice that as a statement not involving other side effects

```
i ++;
```

is the same as

```
++ i;
```

But as an example,

```
x = 10;

y = ++x;
```

sets y to 11

```
x = 10;

y = x++;
```

sets y to 10.

x becomes 11 in both cases.

Note : ++ , -- and unary - have higher precedence than the other arithmetic operators.

Eg. : Following program explain you how to test the increment and decrement operators

```
# include  < stdio.h >

void main (void)
{
        int x = 10, y ;
        y = ++x ;
        printf (" Values of x and  y  = %d and %d \n", x , y );
}
```

or

```
# include  < stdio.h >

void main (void)
{
        int x = 10 ;
        printf (" Values of x and  y  = %d and %d \n", x , ++x );
}
```

3.5 Assignment Operators

We have seen that in C, unlike most other programming languages, assignment not only assigns the result of an expression to a variable but also returns this value. One benefit is that we can use assignment operators more than once in a single statement. We can initialize more than one variable to some initial value (say, 0) with

```
a = b = c = 0;
```

The assignment operator is evaluated right to left, so this multiple assignment is equivalent to

```
a = (b = (c = 0));
```

More about Assignment Operators

```
a = a + b;    is      a += b;

a = a - b;    is      a -= b;

a = a * b;    is      a *= b;

a = a / b;    is      a /= b;

a = a % b;    is      a %= b;
```

3.6 Operations on Bits

On several occasions, we remark that the **C language was developed with systems programming** applications in mind. Pointers are the perfect case in point, since they give the programmer an enormous amount of control over and access into the computer's memory. Along these same lines, systems programmers frequently must get in and "**twiddle with the bits**" of particular computer words. In this area, C once again shines above other programming languages, such as Pascal, because it provides a host of operators specifically designed for performing operations on individual bits.

On most computer systems, a ***byte*** consists of 8 smaller units called ***bits***. A *bit* can assume either of two values: 1 or 0. So a *byte* stored at address in a computer's memory, for example, might be conceptualized as a string of 8 binary digits as shown:

0 1 1 0 0 1 0 0

Bitwise Operators

The operators we've seen so far are common to most higher-level programming languages. C also provides less common operators for shifting bits.

Symbol	Operation
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (Exclusive - OR)
~	Ones Complement
<<	Left Shift
>>	Right Shift

These operations allow us to deal with the details of the machine. We need them because some hardware operations require that individual bits be set and then other bits read to determine whether the operation succeeded. We can also use them to provide faster versions of some numerical operations, and to optimize our use of storage.

All of the operators listed in above table, with the exception of the ones complement operator (~), are binary operators and as such take two operands. Bit operations can be performed on any type of integer value in C (short, long or unsigned) and on characters, but cannot be performed on floating point values.

The Bitwise Logical Operators

There are three bitwise logical operators:

Bitwise AND &

Bitwise OR |

Bitwise XOR ^

These operators work on their operands bit by bit, setting each bit in the result, as shown in below table.

Operands		Result		
Op1	Op2	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND is frequently used for "***masking***" operations. That is, this operator can be used to easily set specific bits of a data item to 0. For example,

```
x = y & 3
```

will assign to x the value of y bitwise AND with the constant 3. This has the effect of setting all of the bits in x, other than the rightmost two bits, to 0, and of "*preserving*" the rightmost two bits from y.

As with all binary arithmetic operators in C, the binary bit operators can also be used as assignment operators by "***tacking***" on an equal sign. So the statement

```
x &= 3;
```

will perform the same function as

```
x = x & 3;
```

and will have the effect of setting all but the rightmost two bits of x to 0.

When using constants in performing bitwise operations, it is usually more convenient to express the constants in either octal or hexadecimal notation. The choice as to which to use is usually influenced by the size of the data that you are dealing with. For example, when dealing with 16-bit computers, hexadecimal notation is often used, since 16 is an even multuple of 4 (the number of bits in a hexadecimal digit).

Bitwise OR is frequently used when it is desired to set some specified bits of a word to 1. For example,

```
x = x | 3;
```

will set the two rightmost bits of x to 1, regardless of the state of these bits before the operation was performed.

The bitwise **Exclusive-OR operator**, which is often called as **XOR operator**. One interesting property of the Exclusive-OR operator is that any value Exclusive-OR with itself produces 0. This trick is frequently used by assembly language programmers as a fast way to set a value to 0, or to compare two values to see if they are equal. This method is not recommended for use in C programs, however, as it won't save any time and will most likely make the program more obscure.

Ex : evaluate the following :

(i) 0xf1 & 0x35 yields 0x31

(ii) 0xf1 | 0x35 yields 0xf5

(iii) 0xf1 ^ 0x35 yields 0xc4

Similarly we can do above three operations for the octal and binary as well.

The Ones Complement Operator

The ones complement operator is a **unary operator**, and its effect is to simply "*flip*" the bits of its operand. So each bit of the operand that is a 1 is changed to a 0, and each bit that is a 0 is changed to a 1.

The **ones complement operator** (~) should not be confused with the **arithmetic minus operator** (-) or with **the logical negation operator** (!). So if w1 is defined as an *int*, and set equal to 0, then -w1 still results in 0. If we apply the ones complement operator to w1, we end up with w1 being set to all ones, which is **negative** when treated as a signed value.

The ones complement operator is useful in operations where we don't know the precise bit size of the quantity that we are dealing with. Its use can help make a program more "***portable***" - i.e., less dependent on the particular computer that the program is running on, and therefore easier to get running on a different machine. For example, in order to set the low-order bit of an *int* called w1 to 0, we can AND w1 with an *int* consisting of all 1's except for a single 0 in the rightmost bit. So a statement in C such as

```
w1 &= 14 ;
```

will work fine on machines in which an integer is represented by 16 bits. However, on machines that represent integers using more than 16 bits, this statement will not produce the desired results.

If we replace the preceding statement with

```
w1  &=  ~1;
```

then w1 will get AND with the correct value on any machine, since the ones complement of 1 will be calculated and will consist of as many leftmost one bits as are necessary to fill the size of an *int* (15 leftmost bits on a 16-bit integer machine, and 31 leftmost bits on a 32-bit integer machine).

Bit Shifts

The ***left*** (<<) and ***right*** (>>) shifts take an operand and return its value **shifted left or right** by a specified number of bits. The operand itself is not affected. The form of the operators is

op **>>** ***n*** and ***op*** **<<** ***n*** .

The Left Shift Operator

When a **left shift operation** is performed on a value, the bits contained in the value are literally shifted to the left. Associated with this operation is the number of places (bits) that the value is to be shifted. The left shift moves the bits to the left and sets the right most bit (least significant bit) to 0. The left most bit (most significant bit) shifted out is discarded.

```
int  A = 65;     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1

( A << 1 )       0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 =        130

( A << 2 )       0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 =        260
```

Left shifting actually has the effect of multiplying the value that is shifted by two. In fact, some C compilers will automatically perform multiplication by a power of two by left shifting the value the appropriate number of places, since shifting is a much faster operation than multiplication on most computers.

The Right Shift Operator

As implied from its name, the **right shift operator** >> shifts the bits of a value to the right. Bits shifted out of the low-order bit of the value are lost.

```
int  A = 65;     0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1

( A >> 1 )       0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 =        32

( A >> 2 )       0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 =        16
```

Right shifting actually has the effect of dividing the value that is shifted by two.

Right shifting an unsigned value will always result in 0's being shifted in on the left, i.e., through the high-order bits. What is shifted in on the left for signed values depends on the sign of the value that is being shifted and also on how this operation is implemented on your particular computer system. If the sign bit is 0 (meaning the value is positive), then 0's will be shifted in no matter what machine we are talking about. However if the sign bit is 1, then on some machines 1's will be shifted in, and on others 0's will be shifted in. This former type of operation is known as an **arithmetic right shift**, while the latter is known as **a logical right shift**.

So, for example, if w1 is an *unsigned int*, which is represented in 16 bits, and w1 is set equal to binary 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1, then shifting w1 one place to the right with the statement

```
w1 >>= 1;
```

will set w1 equal to binary 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 :

```
w1          1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1

w1 >> 1     0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1
```

If w1 were declared to be a (signed) *int*, then the same result would be produced on some computers, while, on others, the result would be

```
1 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1
```

if the operation were performed as an **arithmetic right shift**.

It should be noted that the C language does not produce a defined result if an attempt is made to shift a value to the left or right by an amount that is greater than or equal to the number of bits in the size of the data item. So on a machine that represents integers in 32 bits, for example, shifting an integer to the left or right by 32 or more bits is not guaranteed to produce a defined result in your program.

Bit Fields

With the bit operators discussed above, we can proceed to perform all sorts of sophisticated operations on bits. Bit operations are frequently performed on data items that contain "***packed***" information. Just as a *short int* can be used to conserve memory space on many computers, so can we pack information into the bits of a byte or word if we do not need to use the entire byte or word to represent the data. For example, flags that are used for a **boolean TRUE or FALSE condition can be represented in a single bit** on a computer. Declaring a variable that will be used as a flag will use at least 8 bits (one byte), and most likely at least 16 bits on most computer systems. And if we needed to store many flags inside a large table, then the amount of memory that would be "***wasted***" could become significant.

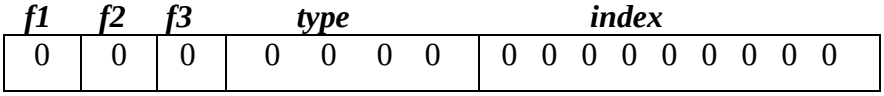
There are two methods in C that can be used to pack information together to make better use of memory. One way is to simply represent the data inside a normal int, for example, and then access the desired bits of the int using the bit operators we have just described. Another way is to define a structure of packed information using a C construct known as a ***bit field***.

To illustrate how the first method can be used, suppose we needed to pack five data values into a single word because we had to maintain a very large table of these values in memory. Assume that three of these data values are flags, which we will call *f1, f2,* and *f3*; the fourth value is an integer called *type* which ranges from 1 through 12; and the final value an integer called *index*, which ranges from 0 to 500.

In order to store the values of the flags *f1, f2,* and *f3*, we would only require three bits of storage, one bit for the TRUE/FALSE value of each flag. To store the value of the integer *type*, which ranges from 1 to 12, would require four bits of storage. Finally, to store the value of the integer *index*, which can assume a value from 0 to 500, we would need nine bits. Therefore, the total amount of storage needed to store the five data values *f1, f2, f3*, *type*, and *index*, would (conveniently) be 16 bits. We could define an integer variable that could be used to contain all five of these values, as in

```
unsigned int   packed_data ;
```

and could then arbitrarily assign specific bits or *fields* inside *packed_data* to be used to store the five data values. One such assignment is depicted in following figure, which assumes that the size of *packed_data* is 16 bits. If it were larger than 16 bits, then we could still conceptualize these field assignments as occupying the 16 rightmost bits of the integer.



Bit field assignments in packed_data

We can now apply the correct sequence of bit operations to *packed_data* to set and retrieve values to and from the various fields of the integer. For example, we can set the *type* field of *packed_data* to 7 by shifting the value 7 the appropriate number of places to the left :

```
packed_data  =  7  <<  9 ;
```

Or we can set the *type* field to the value n, where n is between 0 and 15, by the statement

```
packed_data  =  n  <<  9 ;
```

(To ensure that n is between 0 and 15, we can AND it with 0xf before it is shifted.) Of course, the preceding statements will work only if we know that the *type* field is zero, otherwise we must zero it first by AND it with a value (frequently called a mask) that consists of 0s in the four bit locations of the *type* field and 1s everywhere else:

```
packed_data  &=  0xe1ff ;
```

The C language does provide a more convenient way of dealing with bit fields. This method employs the use of a special syntax in the structure definition that allows you to define a field of bits and assign a name to that field. Whenever the term "bit fields" is applied to C, it is this approach that is referenced.

In order to define the bit field assignments previously mentioned, we can define a structure called packed_struct, for example, as follows:

```
struct  packed_struct
{
        unsigned  int   f1:1 ;
        unsigned  int   f2:1 ;
        unsigned  int   f3:1 ;
        unsigned  int   type:4 ;
        unsigned  int   index:9 ;
} ;
```

The structure ***packed_struct*** is defined to contain five members. The first member, called *f1*, is an unsigned int. The :1 that immediately follows the member name specifies that this member is to be stored in one bit. The flags *f2* and *f3* are similarly defined as being a single bit in length. The member *type* is defined to occupy four bits, while the member *index* is defined as being nine bits long.

The C compiler automatically packs the preceding bit field definitions together. The nice thing about this approach is that the fields of a variable defined to be of type ***packed_struct*** can now be referenced in the same convenient way normal structure members are referenced. So, if we were to declare a variable called ***packed_data*** as follows:

```
struct   packed_struct   packed_data ;
```

then we could easily set the *type* field of packed_data to 7 with the simple statement

```
packed_data.type  =  7 ;
```

or we could set this field to the value of n with the similar statement

```
packed_data.type  =  n ;
```

In this last case, we needn't worry about whether the value of n is too large to fit into the *type* field; only the low-order four bits of n will be assigned to packed_data.type.

Extraction of the value from a bit field is also automatically handled, so the statement

```
n  =  packed_data.type ;
```

will extract the *type* field from packed_data (automatically shifting it into the low-order bits as required) and assign it to n.

We can also include "normal" data types within a structure that contains bit fields. So if we wanted to define a structure that contained an *int*, a *char* and two one-bit flags, the following definition would be valid:

```
struct  table_entry
{
        int             count ;
        char            C ;
        unsigned int    f1:1 ;
        unsigned int    f2:1 ;
} ;
```

Certain points are worth mentioning with respect to bit fields. Bit fields may always be treated as unsigned on some machines, whether or not they are declared as such. Furthermore, some C compilers do not support bit fields that are larger than the size of a word. A bit field cannot be dimensioned; that is, we cannot have an array of fields, such as flag :1[5]. **Finally, we cannot take the address of a bit field, and since this is the case, there is obviously no such thing as a variable of type "pointer to bit field.**"

Bit fields are packed into words as they appear in the structure definition. If a particular field does not fit into a word, then the remainder of the word is skipped and the field is placed into the next word. If the following structure definition were used

```
struct  bits
{
        unsigned int    f1:1 ;
        int             word ;
        ùnsigned int    f2:1 ;
} ;
```

then *f1* and *f2* would not get packed into the same word since the definition of word comes between them. The C compiler will *not* rearrange the bit field definitions to try to optimize storage space.

A bit field that has no name can be specified to cause bits inside a word to be "skipped." So the definition

```
struct  x_entry
{
        unsigned  int   type : 4 ;
        unsigned  int   : 3 ;
        unsigned  int   count : 9 ;
} ;
```

would define a structure x_entry that contained a four-bit field called *type* and a nine-bit field called *count*. The unnamed field specifies that three bits separate the *type* from the *count* field.

A final point concerning the specification of fields concerns the special case of an unnamed field of length 0. This may be used to force alignment of the next field in the structure at the start of a word boundary.

3.7 Other Operators

There are three other operators of interest, the *comma operator*, the ***sizeof*** *operator*, and the ***conditional operator***. None of these have analogs in other high-level languages like Pascal or Basic.

The Comma Operator

We use the ***comma*** *operator* to link related expressions together, making our programs more compact. A comma-separated list of expressions is treated as a single expression and evaluated left to right, with the value of the rightmost expression returned as the expression's value.

We can use the comma operator to rewrite a program fragment to exchange the values of two variables x and y

```
temp = x ;                          /*  swap x and y  */
x = y ;
y = temp ;
```

more concisely as

```
temp = x , x = y , y = temp ;       /*  swap x and y  */
```

We can also use the comma operator to eliminate embedded assignments from tests.

```
while ( ( C = getchar( ) )  !=  EOF )
        putchar (C) ;
```

as

```
while (  C = getchar( ) ,  C != EOF )
        putchar (C) ;
```

separating reading the character from testing for end of file. Because the comma operator evaluates left to right, the rightmost expression's value (here, the test for end of file) controls the *while*'s execution. Either method is acceptable and we find ourselves using them interchangeably; use the one you find easiest to read.

The comma operator has the lowest precedence of any of C's operators, so we can safely use it to turn any list of expressions into a single statement. *Warning:* ***The comma used to separate the parameters in function calls is not a comma operator and does not guarantee left-to-right evaluation***.

The Sizeof Operator

The ***sizeof operator*** returns the number of bytes in its operand, which may be a constant, a variable, or a data type. *sizeof* requires parentheses around a type but not around a variable.

What exactly is a byte? *sizeof* loosely defines it as the size of a character, which is eight bits on most but not all machines. If the variable is an array or another constructed type, *sizeof* returns the total number of bytes needed. For arrays, *sizeof* returns the size of the base type times the declared size of the array. If *a* is an array of 100 int**s**, *sizeof(a)* is 200, assuming 2-byte (16 bit) int**s**. The *sizeof* operator is unique because it is evaluated at compile time instead of run time. The compiler replaces the call with a constant.

The Conditional Operator

The ***conditional operator*** is an ***if*** statement in disguise.

expression ? true-expression : false-expression

It first evaluates ***expression***, and if it is nonzero, then evaluates and returns ***true-expression***. Otherwise, it evaluates and returns ***false-expression***. At most, one of ***true-expression*** and ***false-expression*** is evaluated.

The ***conditional operator*** provides a convenient shorthand for *if* statements that decide which of two values a particular variable should be assigned. We can use **? :** to rewrite the following *if* that decides which of two values is smaller

```
if ( x < y )
        min = x ;
else
        min = y ;
```

as

```
min = ( x < y ) ? x : y ;
```

We don't have to parenthesize the test, but doing so is a good idea; the parentheses help distinguish the test from the values returned.

We can nest conditional operators, which allows us to rewrite more complex decisions. The following messy expression returns the middle value in a set of three values, x, y, and z.

```
( x > y ) ? ( y > z ? y : ( ( x > z ) ? z : x ) )
          : ( y < z ? y : ( ( x < z ) ? z : x ) ) ;
```

In addition to parenthesizing the expression tested by the conditional operators, we parenthesized the nested conditional operators. Again, although not strictly necessary, this is a good habit to get into since it protects against possible precedence problems.

In their neverending quest for compact code most programmers have a tendency to go overboard with conditional operators. Using them leads to more concise, possibly more efficient, and definitely less readable code. Anything with more than a single nested conditional operator is better written using *if*s.

The conditional operator often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
for ( i = 0 ;  i < n ;  i++ )
        printf( "%6d%c", a[i] , ( i%10 == 9 || i == n-1 ) ?  '\n'  :  ' '  ) ;
```

A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent if-else. Another good example is

```
printf ( " You have %d item%s. \n",  n  ,  n==1  ?  " "  :  "s" ) ;
```

Ex : Write a function *lower*, **which converts upper case letters to lower case**, with a ***conditional operator*** instead of ***if-else***.

You can use the ***conditional operator*** in normal coding, but its main use is creating ***macros***. In C, definitions are frequently called ***macros***. This terminology is more often applied to definitions which take one or more arguments. An advantage of implementing something in C as a macro, as opposed to as a function, is that in the former case the type of the argument is not important. The ***conditional operator*** can be particularly handy when defining *macros*. The following defines a macro called MAX that gives the maximum of two values:

```
# define MAX ( a , b )( (  (a)  >  (b)   ?   (a)   :   (b)   )
```

This macro enables us to subsequently write statements such as

```
limit   =    MAX (  x + y  ,  min_value ) ;
```

The following macro can be used to test if a character is a lowercase character:

```
# define   IS_LOWER_CASE (x)      ( (  (x) >= ' a ' )  &&  (  (x) <= ' z  ' ) )
```

and thereby permits expressions such as

```
if ( IS_LOWER_CASE (c) )
        .......
```

to be written. We can even use this macro in a subsequent macro definition to convert a character from lowercase to uppercase, leaving any non-lowercase character unchanged.

```
# define TO_UPPER     ( IS_LOWER_CASE (x)  ?  (x) - ‘ a ’ + ‘ A ’ :  (x)  )
```

The program loop

```
while ( *string != ‘\0’ )
{
        *string = TO_UPPER ( *string ) ;
        ++string;
}
```

would sequence through the characters pointed to by *string*, converting any lowercase characters in the string to uppercase.

3.8 Operator Classes and Precedence

Much of a programming language’s power derives from the operators it provides. And C possesses a wide selection of operators. The following table summarizes these operators, the operations they perform, and their relative precedence. The table is organized in order of decreasing precedence.

Most operators expect their operands to agree in type. When they don’t, fairly straightforward automatic conversions occur (to be discussed in the next section). In addition, some operators expect operands of a certain type. The operation may still be performed if we provide an operand of an incorrect type-with strange and unanticipated results. *Warning:* ***It is your responsibility to guarantee that operands are the correct type, in range, and that the result has not produced overflow or underflow***. The alternative is for the compiler or run-time environment to capture incorrect operand types, as with languages such as Pascal. But doing so requires longer compilations or time-consuming run-time type checking. The benefit with C is that if our programs run, they run more efficiently.

Category	Operator	Associativity
primary	( ) [ ] . ->	left to right
unary	* & ++ -- ~ - sizeof(type-name) !	right to left
binary	* / % + - << >> < > <= >= == != & ^ | && ||	left to right
Conditional	? :	right to left
assignment	= += -= *= /= %= >>= <<= &= ^= |=	right to left
comma	,	left to right

3.9 Type Conversions and Casts

Our discussion of operators has ignored an important question: What happens when an operator's operands are not both the same type?

Automatic conversion in an arithmetic expression

An arithmetic expression such as x+y has both a value and a type. For example if x and y are both variables of the same type, say int, then x+y is also an int. However, if x and y are of different types, then x+y is a mixed expression. Suppose x is a short and y is an int. Then the value of x is converted to an int and the expression x+y has type int. Note carefully that the value of x as stored in memory is unchanged. It is only a temporary copy of x that is converted during the computation of the value of the expression. Now suppose that both x and y are of type short. Eventhough x+y is not a mixed expression, automatic conversion again takes place. ie. both x and y are promoted to int and the expression is of type int.

General Rules

(1). Any **char** or **short**, including the **signed** and **unsigned** versions of these, is promoted to **int**

(2). If after the first step the expression is of mixed type, then according to the hierarchy of types

int < signed long int < unsigned int < unsigned long int < float < double < long double

the operand of lower type is promoted to that of the higher type and the value of the expression as a whole has that type.

In addition to automatic conversions in mixed expressions, an automatic conversion also can occur across an assignment.

eg. double d ; int i ;

d = i ;

Here i is converted to a double and then assigned to d. Expression as a whole is of type double.

i = d ;

Fractional part of d will be discarded.

Type Conversions Using the Cast Operator

As we've just discovered, C performs some type conversions automatically. There are times, however, when we want to force a type conversion in a way that is different from the automatic conversion. We call such a process ***casting*** a value. We specify a cast by giving a type in parentheses followed by the expression to be cast:

(type) expression

The cast causes the result of the expression to be converted to the specified type. As an example, we can use a cast to force the floating point number 17.7 to be treated as *int*.

```
a = ( int ) 17.7  *  2 ;
```

This casts 17.7 (and only 17.7, not the entire expression) to an *int* by truncation. The result of this assignment is that *a* receives the value 34. Here 17.7 is cast to an *int*, 17, rather than the entire product, since the precedence of the cast operator is higher than that of most other operators.

We normally cast a variable to ensure that the arithmetic is carried out with the type on the left-hand side. We can round rather (rather than truncate) a variable to an integer by adding 0.5 to it and then casting to an *int*.

```
a = ( int )  ( val + 0.5 ) ;
```

If *val* is 37.8, adding 0.5 to it yields 38.3; casting this to an *int* truncates the result to 38, the value that is then assigned to *a*. Of course, the variable or expression being cast is not changed; a cast simply returns a value of the cast type. Because of the high precedence of the cast operator, we had to parenthesize the expression to be cast, as we did above. Contrast this to the effect of writing

```
a = ( int ) val  +  0.5 ;
```

Another typical use of a cast is in to force division to return a real number when both operands are *int*s. For example, a program to average a series of integers might accumulate a total in an integer variable *sum*, and a count of the number of values read in the integer *n*. We compute the average with:

```
ave = ( double ) sum / n ;
```

Casting *sum* to a *double* causes the division to be carried out as floating point division. Without the cast, truncated integer division is performed, since both *sum* and *n* are integers.

Ex : If the following Declarations and Assignments are made ,

```
int  i , j , m , n , l ;
float  f , g ;
char C ;

i = j = l = 2 ;
m = n = 5 ;
f = 12 ;  g = 4 ;  C = 'X' ;
```

Calculate the following expressions.

(1) 1 + 12 * m
(2) m % j
(3) n % j
(4) m / j * l
(5) f + 10 * 5 + g
(6) (l++) * n
(7) l++ * n
(8) -12l * (g - l)
(9) 2 + C
(10) (m++) - (m++)
(11) n = --n
(12) m = n = j--

Ex : If the following Declarations and Assignments are made ,

```
char C = 'w' ;
int  i = 1 , j = 4 , k = 7 ;
double x = 7e + 33 , y = 0.001 ;
```

Evaluate the following expressions.

(1) 'A' + 1 < C
(2) -i -5 * j >= k + 1
(3) 3 < j < 5
(4) x - 3.333 <= x + y
(5) x < x + y

Ex : If the following Declarations and Assignments are made ,

```
char C = 'B' ;
int  i = 3 , j = 3 , k = 3 ;
double x = 0.0 , y = 2.3  ;
```

Evaluate the following expressions.

(1) i && j && k
(2) x || i && j - 3
(3) i < j && x < y
(4) i < j || x < y
(5) 'A' <= C && C <= 'Z'
(6) C - 1 == 'A' || C + 1 == 'Z'