Introduction:

C++ is an intermediate level programming language. Unlike C, C++ follows object oriented paradigm. A program in C++ is made up of letters , words , sentences , and constructs just like english language. In this unit we will discuss these elements of C++ language along with different operators applicable over them.

Token

The smallest individual units in a program are known as tokens. C++ has following tokens:

- Keywords
- Identifiers
- Constants
- Operators

A C++ program is written using these tokens, white spaces , and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

Keywords:

The keywords in C++ are the reserved words. They have fixed meaning and these meanings cannot be changed. All the keywords in C are also the keywords in C++. C++ has extra more keywords.

Keywords from C

auto	double	int	struct
break	else	long	switch

case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Some additional keywords in C++ are

catch	private	try	class	protected
virtual	delete	public	friend	template
inline	this	new	throw	

Identifiers

Identifiers refers to the name of variable , functions , arrays, classes, pointers etc that are created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++.

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot with a digit.
- Uppercase and lowercase letters are not distinct.
- A reserved keyword cannot be used as a variable name.
- Must not contain white spaces.

A major difference between C and C++ is the limit on the length of a name. While C recognizes only the first 32 characters in a name, C++ places no limit on its length and therefore all the characters in a name are significant. However some operating system may impose a restriction on the length of variable name.

Constant

A constant is any expression that has a fixed value. Their value do not change during the execution of a program. Constant can be divided into Integer Numbers , Floating-Point Numbers , Characters and Strings.

Integer Numbers

1889 , 709 , -275

Whenever we are writing 1889 in our program we will be referring to the value 1889. In addition to the decimal numbers, C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16).

- If we want to express as octal number we precede with a 0 character (Zero Character).
- If we want to express as hexadecimal number we have to precede it with the characters 0x (Zero, x).

Example: All of them represent the same number 75.

```
75       //decimal
0113     //octal
0x4b     //hexadecimal
```

Floating Point Numbers

```
3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0
```

Here, the first number is PI, second one is the number of Avogadro, the third is the electric charge of an electron, and the last one is the number 3 expressed as floating point number.

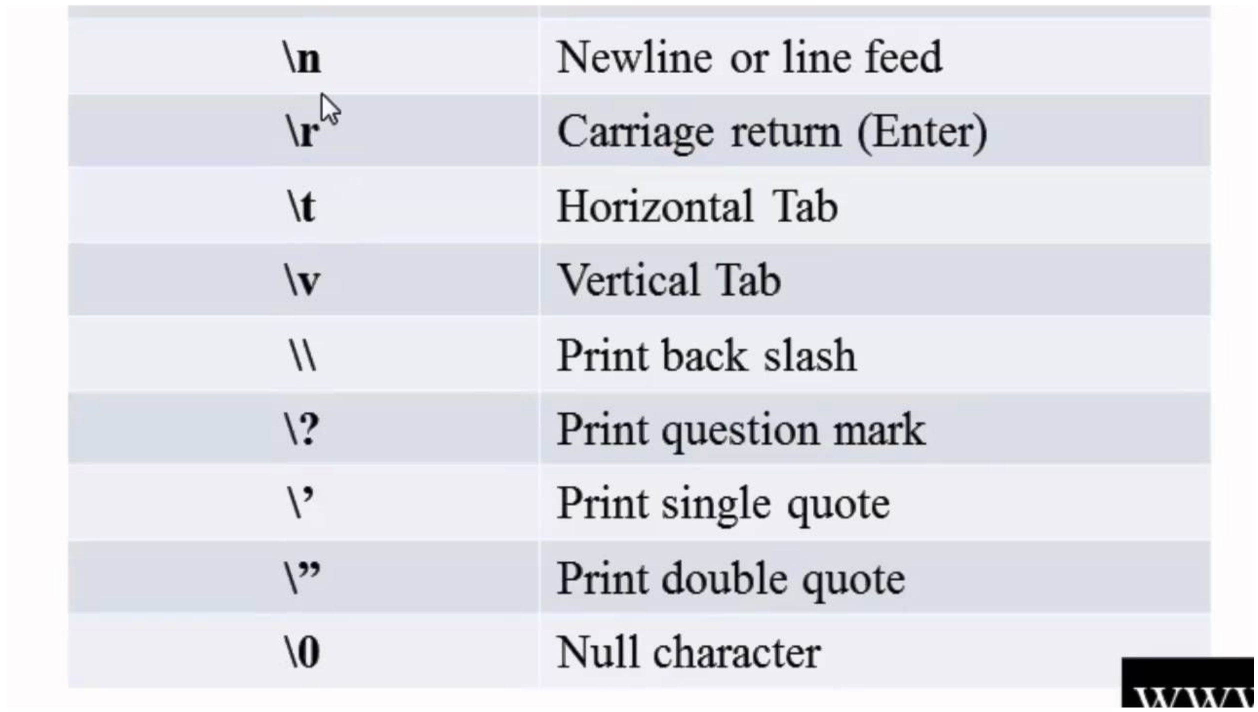
Characters and Strings

```
‘X’
‘Z’
“Hello World”
“How are you ?”
```

The first two are characters and are enclosed by single quote (‘) while the last two are strings and are enclosed by double quote (“).

Escape Characters (Escape Sequence)

An escape sequence is a sequence of characters that does not represent itself ,but is translated into a sequence of characters that may be difficult or impossible to represent directly. Here is the list of such escape codes:



\n	Newline or line feed
\r	Carriage return (Enter)
\t	Horizontal Tab
\v	Vertical Tab
\\	Print back slash
\?	Print question mark
\’	Print single quote
\”	Print double quote
\0	Null character

Operators:

Operators are symbols or combinations of symbols that directs the computer to perform some operation upon operands. C++ includes large number of operators, which fall into several different categories.

- Arithmetic Operators
- Assignment Operators
- Unary Operators
- Comparison Operators
- Shift Operators
- Bit-wise Operators

- Logical Operators
- Conditional Operators

Arithmetic Operators

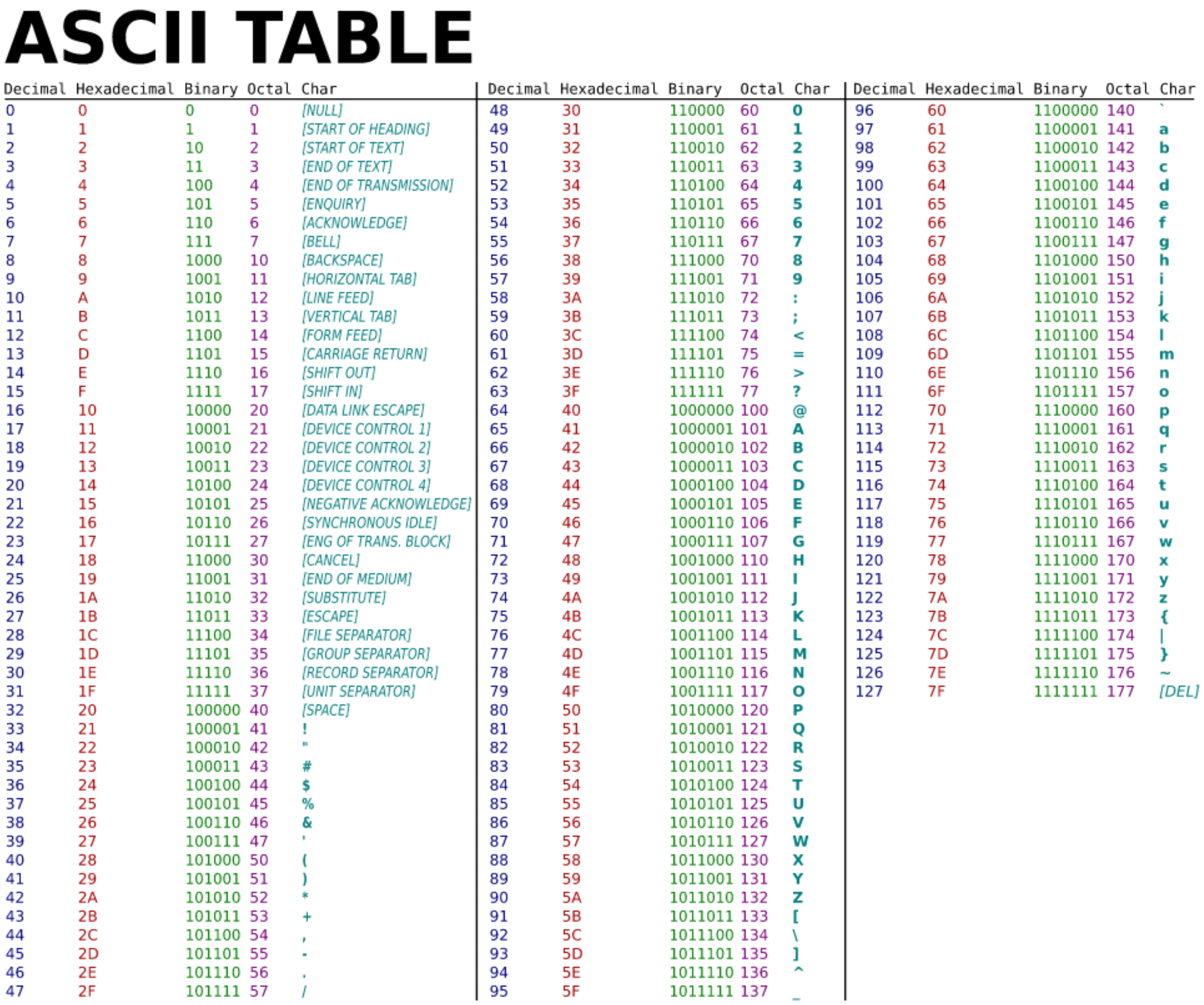
There are five arithmetic Operators in C++. They are:

Suppose variable a holds 10 and b holds 20

Operator	Description	Expression	Value
+	Adds two operands	a+b	30
-	Subtracts second operand from the first	a-b	-10
*	Multiplies both operands	a*b	200
/	Divides numerator by de-numerator	b/a	2
%	Modulus Operator (remainder after an integer division)	b%a	0

Notes:

- There is no exponential operator in C++. However there is a library function (pow) to carry out exponential.
- The operands can be integer , floating-point or characters (remember that character constants represent integer value determined by computer's character set)
- The remainder operator and division operator requires second operand to be nonzero.



ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Suppose X1 and X2 are character-type variables that represent the character M and U respectively. Some arithmetic expressions that make use of these variables are shown below.

X1 + X2 =162
X1+ X2+ ‘5’ = 215

Note that M is encoded as 77 , U is encoded as 85, and 5 is encoded as 53 in ASCII character set.

Assignment Operator:

Assignment operator assigns the value of right operator or expression to the variable in left side. There are many variations of assignment operator as described below;

Operator	Description	Example
=	Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Adds the operands and assigns the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A

/=	Divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

Unary Operators:

A unary operator in C++, is an operator that takes a single operand in an expression or a statement. There are basically two unary operators in C++. They are:

Operator	Description	Example	Explanation
++	Increases the value of the operand by one.	a++	Equivalent to a=a+1
--	Decreases the value of the operand by one.	a--	Equivalent to a=a-1

The increment operator ++, can be used in two ways:

- As a prefix
  In prefix, the operator precedes the variable i.e. ++var . In this form the value of the variable is first incremented and then used in the expression as illustrated below;

  var1=20;   var2=++var1;

This code is equivalent to the following set of codes:

```
var1=20;      var1=var1+1;         var2=var1;
```

At the end, both variables var1 and var2 store value 21.

- As a postfix

  Likewise, in postfix, the operator follows the variable i.e. var++. In this form, the value of variable is used in the expression and then incremented as illustrated below;

```
var1=20;      var2=var1++;
```

The equivalent of this code is:

```
var1=20;      var2=var1;   var1=var1+1;
```

At the end, var1 has the value 21 while var2 remainds set to 20.

Comparison Operator

Comparison operators evaluate to true or false. They are also called relational operators.

Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of	(A >= B) is not true.

	right operand, if yes then condition becomes true.	
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Shift Operators

Data is stored internally in binary format. A bit can have a value of one or zero. Eight bits form a byte. Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. We can use them only on integer data type and not on the char , float , or double data types.

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bir at the left.	a=10>>3	The result of this is 10 divided by 2^3.
<<	Shifts bits to the left, filling zeros at the right.	a=10<<3	The result if this is 10 multiplied by 2^3.

If the int data type occupies four byte in the memory, the rightmost eight bits of the number are represented in binary as “00001010”

When we do right shift by 3, the result is “00000001” which is equivalent to 1 and when we do left shift by 3, the result is “01010000” which is equivalent to 80.

Bit-wise operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

Binary And Operator:

A&B = 0000 1100

Binary Or Operator:

A|B = 0011 1101

Binary XOR Operator:

A^B = 0011 0001

Binary Ones Complement Operator:

~A = 1100 0011

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a	!(A && B) is true.

	condition is true, then Logical NOT operator will make false.	

Miscellaneous Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	sizeof **sizeof operator** returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	Condition ? X : Y **Conditional operator (?)**. If Condition is true then it returns value of X otherwise returns value of Y. Example: result: (marks>50 ? "pass": "fail") ; In above example, if the marks obtained is more than 50, result will be pass else fail.

3	, **Comma operator** causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	. (dot) and -> (arrow) **Member operators** are used to reference individual members of classes, structures, and unions.
5	Cast **Casting operators** convert one data type to another. For example, int(2.2000) would return 2.
6	& **Pointer operator &** returns the address of a variable. For example &a; will give actual address of the variable.
7	* **Pointer operator *** is pointer to a variable. For example *var; will pointer to a variable var.

Operator Precedence:

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right

Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Structure of C++ Program:

```
#include<iostream.h>
#include<conio.h>

class Test
{
        private:
                int a, b;
        public:
                void getData(int p, int q)
                {
                        a=p;
                        b=q;
                }
                void display()
                {
                        cout<< “Addition of two numbers, a and b is”<< a+b ;
                }
} ;

int main()
{
        clrscr();
        Test obj;
        obj.getData(10,20);
        obj.display();
        getch();
```

```
	return 0;
}
```

Output:

```
Addition of two numbers, a and b is 30
```

Insertion and Extraction Operator

In C++, input and output (I/O) operators are used to take input and display output. The operator used for taking the input is known as the **extraction** or get **from operator** (>>), while the operator used for displaying the output is known as the **insertion** or **put to operator** (<<).

Extraction Operator (>>)

“>>” is extraction operator because it extracts data enter by user from console or input screen to some storage location identified by variable.

1. **int** a;
2. cin >> a;

now when you run this, you will prompt console waiting for input. As you enter say ‘5’, number 5 value is “extracted” from console and stored at the address refer by that variable say “a”.

Insertion Operator (<<)

“<<” is insertion operator because it inserts data from some variable or stream to the console.

1. **int** b = 7;
2. cout << b;

now when you run this, data from variable “b” is inserted to the console. and next line insert data to console as stream.

Example to demonstrate external and internal operator:

```
#include<iostream.h>
#include<conio.h>

class rectangle
{
	public:
		int calculate(int l, int b)
		{
			return l*b;
		}
};

int main()
{
	clrscr();
	rectangle rect;
	cout<<"Area of rectangle:";
	cout<<rect.calculate(5,4);
	getch();
	return 0;
}
```

Output

```
Area of rectangle:20
```

Cascading of Input/Output Operators

The cascading of the input and output operators refers to the consecutive occurrence of input or output operators in a single statement.

To understand the concept of cascading of the input/output operator, consider following example.

```
#include<iostream.h>
#include<conio.h>

class rectangle
{
        private:
                int length, breadth, result;
        public:
                void getData()
                {
                        cout<<"Enter length and breadth:\n";
                        cin>>length>>breadth;           //cascading of input operator
                }

                int  calculate()
                {
                        return length*breadth;
                }
};

int main()
{
        clrscr();
        rectangle rect;
        rect.getData();
        cout<<"Area of rectangle is:"<<rect.calculate();        //cascading of output operator
        getch();
        return 0;
}
```

<u>Output</u>

```
Enter length and breadth:
9
5
Area of rectangle is: 45
```

Type Conversion in C++

Converting an expression of a given type into another type is known as type conversion or type-casting. There are two types of type conversion:

- Automatic conversion (Implicit Type Conversion)
- Type casting (Explicit Type Conversion)

Implicit Type Conversion

It is also known as ‘automatic type conversion’.It is done by the compiler on its own, without any external trigger from the user.Generally it takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> int -> long -> float -> double -> long double

Example:

```
#include <iostream.h>
#include<conio.h>

int main()
{
   int x = 10; // integer x
   char y = 'a'; // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.5;

   cout << "x = " << x << "\n" << "y = " << y << "\n"<< "z = " << z << "\n";
   getch();
   return 0;
}
```

Output:

```
x = 107
y = a
z = 108.5
```

Explicit Type Conversion:

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

- Converting by assignment:
- Conversion using Cast operator:

Converting by assignment:

This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:`(type) expression`

where *type* indicates the data type to which the final result is converted.

Example:

```
#include <iostream.h>
#include<conio.h>

int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   cout << "Sum = " << sum;
   getch();
   return 0;
}
```

Output:

Sum=2

Conversion using Cast operator

A Cast operator is an unary operator which forces one data type to be converted into another data type.
C++ supports four types of casting:

- Static cast
- Dynamic cast
- Dynamic cast
- Reinterpret cast

Example:

```
#include <iostream.h>
#include <conio.h>

int main()
{
   float f = 3.5;

   // using cast operator
   int b = static_cast<int>(f);
   getch();
   cout << b;
}
```

Output:

```
3
```

Arrays in C++

In programming, one of the frequently arising problem is to handle numerous data of same type.

Consider this situation, you are taking a survey of 100 people and you have to store their age. To solve this problem in C++, you can create an integer array having 100 elements.

An array is a collection of data that holds fixed number of values of same type.

For example:

```
int age[100];
```

Here, the `age` array can hold maximum of 100 elements of integer type.

The size and type of arrays cannot be changed after its declaration.

How to declare an array in C++?

```
dataType arrayName[arraySize];
```

For example,

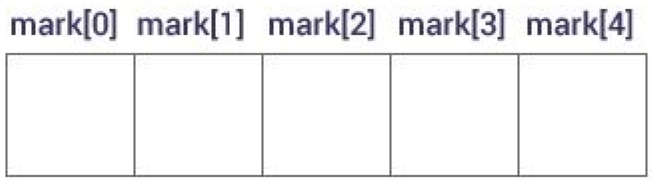
```
float mark[5];
```

Here, we declared an array, `mark`, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and How to access them?

You can access elements of an array by using indices.

Suppose you declared an array `mark` as above. The first element is `mark[0]`, second element is `mark[1]` and so on.



Few key notes:

- Arrays have 0 as the first index not 1. In this example, mark[0] is the first element.
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark[4] is the last element.
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of float is 4 bytes.

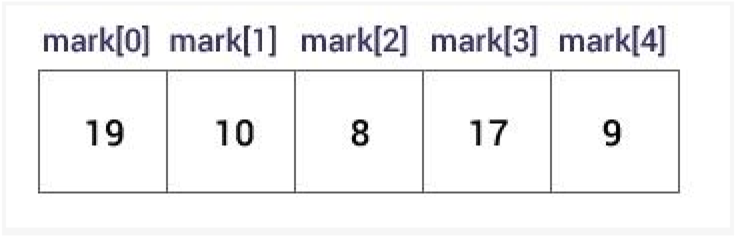
How to initialize an array in C++ programming?

It's possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

Another method to initialize array during declaration:

int mark[] = {19, 10, 8, 17, 9};



Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

Example:

C++ program to store and calculate the sum of 5 numbers entered by the user using arrays.

Using Class and Object:

```
#include <iostream.h>
#include<conio.h>

class array
{
        private:
                int num[5],sum=0 ;
        public:
                void getData()
                {
                        for(int i=0; i<5;i++)
                        {
                                cout<<"Enter "<<i+1<< " number:\n";
                                cin>>num[i];
                        }
                }

                void printSum()
                {
                        for(int i=0;i<5;i++)
                        {
                                sum=sum+num[i];
                        }
                        cout<<"Total Sum="<<sum;
                }


};

int main()
{
        clrscr();
        array arr;
        arr.getData();
        arr.printSum();
        getch();
        return 0;
}
```

Output:

```
Enter 1 number:
1
Enter 2 number:
2
Enter 3 number:
3
Enter 4 number:
4
Enter 5 number:
5
Total Sum=15
```

Without using Class and Object:

```
#include <iostream>
using namespace std;


int main()
{
                int num[5],sum=0 ;
                for(int i=0; i<5;i++)
                {
                        cout<<"Enter "<<i+1<< " number:\n";
                        cin>>num[i];
                }
                for(int i=0;i<5;i++)
                {
                        sum=sum+num[i];
                }
```

```
                    cout<<"Total Sum="<<sum;
}
```

<u>Output:</u>

Same as above.

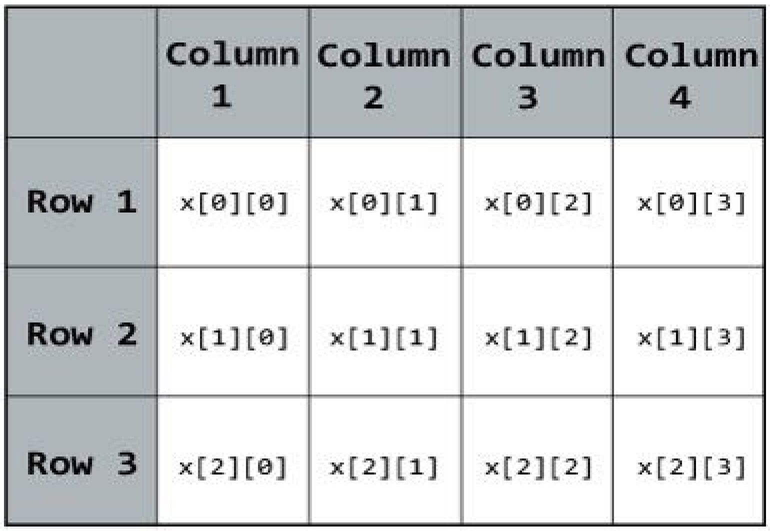
Multidimensional Array

In C++, you can create an array of an array known as multi-dimensional array. For example:

```
int x[3][4];
```

Here, x is a two dimensional array. It can hold a maximum of 12 elements.

You can think this array as table with 3 rows and each row has 4 columns as shown below.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Three dimensional array also works in a similar way. For example:

```
float x[2][4][3];
```

This array x can hold a maximum of 24 elements. You can think this example as: Each of the 2 elements can hold 4 elements, which makes 8 elements and each of those 8 elements can hold 3 elements. Hence, total number of elements this array can hold is 24.

Multidimensional Array Initialisation

You can initialise a multidimensional array in more than one way.

Initialisation of two dimensional array

```
int test[2][3] = {2, 4, -5, 9, 0, 9};
```

Better way to initialise this array with same array elements as above.

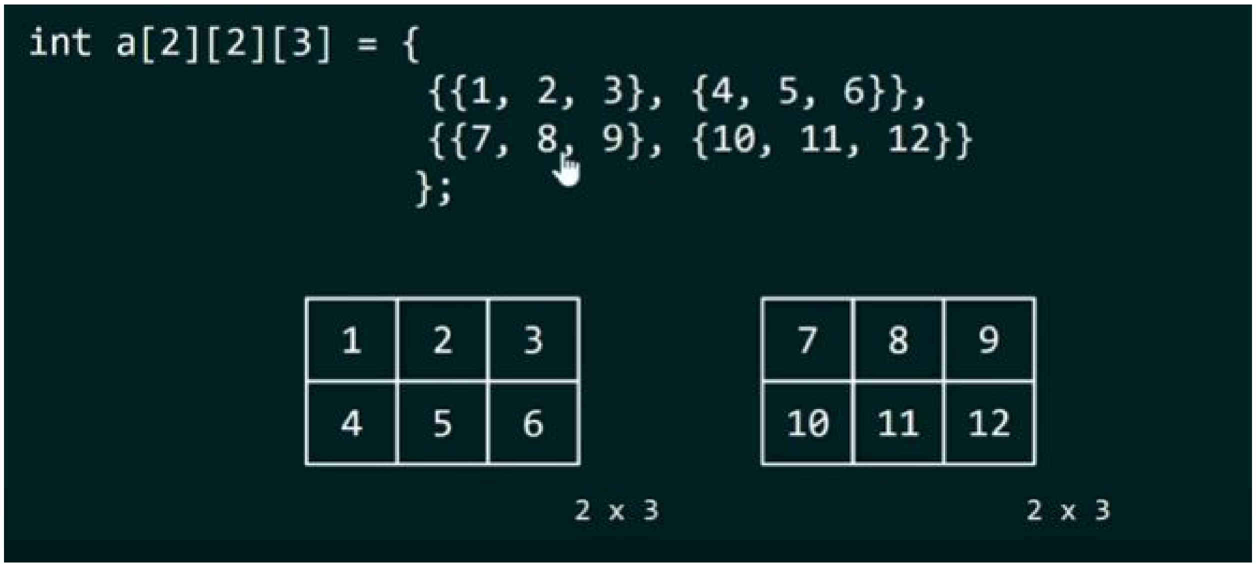
```
int  test[2][3] = { {2, 4, 5}, {9, 0 0}};
```

Initialisation of three dimensional array

```
int test[2][3][4] = {3, 4, 2, 3, 0, -3, 9, 11, 23, 12, 23,
                     2, 13, 4, 56, 3, 5, 9, 3, 5, 5, 1, 4, 9};
```

Better way to initialise this array with same elements as above.

```
int test[2][3][4] = {
                        { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                        { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
                    };
```



```
int a[2][2][3] = {
                  {{1, 2, 3}, {4, 5, 6}},
                  {{7, 8, 9}, {10, 11, 12}}
                 };
```

Example:

C++ Program to display all elements of an initialised two dimensional array.

```
#include <iostream.h>
#include<conio.h>
int main()
{
   int test[3][2] =
   {
      {2, -5},
      {4, 0},
      {9, 1}
   };

   for(int i = 0; i < 3; ++i)
   {
      for(int j = 0; j < 2; ++j)
      {
         cout<< "test[" << i << "][" << j << "] = " << test[i][j] << "\n";
      }
   }
  getch();
```

```
   return 0;
}
```

Output:

```
test[0][0] = 2
test[0][1] = -5
test[1][0] = 4
test[1][1] = 0
test[2][0] = 9
test[2][1] = 1
```

Example:

C++ Program to store temperature of two different cities for a week and display it.

```
#include <iostream.h>
#include<conio.h>

const int CITY = 2;
const int WEEK = 7;
int main()
{
   int temperature[CITY][WEEK];
   cout << "Enter all temperature for a week of first city and then second city. \n";
   // Inserting the values into the temperature array
   for (int i = 0; i < CITY; ++i)
   {
      for(int j = 0; j < WEEK; ++j)
      {
         cout << "City " << i + 1 << ", Day " << j + 1 << " : ";
         cin >> temperature[i][j];
      }
   }
   cout << "\n\n Displaying Values:\n";
```

```
    // Accessing the values from the temperature array
    for (int i = 0; i < CITY; ++i)
    {
        for(int j = 0; j < WEEK; ++j)
        {
            cout << "City " << i + 1 << ", Day " << j + 1 << " = " << temperature[i][j] << endl;
        }
    }
 getch();
 return 0;
}
```

<u>Output:</u>

```
Enter all temperature for a week of first city and then second city.
City 1, Day 1 : 32
City 1, Day 2 : 33
City 1, Day 3 : 32
City 1, Day 4 : 34
City 1, Day 5 : 35
City 1, Day 6 : 36
City 1, Day 7 : 38
City 2, Day 1 : 23
City 2, Day 2 : 24
City 2, Day 3 : 26
City 2, Day 4 : 22
City 2, Day 5 : 29
City 2, Day 6 : 27
City 2, Day 7 : 23


Displaying Values:
City 1, Day 1 = 32
City 1, Day 2 = 33
City 1, Day 3 = 32
City 1, Day 4 = 34
City 1, Day 5 = 35
City 1, Day 6 = 36
City 1, Day 7 = 38
City 2, Day 1 = 23
City 2, Day 2 = 24
City 2, Day 3 = 26
City 2, Day 4 = 22
City 2, Day 5 = 29
City 2, Day 6 = 27
City 2, Day 7 = 23
```

Example:

C++ Program to Store value entered by user in three dimensional array and display it.

```
#include <iostream.h>
#include <conio.h>

int main()
{
   // This array can store upto 12 elements (2x3x2)
   int test[2][3][2];

   cout << "Enter 12 values: \n";

   // Inserting the values into the test array
   // using 3 nested for loops.
   for(int i = 0; i < 2; ++i)
   {
      for (int j = 0; j < 3; ++j)
      {
         for(int k = 0; k < 2; ++k )
         {
            cin >> test[i][j][k];
         }
      }
   }
```

```
    cout<<"\nDisplaying Value stored:"<<endl;

    // Displaying the values with proper index.
    for(int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k)
            {
                cout << "test[" << i << "][" << j << "][" << k << "] = " << test[i][j][k] << endl;
            }
        }
    }

    return 0;
}
```

<u>Output:</u>

```
Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

Displaying Value stored:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
```

```
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

Pointers in C++

Pointers variables are variables that points to a specific address in the memory pointed by another variable.

How to declare a pointer?

```
int *p;
```

OR,

```
int* p;
```

The statement above defines a pointer variable p. It holds the memory address

Reference operator (&) and Dereference operator (*)

Reference operator (&) gives the address of a variable. Likewise,to get the value stored in the memory address, we use the dereference operator (*).

<u>For example</u>:

If a number variable is stored in the memory address **0x123**, and it contains a value **5**.

The **reference (&)** operator gives the value **0x123**, while the **dereference (*)** operator gives the value **5**.

Note: The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

Example:

```
#include <iostream.h>
#include<conio.h>

int main()
{
        int number=5;
        int *ptr;
        ptr=&number;
        cout<<"Address="<<ptr<<"\n";
        cout<<"Address="<<&number<<"\n";
        cout<<"Value="<<*ptr;
        getch();
        return 0;
}
```

Output

```
Address=0x6ffe34
Address=0x6ffe34
Value=5
```

C++ pointers and Arrays

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:

```
int* ptr;

int a[5];

ptr=&a[0]         //&a[0] is the address of first element of a[5]

ptr+1 = &a[1];   // &a[1] is the address of second element of a[5].

ptr+2 = &a[2];  // &a[2] is the address of third element of a[5].

ptr +3 = &a[3];  //&a[2] is the address of fourth element of a[5].

ptr+4 = &a[4];  //&a[4] is the address of fifth element of a[5].
```

To access value for each , we use dereference operator (*)

Pointers to Arrays

Example:

C++ program to insert array data and display address and value for each array element.

using class and object

```
#include<iostream.h>
#include<conio.h>


class pointer{
        private:
                int a[5],i;
                int *ptr;

        public:

                void getData(){
                        for(i=0;i<5;i++){
                                cout<<"Enter "<<i+1<<" value:\n";
                                cin>>a[i];
                        }
                }

                void displayAddress(){
                        ptr=a;
                        for(i=0;i<5;i++){
                                cout<<"address "<<i+1<<"="<<ptr+i<<"\n";
                        }
                }

                void displayValue(){
                        ptr=a;
                        for(i=0;i<5;i++){
                                        cout<<"Value "<<i+1<<"="<<*(ptr+i)<<"\n";
                        }
                }
};

int main(){
        pointer point;
        point.getData();
        point.displayAddress();
        point.displayValue();
        getch();
        return 0;
}
```

<u>Output</u>

```
Enter 1 value:
1
Enter 2 value:
```

```
3
Enter 3 value:
45
Enter 4 value:
55
Enter 5 value:
44
address 1=0x6ffe10
address 2=0x6ffe14
address 3=0x6ffe18
address 4=0x6ffe1c
address 5=0x6ffe20
Value 1=1
Value 2=3
Value 3=45
Value 4=55
Value 5=44
Total Sum=148
```

<u>Without using class and object:</u>

```
#include<iostream>
using namespace std;


int main(){
            int a[5],i,*ptr, sum=0;

            for(i=0;i<5;i++)
```

```
		{
			cout<<"Enter "<<i+1<<" value:\n";
			cin>>a[i];
		}
		ptr=a;
		for(i=0;i<5;i++)
		{
			cout<<"address "<<i+1<<"="<<ptr+i<<"\n";
		}
		for(i=0;i<5;i++)
		{
			cout<<"Value "<<i+1<<"="<<*(ptr+i)<<"\n";
			sum=sum+*(ptr+i);

		}
		cout<<"Total Sum="<<sum;

	return 0;
}
```

<u>Output:</u>

Same as above

Arrays of Pointers

Following is the declaration of an array of pointers to an integer −

```
int *ptr[MAX];
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Example that makes use of three integers which will be stored in an array of pointers

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; // assign the address of integer.
   }

   for (int i = 0; i < MAX; i++) {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }

   return 0;
}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Manipulators

Manipulators are the functions which are used along with insertion operator (<<) and extraction operator (>>) in order to manipulate (format) the output. There are several manipulators in C++ . Some of the commonly used manipulators are:

- endl
- setw
- setprecision

Manipulator	**Declaration in**
endl	iostream.h
setw	iomanip.h
setprecision	iomanip.h
setf	iomanip.h
setfill	iomanip.h

endl

This manipulator has the same functionality as '\n' (newline character). When writing output in C++,you can use either endl or '\n' to produce a newline, but each has a different effect.

- endl sends a newline character '\n' and flushes the output buffer.

- '\n' sends the newline character, but does not flush the output buffer.

Example:

```
#include<iostream>

using namespace std;

int main() {

   cout << "Hello" << endl << "World!";

  return 0;

}
```

Output

```
Hello

World!
```

setw

This manipulator causes the output stream to be printed within a field of n character wide, when n is the argument to setw(n).

Example:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
   cout << "no setw:" << 42 << '\n'
              << "setw(6):" << setw(6) << 42 << '\n';
```

```
    return 0;
}
```

Output

```
no setw:42
setw(6):    42
```

setprecision

This manipulator cause the floating point to be printed with n digits of precision , where n is the argument of setprecision(n).

Example:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
   const long double pi = 3.141592653589793239;
   cout << "default precision (6): " << pi << '\n'
             << "setprecision(10): " << setprecision(10) << pi << '\n';
   return 0;
}
```

Output

```
default precision (6): 3.14159
std::setprecision(10): 3.141592654
```

setfill

It is used to sets c as the stream's *fill character*, c is the character argument to setfill(‘c’).

Example:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main () {
  cout << setfill ('x') << setw (10);
  cout << 77 << endl;
  return 0;
}
```

Output

```
xxxxxxxx77
```

Enumeration

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

```
enum season { spring, summer, autumn, winter };
```

Here, the name of the enumeration is season.

And, spring, summer and winter are values of type season.

By default, spring is 0, summer is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

```
enum season
{   spring = 0,
    summer = 4,
    autumn = 8,
    winter = 12
};
```

Example:

```
#include <iostream>
using namespace std;

enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main()
{
   week today;
   today = Wednesday;
   cout << "Day " << today+1;
   return 0;
}
```

Output

Day 4

When to use enum ?

When we need a **predefined** list of values we should use an enum. we should always use enums when a variable (especially a method parameter) can only take one out of a small set of possible values.

Dynamic Memory allocation in C++

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static .For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

How is memory allocated/deallocated in C++?

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

new Operator

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Here is the syntax of new operator in C++ language,

- To allocate memory of any data type, the syntax is:

  pointer_variable = new datatype;

  Example:

  ```
  int *p = NULL;
  p = new int;
  ```

  or

  ```
  int *p = new int;
  ```

- Here is the syntax to initialize the memory,

  pointer_variable = new datatype(value);

  Example:

  ```
  int *p = new int(25);
  float *q = new float(75.25);
  ```

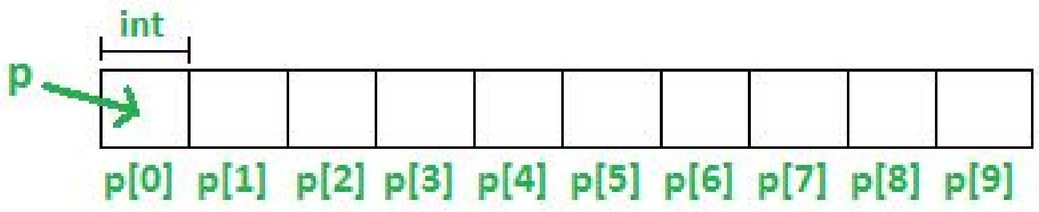
- Here is the syntax to allocate a block of memory,

  pointer_variable = new datatype[size];

  Example:

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer .Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
        cout << "Memory allocation failed\n";
      }
```

Example 1: An example of new operator in C++ language,

```
#include <iostream>
using namespace std;
int main () {

   int *ptr1  = NULL;
   ptr1 = new int;
   float *ptr2 = new float(223.324);
   *ptr1 = 28;
   cout << "Value of pointer variable 1 : " << *ptr1 << endl;
   cout << "Value of pointer variable 2 : " << *ptr2 << endl;

   return 0;
}
```

Output:

```
Value of pointer variable 1: 28
value of pointer variable 2:223.324
```

Example 2: An example of new operator in C++ language,

```
#include <iostream>
using namespace std;


int main () {
   int n;
   cout<<"Enter the size of n:";
   cin>>n;
   int *ptr = new (nothrow) int[n];
```

```
  if (!ptr)
   {
      cout << "Allocation of memory failed\n";
    }
  else {
    for (int i = 0; i < n; i++)
      ptr[i] = i+1;     //   *(ptr+i) = i+1;
    cout << "Value stored are ";
    for (int i = 0; i < n; i++) {
             cout << ptr[i] << " ";   //cout << *(ptr+i) << " ";

             }

  }
  return 0;
}
```

Output

```
Enter size of n:3

Value stored are: 1 2 3
```

delete operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
 delete p;

 delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory pointed by pointer-variable

delete[] pointer-variable;
```

Example:

```
  // It will free the entire array pointed by p.

  delete[] p;
```

Example: program to illustrate deallocation of memory using delete

```
#include <iostream.h>
using namespace std;

int main()
{
    // Creating int pointer
    int* ptr1 = new int;

    // Initializing pointer with value 20
    int* ptr2 = new int(20);

    cout << "Value of ptr1 = " << *ptr1 << "\n";
    cout << "Value of ptr2 = " << *ptr2 << "\n";

    delete ptr1; // Destroying ptr1
    delete ptr2; // Detroying ptr2

    return 0;
}
```

Output:

```
Value of ptr1 = 0
Value of ptr2 = 20
```

const Keyword in C++

Constant is something that doesn't change. In C language and C++ we use the keyword const to make program elements constant. const keyword can be used in many contexts in a C++ program. It can be used with:

1. Variables

```
const int i = 10;
```

2. Pointers

```
const int* u;
```

3. Function arguments and return types

```
void assign(const int i)
{
  a=i;   //works fine
   i++;    // error
}

const int g()
{
   return 1;
}
```

4. Class Data members

```
class Test
{
   const int i;

};
```

5. Class Member functions

```
// constant function

int display() const
  {

     cout << "Falcon has left the Base";
   }
```

6. Objects

```
const Test r;
```