Hexadecimal Integer: In a programming language like C++, it is possible to represent an integer constant in different form. Generally an integer value is represented as a **Decimal** integer. A decimal integer value consists of any 10 digits (0-9). Integers `29`, `73545`, `8545`, `-34`, `-428954` and `3945` are example of **Decimal** integer values. In C++ it is also possible to represent an integer as a **Hexadecimal** integer. A **Hexadecimal** integer value consists of 16 digits (0-9, A-F). Integers `2A`, `4B6C`, `ABCD` and `F16` are example of **Hexadecimal** integer constant. In a C++ program **Hexadecimal** integer constant is prefixed by `0x`. For example `1B4C` is a **Hexadecimal** integer constant but in C++ program it will be represented as `0x1B4C`. An example of decimal integer and Hexadecimal integer is given below:

```
#include<iostream.h>
void main()
{
   int hi=0x1B4C;
   int di=174911;
   cout<<"Dec="<<hi<<","<<"Dec="<<di<<endl;
   cout.setf(ios::hex, ios::basefield);
   cout<<"Hex="<<hi<<","<<"Hex="<<di<<endl;
}
```

Variable **hi** is assigned a **Hexadecimal** integer constant while variable **di** is assigned **Decimal** integer constant. First 2 outputs display values stored in variables **hi** and **di** as **Decimal** integer. Last 2 outputs display values stored in variables **hi** and **di** as **Hexadecimal** integer.

Running of the program produces following output:

```
Dec=6988,Dec=174911
Hex=1B4C,Hex=2AB3F
```

Pointer

A variable in C++ has three characteristics – data type of the variable, value stored in the variable and the address of variable. So far in our programming examples we have only used the first two characteristics, that is, data type of the variable and the value stored in the variable. Address of a variable represents the location of the variable in the computer's main storage (RAM). The concept of address of a variable is similar to address of house / flat / villa / shop in a city / town / village. To get an address of a variable we use address operator (&) before a variable name. In C++ address of a variable is also known as **Pointer**. Pointer (address) is displayed as a Hexadecimal integer. An pointer will display address except for a pointer to a character. Pointer to a character will be discussed later. An example of pointer and address is given below:

```
#include<iostream.h>
void main()
{
   int a=2014;
   double b=89.7;
   cout<<"a="<<a<<" , b="<<b<<endl;
   cout<<"&a="<<&a<<" , &b="<<&b<<endl;
}
```

Variable **a** is assigned a value **20** and variable **b** is assigned a value **88.5**. First two outputs display value stored in the variable **a** and **b**. Last two outputs display address of the variables **a** and **b**. Addresses of the variables are displayed as **Hexadecimal** integers.

Running of the program produces following output:

```
a=2014 , b=89.7
&a=0x0012ff88 , &b=0x0012ff80
```

Diagrammatic representation of variables created (in the above program) their respective addresses:

a	b	→	Variable Names
2014	89.7	→	Values Stored
0012ff88	0012ff80	→	Address of the Variables

Pointer Variable

To store an address of a variable we need to create a special type of variable called **Pointer** variable. Creating a **Pointer** variable is similar to creating a variable of fundamental data type or array type.

Rule:
```
DataType* PointerVarName;
DataType *PointerVarName;
DataType *PointerVarName1, *PointerVarName1, … ;
```

`DataType` could be fundamental data type or derived data type like structure type or class type. Operator star (*) is needed between `DataType` and `PointerVarName`. Operator star (*) implies that the variable that is being created is **Pointer** type. When using the **Pointer** variable in the program, operator star (*) is never used, that is, in the program only `PointerVarName` will be used.

Usage:
```
int* ip1;
int *ip2, *ip3;

char* cp1;
char *cp2, *cp3;

double* dp1;
double *dp2, *dp3;
```

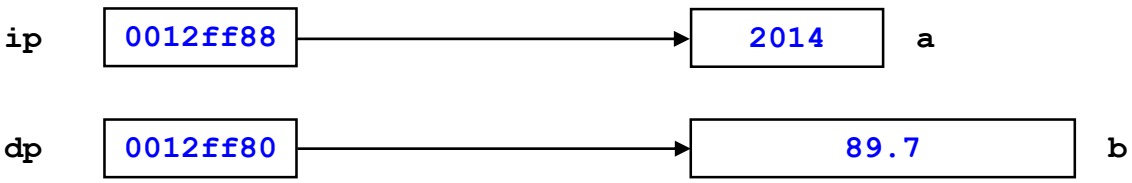
- Statement `int *ip;` creates an integer pointer (pointer to an integer). An integer pointer can store an address of an integer variable
- Statement `char *cp;` creates a character pointer (pointer to a character). A character pointer can store an address of a character variable
- Statement `double *dp;` creates a double pointer (pointer to a double). A double pointer can store an address of a double type
- A pointer variable is allocated 4 bytes of memory

Example:
```
void main()
{
   int a=2014, *ip;
   double b=89.7, *dp;
   ip=&a;
   dp=&b;
   cout<<"a="<<a<<" , b="<<b<<endl;
   cout<<"ip="<<ip;
   cout<<" , dp="<<dp<<endl;
}
```

Variable **a=20** and **ip** (pointer to integer) is created. Variable **b=88.5** and **dp** (pointer to double) is created. Pointer **ip** is assigned address of **a** and **dp** is assigned address of **b**. Creation of pointer variable and assigning an address to it can be combined as one single statement. For example:
```
int *ip=&a;
double *dp=&b;
```

Running of the program produces following output:
```
a=2014 , b=89.7
ip=0x0012ff88 , dp=0x0012ff80
```

Diagrammatic representation of variables and pointers created in the above program, is given below:



Generally it is expected that the data type of the pointer variable and the date type of the variable whose address is being assigned to the pointer variable must be same. A pointer to an integer stores an address of an integer variable and a pointer to double stores address of a double variable. But suppose we mix data type of the pointer variable and data type of the variable whose address is to be stored in the pointer variable, then C++ compiler will flag a **warning** (**Warning message: Suspicious Pointer Conversion**). An example is given below showing mixing data type while assigning address to pointer variables:

```
#include<iostream.h>
void main()
{
   int a=2014, *ip;
   double b=89.7, *dp;
   ip=&b;
   dp=&a;
   cout<<"a="<<a<<" , b="<<b<<endl;
   cout<<"ip="<<ip<<" , dp="<<dp<<endl;
}
```

Pointer **ip** (integer pointer) is assigned address of **b** (double variable) and **dp** (double pointer) is assigned address of **a** (integer variable). Pointer variables **ip** and **dp** displays the address correctly. Then why does compiler flags warning? We discuss this issue later.

Running of the program produces following output:

```
a=2014 , b=89.7
ip=0x0012ff88 , dp=0x0012ff80
```

Any pointer variable can be assigned `NULL` pointer (`NULL` value). `NULL` pointer represent 0 (zero) address. A `NULL` pointer contains address `0x00000000`. An example is given below:

```
#include<iostream.h>
void main()
{
   int *ip=NULL;
   double *dp=NULL;
   cout<<"ip="<<ip<<endl<<"dp="<<dp<<endl;
}
```

Running of the program produces following output:

```
ip=0x00000000
dp=0x00000000
```

Since a pointer variable is variable, so just like any other variable, a pointer variable can either be a global variable or it can be a local variable. A global pointer variable is created just after the header files and before any block where as a local pointer variable is created inside a block. **Default value of a global pointer variable is `NULL` pointer and default value of a local pointer variable is a garbage address**. **A global pointer variable has all the characteristics of a global variable and a local pointer variable has all the characteristics of a local variable**. If global pointer variable and a local pointer variable have same name inside a block then scope resolution operator (::) is to be used with global pointer variable name, so that, both the global pointer variable and the local pointer variable can be used inside the same block.

```
#include<iostream.h>
int *gp, *p;
void main()
{
   int *lp;
   cout<<"gp="<<gp<<" , lp="<<lp<<endl;
   int a=2014, *p=&a;
   cout<<"::p="<<::p<<" , p="<<p<<endl;
}
```

Global pointer variables **gp** and **::p** are created but not initialised and hence display **NULL** address. **Local** pointer variables **lp** and **p** are also created but not assigned any address and therefore display garbage addresses. Scope resolution operator is used with global pointer variable **p** since there is a local pointer variable **p** in the **main**() function block.

Running of the program produces following output:

```
gp=0x00000000 , lp=0x0040ff27
::p=0x00000000 , p=0x0012ff84
```

Dereferencing (Indirection)

A pointer variable contains an address of a variable. Indirectly accessing a variable (memory location where the pointer variable is pointing to) through the pointer variable, is called **Dereferencing** or **Indirection**. Unary operator star (*) is used with a pointer variable as a **dereferencing** operator. An example is given below:

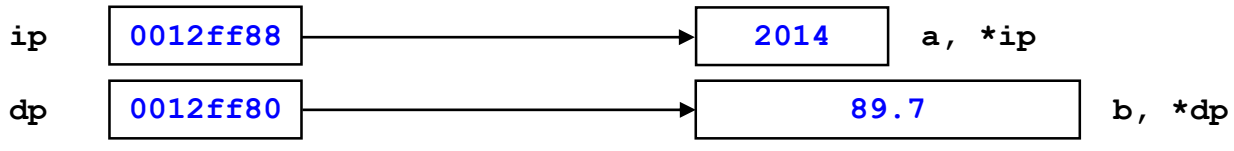
```
#include<iostream.h>
void main()
{
   int a=2014, *ip=&a;
   double b=89.7, *dp=&b;
   cout<<"ip="<<ip<<" , dp="<<dp<<endl;
   cout<<"*a="<<a<<" , b="<<b<<endl;
   cout<<"*ip="<<*ip<<" , *dp="<<*dp<<endl;
   a=2013;
   b=92.3;
   cout<<"Output after 1st updation\n";
   cout<<"*a="<<a<<" , b="<<b<<endl;
   cout<<"*ip="<<*ip<<" , *dp="<<*dp<<endl;
   *ip=2012;
   *dp=91.8;
   cout<<"Output after 2nd updation\n";
   cout<<"*a="<<a<<" , b="<<b<<endl;
   cout<<"*ip="<<*ip<<" , *dp="<<*dp<<endl;
   cout<<"Output after 1st updation\n";
}
```

Pointer variables **ip** and **dp** points to variables **a** and **b** respectively. Expressions ***ip** and ***dp** access variables **a** and **b** respectively. Since variables **a** and ***ip** share same memory location, they are alias of each other. Similarly variables **b** and ***dp** are alias of each other. Any change in the variable **a** will update ***ip** and vice versa. Any updation of either **a** or ***ip** will not update address stored in the pointer variable **ip**. Also any change in the variable **b** will change ***dp** and vice versa. Any updation of either **b** or ***dp** will not update address stored in the pointer variable **dp**.

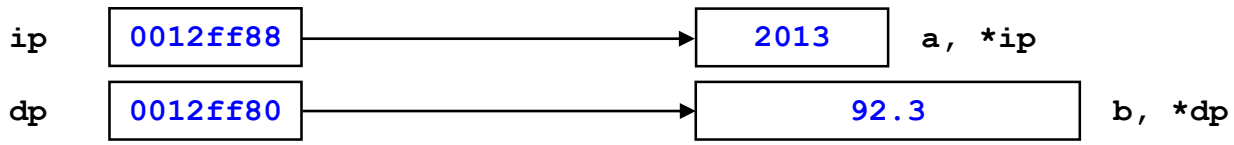
Running of the program produces following output:

```
ip=0x0012ff88 , dp=0x0012ff80
a=2014 , b=89.7
*ip=2014 , *dp=89.7
Output after 1st updation
ip=0x0012ff88 , dp=0x0012ff80
a=2013 , b=92.3
*ip=2013 , *dp=92.3
Output after 2nd updation
ip=0x0012ff88 , dp=0x0012ff80
a=2012 , b=91.8
*ip=2012 , *dp=91.8
```

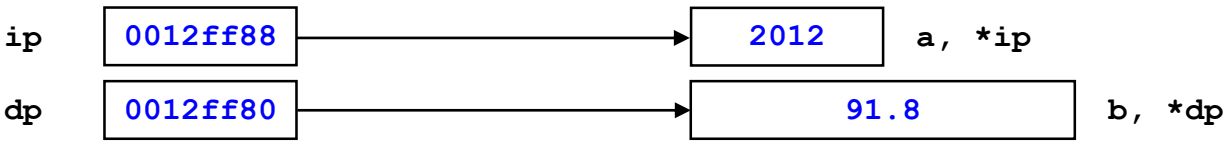
Diagrammatic representation of variables and pointers created in the above program, is given below:



Diagrammatic representation of variables and pointers created in the above program after **1st updation**, is given below:



Diagrammatic representation of variables and pointers created in the above program after **2nd updation**, is given below:



Now coming back to the point, why it is not proper to mix up data type of a pointer variable and the data type of the variable whose address is to be assigned to the pointer variable.

```
#include<iostream.h>
void main()
{
   int a=2014, *ip=&a;
   double b=89.7, *dp=&b;
   cout<<"Before\n";
   cout<<"ip="<<ip<<" , dp="<<dp<<endl;
   cout<<"*ip="<<*ip<<" , *dp="<<*dp<<endl;
   ip=&b;
   dp=&a;
   cout<<"After\n";
   cout<<"ip="<<ip<<" , dp="<<dp<<endl;
   cout<<"*ip="<<*ip<<" , *dp="<<*dp<<endl;
}
```

Running of the program produces following output:

```
Before
ip=0x0018ff50 , dp=0x0018ff48
*ip=2014 , *dp=89.7
After
ip=0x0018ff48 , dp=0x0018ff50
*ip=-858993459 , *dp=3.47641e-308
```

Program is compiled with a warning but addresses are displayed properly. Problem arises with dereferencing. A pointer to an integer **ip**, will access a memory location which is allocated 4 bytes. But the pointer variable **ip** is assigned the address of **b**, which of the type **double**. A variable of the type **double** is allocated 8 bytes. But the pointer variable accesses only 4 bytes and as a result it points to a garbage value. As far as the address is concerned, address is stored correctly but dereferencing generates garbage value. Hence it is a bad practice to mix data type of a pointer variable and data type of the variable.

Pointer to character

Pointer to character (**char***) is little different from pointer to any other data type. In C++ pointer to a character is treated like a string. A string in C++ is terminated by a **nul** character. In C++ array of character, pointer to character and string are used interchangeably. All the string based functions of header file <string.h> uses **char*** as parameter instead of array of character. An example of pointer to a character is given below:

```
#include<iostream.h>
void main()
{
   char x='S';
   char *cp=&x;
   cout<<"cp="<<cp<<endl;
   cout<<"*cp="<<*cp<<endl;
}
```

Running of the program produces following output:

```
cp=S¶ ↕
*cp=S
```

Pointer to **char** (**cp**), does not display address stored in the pointer to **char** (**cp**). It displays value stored in the character variable **x** and then few garbage characters. Because a pointer to a character is treated as a string. In C++, string is an array of character terminated by a **nul** character. So when displaying a pointer to a character, **cout** looks for a terminating **nul** character. So **cout** displays garbage characters after **S** till it encounters terminating **nul** character. The concept of dereferencing remains the same.

<table>
<tr><th>Pointer to char</th><th>Pointer to int (any other data type)</th></tr>
<tr><td>
• Pointer to char does not display address stored in a pointer to char
<pre>
#include<iostream.h>
void main()
{
   char x='S', *cp=&x;
   cout<<"cp="<<cp<<endl;
   cout<<"*cp="<<*cp<<endl;
}
</pre>
Running of the program produces following output:
cp=S¶ ↕
*cp=S
</td><td>
• Pointer to int (pointer to any other data type) displays address in a pointer to int
<pre>
#include<iostream.h>
void main()
{
   int y=100, *ip=&y;
   cout<<"ip="<<ip<<endl;
   cout<<"*ip="<<*ip<<endl;
}
</pre>
Running of the program produces following output:
ip=0012ff80
*ip=100
</td></tr>
<tr><td>
• Inputting a value using pointer to a char is allowed but it may lead to logical error; this is because pointer to a char represents string and a string can be inputted
<pre>
#include<iostream.h>
void main()
{
   char *cp;
   cout<<"String? "; cin>>cp;
   cout<<"cp="<<cp<<endl;
}
</pre>
Running of the program:
String? POINTER
cp=POINTER
Or,
Program may crash.
</td><td>
• Inputting value using a pointer to int (pointer to any other type) will flag syntax error; this because pointer to int represents address and address cannot be inputted
<pre>
#include<iostream.h>
void main()
{
   int *ip;
   cout<<"Address? ";
   cin>>ip;
   cout<<"ip="<<ip<<endl;
}
</pre>
Running of the program:
Compiler flags syntax error in the highlighted line
</td></tr>
</table>

Pointer to struct (structure) / class type

Just like pointer to fundamental data type (**char** / **int** / **float** / **double**) we can also have pointer to derived type like pointer to struct (structure) / class type. A struct / class type has to be declared first then pointer to that struct / class type is to be created. One major difference between pointer to a fundamental data type and pointer structure (class) type is the use of **dereferencing** (**indirection**) operator. For a pointer to a fundamental type unary **star** operator (*) is used as **dereferencing** (**indirection**) operator but generally for pointer to struct / class type binary **arrow** operator (->) is used as **dereferencing** (**indirection**) operator. An **arrow** operator consists of two characters: **dash**/**minus** (-) followed by **greater than sign** (>). An example is given below:

```
#include<iostream.h>
struct student
{
   int roll;
   char name[20];
   double mark;
};
```

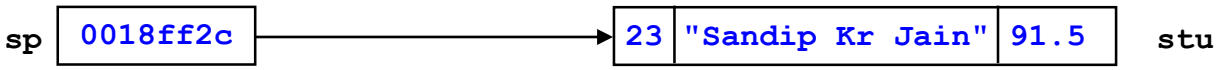
Pointer variable **sp** is a pointer to **student** (struct type) and is assigned the address of the structure variable **stu**. Displaying the pointer variable **sp** will display the address of the variable **stu**. Generally an arrow operator (**->**) is used as a dereferencing operator.

```
void main()
{
   student stu={23, "Sandip Kr Jain", 91.5}, *sp=&stu;
   cout<<"sp="<<sp<<endl;
   cout<<sp->roll<<" , "<<sp->name<<" , "<<sp->mark<<endl;
}
```

Running of the program produces following output:

```
sp=0x0018ff2c
23 , Sandip Kr Jain , 91.5
```

Diagrammatic representation of variables and pointers created in the above program, is given below:



Consider the structure declaration of `student` and the pointer variable `sp` created in the above example, then following C++ statements will flag syntax error (all three statements):

```
cin>>*sp;
cout<<*sp<<endl;
cout<<*sp.roll<<*sp.name<<*sp.mark<<endl;
```

Pointer variable `sp` points to `stu`, that is, `*sp` is of the student (struct) type. C++ statements `cin>>*sp;` and `cout<<*sp;` will flag syntax errors. Using star (`*`) as a dereferencing operator with a pointer to struct (class) type, expressions `*sp.roll`, `*sp.name` and `*sp.mark` will flag syntax errors. Dot (`.`) operator has higher precedence compared to star (`*`) operator. To remove the syntax errors, parenthesis is needed around the expression `*sp`. Corrected C++ statements are given below:

```
cout<<(*p).roll<<(*p).name<<(*p).mark<<endl;
cout<<p->roll<<p->name<<p->mark<<endl;
```

Expressions `(*sp).roll` and `sp->roll` are same but `(*sp).roll` is more complicated compared to `sp->roll`. `*` as a dereferencing operator can be used with pointer to any data type but `->` can only be used with pointer to struct (class) type. An example of pointer to class is give below:

```
#include<iostream.h>
class employee
{
   char nam[20]; double sal;
   public:
      employee(char* n, double s) { strcpy(nam, n); sal=s; }
      void show() { cout<<nam<<" , "<<sal<<endl; }
};
void main()
{
   employee a("Deepak Agarwal", 90000.0), *ep=&a;
   cout<<"ep="<<ep<<endl;
   ep->show();
}
```

Running of the program produces following output:

```
sp=0x0018ff2c
Deepak Agarwal , 90000
```

A pointer to a class type is exactly similar to pointer to struct type. While dereferencing with a pointer to class type, only **public** members of the class **can be** dereferenced with the pointer variable. **Private** members and **protected** members of the class **cannot** be dereferenced with a pointer to a class type. Consider the class declaration of `employee` and the pointer variable `ep` created in the above example, then following C++ statement will flag as syntax error:

```
cout<<ep->nam<<ep->sal<<endl;
```

Compiler will flag syntax errors because `nam` and `sal` are private members of the class `employee`. There are three ways remove the syntax error:

- Change **class** to **struct** because default visibility label of a member of a **struct** is **public**
- Change the visibility labels of the data members `nam` and `sal` from **private** to **public**
- Add two access functions to return the values stored in the private data members `nam` and `sal` and instead of using the using the private data members `nam` and `sal`, use appropriate access functions

Dynamic Variable

Pointer is an address and why do we need to know the address of a variable? Well we are ready to answer this question. Every type of variables that we have discussed so far – variables of fundamental type, array variables, variables of the type struct / class (objects) and pointer variables, all are allocated memory during the compilation time. Once the program is over, memory allocated to these variables are de-allocated. These type variables are called static variable. It is called static because during run-time, no allocation and no de-allocation is possible (or allowed) for these kind of variables. A classic example of a static variable is an array. Array is decided during compilation time since an array size is a positive integer constant. During the run-time, it is neither possible to expand nor possible to contract the size of the array.

So is there any way to create a variable whose memory allocated during run-time and de-allocated during run-time? Answer is yes, it is possible through dynamic variable. A **dynamic variable** is a variable whose memory is allocated and de-allocated during the runtime. To create a dynamic variable we need a pointer variable. Why pointer variable? Because pointer variable will store the address of the dynamic variable. Along with pointer we also need two **unary** operators:

- **`new`** – to allocated memory during the run-time
- **`delete`** – to de-allocated memory during the run-time

Operators **`new`** and **`delete`** are keywords and also called **memory management** operators because these two operators manage allocation/de-allocation of memory during the run-time.

Rule:
```
DataType *PtrVar = new DataType;
delete PtrVar;
```

`DataType` is either fundamental data type or derived data type and `PtrVar` is the name of the pointer variable. Operator **`new`** allocates memory during the run-time and address of the allocated memory is stored in `PtrVar`. When allocating memory during the run-time, data type is important since data type will decide how many byte(s) of memory is(are) to be allocated. Every program is allocated fixed amount of memory. This fixed amount memory space to be used for global variables, local variables and dynamic variables. During the run-time this fixed amount of memory space may get exhausted. If this happens then the operator **`new`** will fail to allocate memory during the run-time and in that case pointer variable `PtrVar` will store a value `NULL`.

Operator **`delete`** de-allocates memory pointed to by `PtrVar`. If memory is allocated but not de-allocated – it will result in memory leakage. Example of operators **`new`** and **`delete`** is given in the next page:

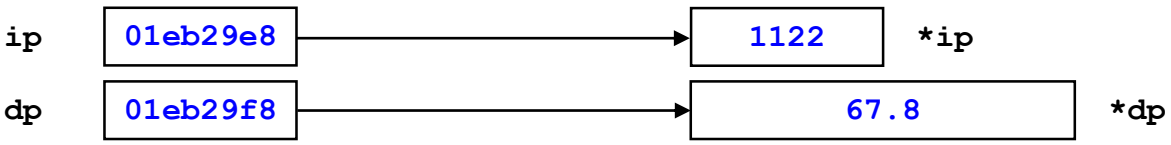
```
#include<iostream.h>
void main()
{
   int *ip=new int;
   double *dp=new double;
   *ip=1122;
   *dp=6.87;
   cout<<"ip="<<ip<<",*ip="<<*ip<<endl;
   cout<<"dp="<<dp<<",*dp="<<*dp<<endl;
   delete dp;
   delete ip;
   cout<<"ip="<<ip<<",*ip="<<*ip<<endl;
   cout<<"dp="<<dp<<",*dp="<<*dp<<endl;
}
```

Running of the program produces following output:

```
ip=0x01d329e8,*ip=1122
dp=0x01d329f8,*dp=6.78
ip=0x01d329e8,*ip=4241844
dp=0x01d329f8,*dp=1.86076e-307
```

Expression **new int** allocate memory dynamically during the run-time whose address is stored in the pointer variable **ip**. Newly allocated memory location is called ***ip**. Expression **new double** allocate memory dynamically during the run-time whose address is stored in the pointer variable **dp**. Newly allocated memory location is called ***dp**. Values are assigned to ***ip** and ***dp**. Expressions delete **ip** de-allocates memory pointed to by **ip**, that is, memory allocated to ***ip** is de-allocated. Expression **delete dp** de-allocates memory pointed to by **dp**. It is bad practice to access a dynamic variable after dereferencing.

Diagrammatic representation pointers and dynamic variables created in the above program is given below:



In the previous example, memory was allocated dynamically and the address was stored in a pointer variable. Value was stored in dynamic variable by using assignment operator. But value can be stored in dynamic variable when the dynamic variable is being created. An example is given below:

```
int *ip=new int (1122);
double *dp=new double (67.8);
cout<<"ip="<<ip<<" , *ip="<<*ip<<endl;
cout<<"dp="<<dp<<" , *dp="<<*dp<<endl;
delete ip;
delete dp;
```

Running of the program segment will produces following output:

```
ip=0x00902a08 , *ip=1122
dp=0x009029e8 , *dp=67.8
```

Value that is to be assigned to the newly created memory location is written within a pair of parenthesis. Statement `int *ip=new int (1122);` does **three** things:

- Creates a pointer variable `ip`
- Address of dynamic variable (`*ip`) is stored in `ip`
- Dynamic variable (newly allocated memory location `*ip`) is initialized with a value `1122`, that is, the memory location `*ip` stores a value `1122`

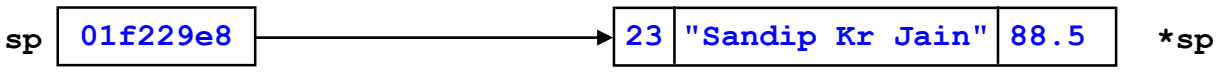
Concept of dynamic variable is also applicable for derived type like struct and class. As mentioned earlier, operators **new** and **delete** can be used with pointer to derived type (struct / class type). Examples are given in the next page showing the use of **new** and **delete** with pointer to struct / class.

```
#include<iostream.h>
struct student
{
   int roll; char name[20]; double marks;
};
void main()
{
   student stu={23, "Sandip Kr Jain", 88.5};
   student *sp=new student (stu);
   cout<<"sp="<<sp<<endl;
   cout<<sp->roll<<" , "<<sp->name<<" , "<<sp->marks<<endl;
   delete sp;
}
```

Running of the program produces following output:

```
sp=0x01f229e8
23 , Sandip Kr Jain , 88.5
```

Diagrammatic representation of pointer and dynamic variable created in the above program, is given below:

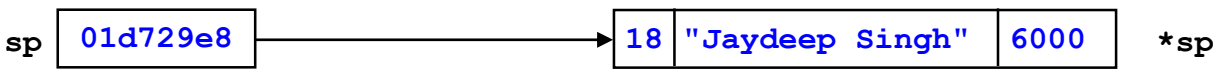



```
#include<iostream.h>
class student
{
   int roll; char name[20]; double fees;
   public:
      student(int ro, char* na, double fe)
      {
         roll=ro;
         strcpy(name, na);
         fees=fe;
      }
      void display() { cout<<roll<<" , "<<name<<" , "<<fees<<endl; }
};
void main()
{
   student *sp=new student(18, "Jaydeep Singh", 6000);
   cout<<"sp="<<sp<<endl;
   sp->display();
   delete sp;
}
```

Running of the program produces following output:

```
sp=0x01d729e8
18 , Jaydeep Singh , 6000
```

Diagrammatic representation of pointer and dynamic variable created in the above program, is given below:



Array and Pointer

In C++ array and pointers are very closely related. Array name is a constant pointer – represents the address of first element of an array. Displaying an array name (except for array of **char** – displays string) will display the starting address of the array. Since an array name is a pointer, array name can be assigned to a pointer variable. It is important to note that array variable's data type and pointer variable's data must be same. If pointer variable's data type and array variable's data type do not match then the compiler will either flag a warning or a syntax error. An example is given below:

```
#include<iostream.h>
void main()
{
   int a[]={12, 35, 46, 89, 63}, *ip=a;
   char b[]="JULY MORNING", *cp=b;
   double c[]={1.2, 3.5, 4.6, 8.9, 6.3}, *dp=c;
   cout<<"Address of a="<<a<<" , "<<&a[0]<<" , "<<ip<<endl;
   cout<<"String b="<<b<<" , "<<&b[0]<<" , "<<cp<<endl;
   cout<<"Address of c="<<c<<" , "<<&c[0]<<" , "<<ip<<endl;
   for (int k=0; k<5; k++)
      cout<<a[k]<<" , "<<c[k]<<endl;
   ip=c; dp=a;
   cout<<"ip="<<ip<<endl;
   cout<<"dp="<<dp<<endl;
}
```

Running of the program produces following output:

```
Address of array a=0x0018ff40 , 0x0018ff40 , 0x0018ff40
String b=JULY MORNING , JULY MORNING , JULY MORNING
Address of array c=0x0018ff08 , 0x0018ff08 , 0x0018ff08
12 , 1.2
35 , 3.5
46 , 4.6
89 , 8.9
63 , 6.3
ip=0x0018ff04
dp=0x0018ff40
```

Borland C++ 5.0 compiler will flag a warning for **ip=c;** and **dp=a;** Since **ip** (pointer to an **int**) is assigned the address of array **c** (array of **double** - pointer to **double**) and **dp** (pointer to **double**) is assigned the address of array **a** (array of **int** - pointer to **int**).

Pointer of the type `void` is called **generic** pointer(**type less** pointer). A generic pointer can store address of any variable / array. But disadvantage of generic pointer is that, **dereferencing** a generic pointer will flag **syntax error**. An example is given below:

```
#include<iostream.h>
void main()
{
   int x=39;
   char y='T';
   double z=2.5;
   void *p=&x;
   cout<<p<<" , "<<*p<<endl;
   p=&y;
   cout<<p<<" , "<<*p<<endl;
   p=&z;
   cout<<p<<" , "<<*p<<endl;
}
```

Pointer variables **p** is a generic pointer (pointer to **void**). Pointer variable **p** first stores address of **x** and next it stores address of **y**. Finally it stores the address of **z**. When compiling the program, expression ***p** will flag syntax error since ***p** is of the type **void**. It is possible to dereference a generic pointer by proper typecasting. An example is given in the next page after making the necessary corrections.

```
#include<iostream.h>
void main()
{
   int x=39;
   char y='T';
   double z=2.5;
   void *p=&x;
   cout<<p<<" , "<<*(int*)p<<endl;
   p=&y;
   cout<<p<<" , "<<*(char*)p<<endl;
   p=&z;
   cout<<p<<" , "<<*(double*)p<<endl;
}
```

Running of the program produces following output:

```
0x0018ff50 , 39
0x0018ff4f , T
0x0018ff44 , 2.5
```

Expression **`*(int*)p`** first type cast pointer variable **`p`** (generic pointer) to pointer to **`int`** (**`int*`**). Next **`*(int*)p`** will represent the value in the memory location pointed to by the **`p`**.
Expression **`*(char*)p`** first type cast pointer variable **`p`** (generic pointer) to pointer to **`char`** (**`char*`**). Next **`*(char*)p`** will represent the value in the memory location pointed to by the **`p`**.
Expression **`*(double*) p`** first type cast pointer variable **`p`** (generic pointer) to pointer to **`double`** (**`double*`**). Next **`*(double*)p`** will represent the value in the memory location pointed to by the **`p`**.

Dynamic Array

Any array in C++ is allocated memory during compilation time and that is reason why array size in C++ has to be a positive integer constant. Any attempt to create an array variable where array size is a variable will result in syntax error. But with dynamic memory it is possible to create an array whose size can be decided during the run-time and memory allocated to a dynamic array can be de-allocated during the run-time Dynamic array is created during the run-time by using the operator **`new`** and it is de-allocated during the run-time by using the operator **`delete`**.

Rule:
```
DataType *PtrVar = new DataType [Size];
delete []PtrVar;
```

`DataType` is the data type is either fundamental type or derived type and `PtrVar` is the name of the pointer variable. `Size` represents size of the array. `Size` could either be or **positive integer constant** or **positive integer variable** or **positive integer expression**. Operator **`new`** allocates a block of memory with `Size` number of contiguous memory locations and starting address of the block will be stored in the pointer variable `PtrVar` (dynamic array name). The pointer variable `PtrVar` will a value `NULL` if no more memory space is available for allocation during the run-time.
Operator **`delete`** de-allocates contiguous memory block during the run-time pointed to by the pointer variable `PtrVar` (dynamic array name). Entire block of memory location will be de-allocated. Use of [] before the pointer variable `PtrVar` is very important because without [], **`delete`** only de-allocates the first memory location in the block.

Example 1 (Dynamic Array of Integers):

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
   int n;
   cout<<"Number of elements? "; cin>>n;
   int *arr=new int[n];
   for (int x=0; x<n; x++)
      arr[x]=random(90)+10;
```

```
   for (int k=1; k<n; k++)
      for (int j=0; j<n-k; j++)
         if (arr[j]>arr[j+1])
         {
            int t=arr[j]; arr[j]=arr[j+1]; arr[j+1]=t;
         }
   for (int c=0; c<n; c++)
      cout<<arr[c]<<"  ";
   delete []arr;
}
```

Running of the program produces following output:

```
Number of elements? 15
14  20  22  26  34  36  37  42  66  67  76  85  90  96  97
```

Example 2 (Dynamic Array of floating point values):

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
   int n;
   cout<<"Number of elements? "; cin>>n;
   double *arr=new double[n];
   for (int x=0; x<n; x++)
      arr[x]=(random(90)+10)/10.0;
   for (int k=1; k<n; k++)
      for (int j=0; j<n-k; j++)
         if (arr[j]>arr[j+1])
         {
            double t=arr[j]; arr[j]=arr[j+1]; arr[j+1]=t;
         }
   for (int c=0; c<n; c++)
      cout<<arr[c]<<"  ";
   delete []arr;
}
```

Running of the program produces following output:

```
Number of elements? 10
2.2  2.4  4.1  5  6  6.3  7.3  8.6  8.9  9.2
```

Example 3 (Dynamic Array of characters):

```
#include<iostream.h>
#include<stdio.h>
void main()
{
   char *str=new char[40];
   cout<<"Input a string? "; gets(str);
   cout<<"Inputted string="<<str<<endl;
   delete []str;
}
```

Running of the program produces following output:

```
Input a string? Weekends are Friday and Saturday
Inputted string=Weekends are Fridays and Saturdays
```

Example 4 (Dynamic Array of structure type):

```
#include<iostream.h>
struct student
{
   char name[10]; double mark;
};
void main()
{
   int n;
   cout<<"Positive integer? "; cin>>n;
   student* arr=new student[n];
   for (int x=0; x<n; x++)
   {
      cout<<"Name and Mark? "; cin>>arr[x].name>>arr[x].mark;
   }
   for (int k=1; k<n; k++)
      for (int j=0; j<n-k; j++)
         if (arr[j].mark<arr[j+1].mark)
         {
            student t=arr[j]; arr[j]=arr[j+1]; arr[j+1]=t;
         }
   cout<<"Displaying array sorted in descending order on Mark\n";
   for (int c=0; c<n; c++)
      cout<<arr[c].name<<" , "<<arr[c].mark<<endl;
   delete []arr;
}
```

Running of the program produces following output:

```
Number of elements? 5
Name and Mark? Ankita 90.5
Name and Mark? Hitesh 78.5
Name and Mark? Sooraj 93.5
Name and Mark? Deepak 88.5
Name and Mark? Farida 82.5
Displaying array sorted in descending order on Mark
Sooraj , 93.5
Ankita , 90.5
Deepak , 88.5
Farida , 82.5
Hitesh , 78.5
```

What can / cannot be done with a pointer variable?

1. A pointer variable can be assigned an address.
   - A pointer variable can be assigned a value NULL

```
int *ip=NULL
double *dp=NULL;
```

   - A pointer variable can be assigned an address of a variable / array

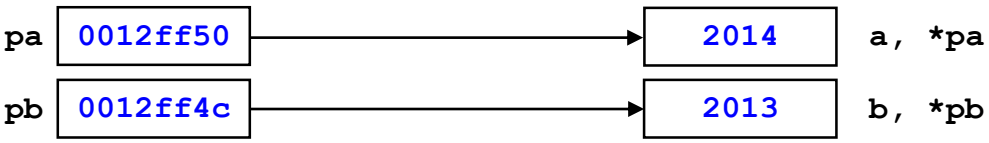
```
int a=2014, arr1[]={10, 20, 30, 40, 50};
char str[]="Summer Break!", *cp=str;
double b=91.4, arr2[]={1.4, 3.2, 5.3, 2.7, 4.8};
int *ip1=&a, *ip2=arr1;
double *dp1=&b, *dp2=arr2;
```

- A pointer variable can be assigned an address of another pointer variable (same data type)

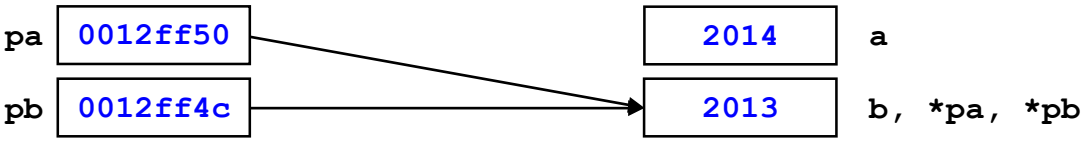
```
int a=2014, b=2013, *pa=&a, *pb=&b;
cout<<"pa="<<pa<<" , *pa="<<*pa<<endl;
cout<<"pb="<<pb<<" , *pb="<<*pb<<endl;
pa=pb;
cout<<"pa="<<pa<<" , *pa="<<*pa<<endl;
cout<<"pb="<<pb<<" , *pb="<<*pb<<endl;
```

Running of the program segment produces following output:

```
pa=0x0018ff50 , *pa=2014
pb=0x0018ff4c , *pb=2013
pa=0x0018ff4c , *pa=2013
pb=0x0018ff4c , *pb=2013
```



After assigning address stored in the variable `pb` to the pointer variable `pa` (`pa=pb;`):



- A pointer variable can be assigned an address of a dynamic variable / dynamic array

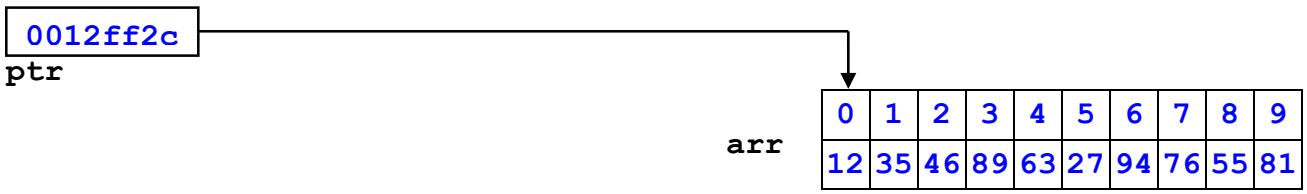
```
int *ip=new int;
char *cp=new char;
double *dp=new double;
int *arr1=new int[20];
char *arr2=new char[80];
double *arr3=new double[10];
```

2. A pointer variable can be displayed.

```
int a=2014, arr1[]={10, 20, 30, 40, 50};
double b=91.4, arr2[]={1.4, 3.2, 5.3, 2.7, 4.8};
int *ip1=&a, *ip2=arr1;
double *dp1=&b, *dp2=arr2;
cout<<"ip1="<<ip1<<" , ip2="<<ip2<<endl;
cout<<"dp1="<<dp1<<" , dp2="<<dp2<<endl;
```

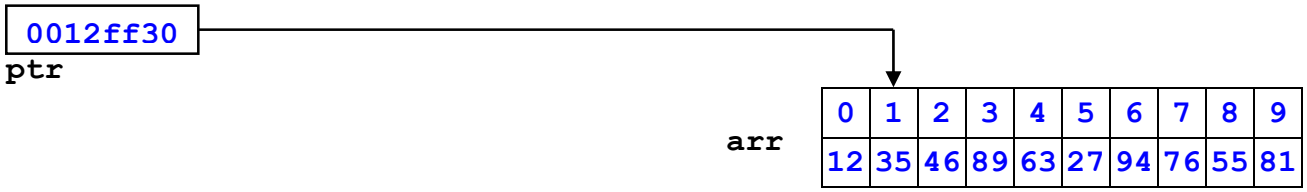
3. Increment (++) and decrement (--) operators can be used with a pointer variable
Increment and decrement operator used with a pointer variable legally, when a pointer variable stores an address of an array.

```
int arr[]={12, 35, 46, 89, 63, 27, 94, 76, 55, 81}, *ptr=arr;
```



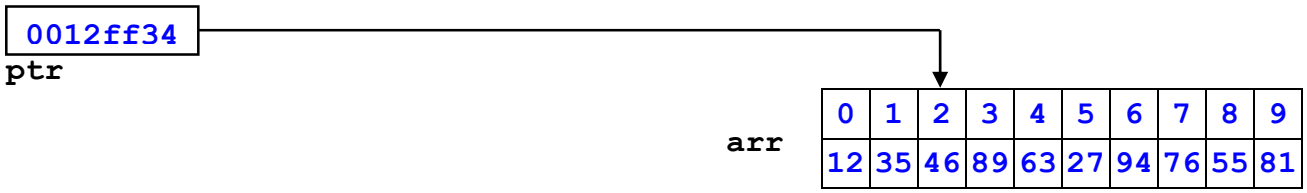
`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the first element of the array and the value stored in the first element in the array (`0x0018ff2c , 12`) will be displayed. `ptr++;`

(Or `++ptr;` Or `ptr=ptr+1;` Or `ptr+=1;`) will update the pointer variable `ptr` to point to the second element (index 1) of the array.



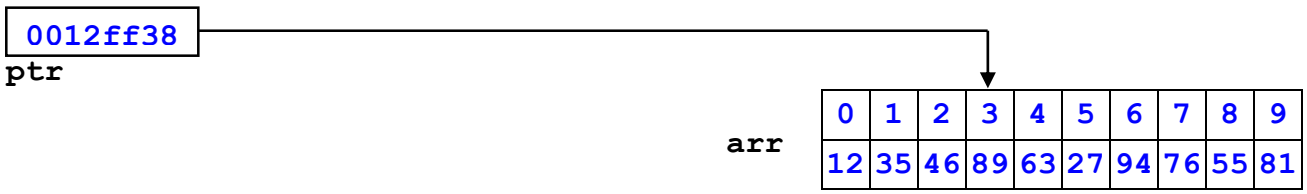
`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the second element of the array and the value stored in the second element in the array (`0x0018ff30 , 35`) will be displayed.

`ptr++;` will update the pointer variable `ptr` to point to the third element (index 2) of the array.



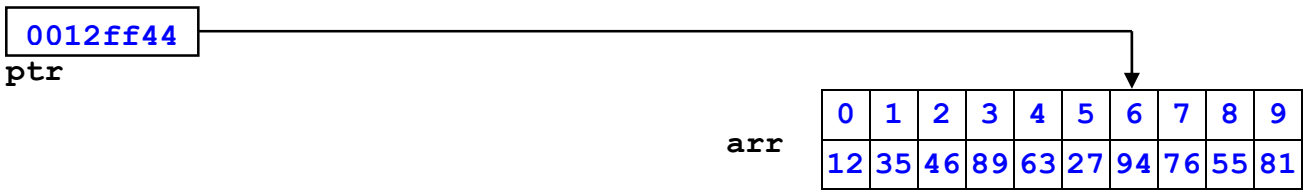
`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the third element of the array and the value stored in the third element in the array (`0x0018ff34 , 46`) will be displayed.

`ptr++;` will update the pointer variable `ptr` to point to the fourth element (index 3) of the array.



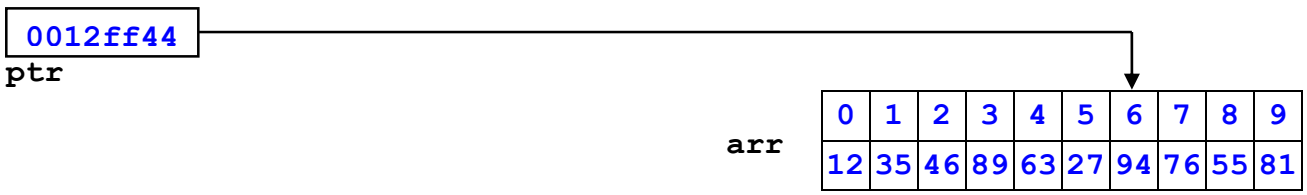
`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the fourth element of the array and the value stored in the fourth element in the array (`0x0018ff38 , 89`) will be displayed.

`ptr+=3;` will update the pointer variable `ptr` to point to the seventh element (index 6) of the array.



`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the seventh element of the array and the value stored in the seventh element in the array (`0x0018ff44 , 94`) will be displayed.

`ptr+=3;` will update the pointer variable `ptr` to point to the tenth element (index 6) of the array.



`cout<<ptr<<" , "<<*ptr<<endl;` will display address of the seventh element of the array and the value stored in the seventh element in the array (`0x0018ff44 , 94`) will be displayed.

So it is very clear that `ptr++` will update the pointer variable to point to the next element of the array. But if the pointer variable `ptr` is pointing to the last element of the array, then `ptr++` will point to an unallocated memory location containing garbage value. The example above uses an array of **`int`** and pointer to an **`int`** but the concept is applicable for any data type including **`char`** type derived data type struct / class.

4. A pointer variable can be passed as a value / reference parameter to a function
5. A value cannot be inputted into a pointer variable except pointer to a character