Hexadecimal Integer Constant: In a programming language like C++, it is possible to represent an integer constant in different form. Generally an integer constant is represented as a **Decimal** integer constant. A decimal integer constant consists of any 10 digits (0-9). Integers `29`, `73545`, `8545`, `-34`, `-428954` and `3945` are example of **Decimal** integer constant. In C++ it is also possible to represent an integer constant as a **Hexadecimal** integer. A **Hexadecimal** integer constant consists of any 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`. A example is given below:

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

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

Running of the program produces following output:

```
Hex=6988 , Dec=174911
Hex=1B4C , Dec=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 values 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 the 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 Hexadecimal integer. An example is given below:

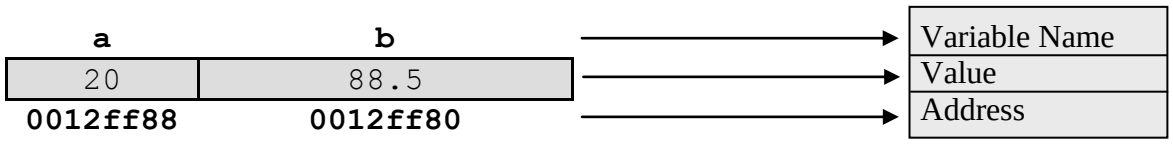
```
#include<iostream.h>
void main()
{
   int a=20;
   double b=88.5;
   cout<<"a="<<a<<" , b="<<b<<endl;
   cout<<"&a="<<&a<<" , &b="<<&b<<endl;
   cout<<"&b="<<&b<<endl;
}
```

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** integer.

Running of the program produces following output:

```
roll=20 , mark=88.5
&roll=0x0012ff88 , &mark=0x0012ff80
```

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



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* ip;
int *ip1, *ip2;

char* cp;
char *cp1, *cp2;

double* dp;
double *dp1, *dp2;
```

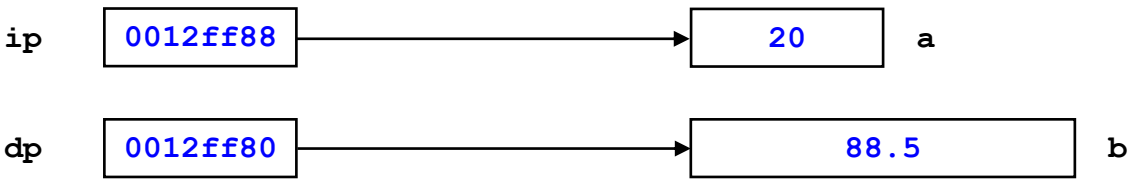
- Statement `int* ip;` creates an integer pointer (pointer to an integer). An integer pointer can store an address of an integer variable.
- Statement `char* cp;` creates a character pointer (pointer to a character). A character pointer can store an address of a character type variable.
- Statement `double*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

Example:
```
void main()
{
   int a=20, *ip;
   double b=88.5, *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** stores address of **a** and **dp** stores address of **b**. But creation of pointer variable and assigning an address to it can be combined as one statement. For example:
```
int* ip=&a;
double* dp=&b;
```

Running of the program produces following output:
```
a=20 , b=88.5
ip=0x0012ff88 , dp=0x0012ff80
```

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



Generally it is expected that data type of the pointer variable and the date type of the variable whose address is to be stored in 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:

```
#include<iostream.h>
void main()
{
   int a=20, *ip;
   double b=88.5, *dp;
   ip=&b;
   dp=&a;
   cout<<"a="<<a<<" , b="<<b<<endl;
   cout<<"ip="<<ip<<" , dp="<<dp<<endl;
}
```

Pointer **ip** (integer pointer) stores address of **b** (double variable) and **dp** (double pointer) stores address of **a** (integer variable). C++ compiler will flag a **warning** and the Warning Message is: **Suspicious Pointer Conversion**. But the two pointer variables display correct addresses on the screen.

Running of the program produces following output:

```
a=20 , b=88.5
ip=0x0012ff88 , dp=0x0012ff80
```

A pointer variable just like any other variable can be 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 (**zero address**) 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 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 block. A value **NULL** can be assigned to any pointer variable (similar to assigning 0 to an integer or a floating point variable).

```
#include<iostream.h>
int* p1;
double *p2, *p;
void main()
{
   int *p1, p;
   double* p2;
   cout<<"p1="<<p1<<" , p2="<<p2<<endl;
   cout<<"g1="<<g1<<" , g2="<<g2<<endl;
   p1=NULL;
   p2=NULL;
   cout<<"p1="<<p1<<" , p2="<<p2<<endl;
   int a=20, *p=&a;
   double b=88.5;
   ::p=&b;
   cout<<"p="<<p<<" , ::p="<<::p<<endl;
}
```

Local pointer variables **p1** and **p2** are created but not initialised and hence they display **garbage** address but **Global** pointer variables **g1** and **g2** will display **NULL** (**zero** address – default value of a global pointer). After assigning **NULL** to the local pointer variables **p1** and **p2**, pointer variables **lip** and **ldp** display **0x00000000** – zero address (NULL). Note the spelling of **NULL**, all letters in uppercase and the spelling is case sensitive. A pointer containing **NULL**, is said to be **grounded**.

Global pointer variable **p** is created as pointer to **double**. **Local** pointer variable **p** is created as pointer to **int**. To use the **global** pointer variable **p** and the **local** pointer variable **p**, **scope resolution operator** (::) is used with the **global** pointer variable name.

Running of the program produces following output:

```
p1=0x0040a02c , p2=0x0040ff27
g1=0x00000000 , g2=0x00000000
p1=0x00000000 , p2=0x00000000
p=0x0012ff88 , ::p=0x0012ff80
```

Dereferencing (Indirection)

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

```
#include<iostream.h>
void main()
{
   int a=20, *ip=&a;
   double b=88.5, *dp=&b;
   cout<<"ip="<<ip<<endl;
   cout<<"dp="<<dp<<endl;
   cout<<"*ip="<<*ip<<endl;
   cout<<"*dp="<<*dp<<endl;
}
```

Pointer variables **ip** and **dp** points to variables **a** and **b** respectively. Expressions ***ip** and ***dp** access variables **a** and **b** respectively.

```
cout<<"*ip="<<*ip<<endl;
```

Statement displays values stored in **a** indirectly through the pointer **ip**.

```
cout<<"*dp="<<*dp<<endl;
```

Statement displays values stored in **b** indirectly through the pointer variable **dp**.

Running of the program produces following output:

```
ip=0x0012ff88
dp=0x0012ff80
*ip=20
*dp=88.5
```

Operations on Pointer Variable

- A pointer variable can be assigned a value NULL.

```
struct student
{
   char name[10];
   double mark;
};
int* ip=NULL;
double* dp=NULL;
student* sp=NULL;
```

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

```
struct student
{
   char name[10];
   double mark;
};
int x1=79, arr1[5]={34, 56, 44, 29, 62};
double x2=6.5, arr2[5]={4.5, 1.2, 3.4, 2.3, 5.6};
student x3={23, "Gajendra", 88.5};
student arr3[3]={ {"Chandana", 93.5},
                  {"Animesh", 90.0},
                  {"Farida", 82.5}};
int *ip1=&x1, *ip2=arr1;
double *dp1=&x2, *dp2=arr2;
student *sp1=&x3, *sp2=arr3;
```

- A pointer variable can be assigned a value stored in another pointer variable, provided both the pointer variables are of the same data type

```
struct student
{
   char name[10];
   double mark;
};
```

```
int x1=79, *ip1=&x1;
double x2=6.5, *dp1=&x2;
student x3={"Gajendra", 88.5}, *sp1=&x3;
int *ip2=ip1;
double *dp2=dp1;
student *sp2=sp1;
```

- A pointer variable can be assigned an address of a dynamically allocated memory location using operator **new.** An operator **delete** can be used to deallocate dynamically allocated memory pointed to by a pointer variable. Examples are given in the next page.

```
struct student
{
   char name[10];
   double mark;
};
int* ip=new int (79);
double* dp=new double (6.5);
student* sp=new student;
strcpy(sp->name, "Sandip");
sp->mark=88.5;
cout<<"*ip="<<*ip<<endl;
cout<<"*dp="<<*dp<<endl;
cout<<"Name ="<<sp->name<<endl;
cout<<"Mark ="<<sp->mark<<endl;
delete ip;
delete dp;
delete sp;
```

- Value stored in pointer variable can be displayed with `cout`

```
struct student
{
   char name[10];
   double mark;
};
int x1=79, *ip=&x1;
double x2=6.5, *dp=&x2
student x3={"Gajendra", 88.5}, *sp=&x3;
cout<<"ip="<<ip<<" , *ip="<<endl;
cout<<"dp="<<dp<<" , *dp="<<endl;
cout<<"sp="<<sp<<endl;
cout<<"Name="<<sp->name<<endl;
cout<<"Mark="<<sp->mark<<endl;
```

- Arithmetic operators + and − can be used with a pointer variable
  An expression involving a pointer variable is a pointer. Generally the operators plus (+) and minus (−) are used with a pointer variable. Examples are given below:

Example 1:

```
int ar[5]={25, 85, 13, 47, 78}, *p=arr;
cout<<"*p    ="<<*p<<endl;
cout<<"*(p+1)="<<*(p+1)<<endl;
```

```
cout<<"*(p+2)="<<*(p+2)<<endl;
ptr+=4;
cout<<"*p    ="<<*p<<endl;
cout<<"*(p-1)="<<*(p-1)<<endl;
cout<<"*(p-2)="<<*(p-2)<<endl;
```

Running of the program segment produces following output:

```
*p    =25
*(p+1)=85
*(p+2)=13
*p    =78
*(p-1)=47
*(p-2)=13
```

Pointer `p` points to **1st** element of the array `ar[]`. Statement `cout<<*p<<endl;` displays value stored in the **1st** element of array `ar[]`. Expression `p+1` points to **2nd** element of array `ar[]`. Statement `cout<<*(p+1)<<endl;` displays value stored in the **2nd** element of array `ar[]`. Expression `p+2` points to **3rd** element of array `ar[]`. Statement `cout<<*(p+2)<<endl;` displays value stored in the **3rd** element of array `ar[]`. Statement `p+=4;` updates the address stored in pointer `p`, pointer `p` points to **5th** element of array `ar[]`. Statement `cout<<*p<<endl;` displays value stored in the **5th** element of array `ar[]`. Expression `p-1` points to **4th** element of array `ar[]`. Statement `cout<<*(p-1)<<endl;` displays value stored in the **4th** element of array `ar[]`. Expression `p-2` points to **3rd** element of array `ar[]`. Statement `cout<<*(p-2)<<endl;` displays value stored in the **3rd** element of array `ar[]`.

Example 2:

```
double arr[5]={2.3, 8.5, 4.7, 9.2, 6.3}, *p=arr;
cout<<"*p    ="<<*ptr<<endl;
cout<<"*(p+2)="<<*(p+2)<<endl;
ptr+=3;
cout<<"*p    ="<<*p<<endl;
cout<<"*(p-1)="<<*(p-1)<<endl;
p-=2;
cout<<"*p    ="<<*p<<endl;
```

Running of the program segment produces following output:

```
*p    =2.3
*(p+2)=4.7
*p    =9.2
*(p-1)=4.7
*p    =8.5
```

Example 3:

```
char song[]="Metallica-Nothing Else Matter", *p=song;
while (*p)
{
   cout<<*p;
   p+=1;
}
```

Running of the program segment produces following output:

```
Metallica-Nothing Else Matter
```

Example 4:

```
struct student
{
   char name[10];
   double mark;
};
student a[6]={{"Suresh",93.5},{"Ankita",90.5},{"Dileep",88.5},
              {"Farida",82.5},{"Biresh",78.5},{"Nalini",75.5}};
student* p=a;
cout<<p->name<<" , "<<p->mark<<endl;
p+=2;
cout<<p->name<<" , "<<p->mark<<endl;
p+=1;
cout<<ptr->name<<" , "<<ptr->mark<<endl;
```

Running of the program segment produces following output:

```
Suresh , 93.5
Dileep , 88.5
Farida , 82.5
```

- Increment operator (++) and decrement operator (--) can be used with a pointer variable
  Increment operator and decrement operator is used with a pointer variable when the pointer variable is pointing to an array. If `ptr` is a pointer pointing to an element of an array, then `ptr++` will point to next element of the array and `ptr--` will point to previous element of the array.

```
int a[10]={25,85,13,47,78,92,63,31,52,76}, *p1=a, *p2=&a[9];
for (int k=0; k<10; k++, p1++)
   cout<<*p1<<" ";
cout<<endl;
for (int x=0; x<10; x++, p2--)
   cout<<*p2<<" ";
```

Running of the program segment produces following output:

```
25 85 13 47 78 92 63 31 52 76
76 52 31 63 92 78 47 13 85 25
```

- Relational operators == and != can be used with a pointer variable

```
int x1=30, x2=40, *p1=&x1, *p2=&x2;
if (p1==p2)
   cout<<"Same Address        "<<p1<<"=="<<p2<<endl;
else
   cout<<"Different Addresses "<<p1<<"!="<<p2<<endl;
p1=p2;          //Or p2=p1;
if (p1==p2)
   cout<<"Same Address        "<<p1<<"=="<<p2<<endl;
else
   cout<<"Different Addresses "<<p1<<"!="<<p2<<endl;
```

Running of the program segment produces following output:

```
Different Addresses 0x0012ff88!=0x0012ff84
Same Address        0x0012ff84==0x0012ff84
```

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. There are two major differences between pointer to **char** and pointer to any other data type:

1. Displaying **pointer** to **char** will display a string starting from where the **pointer** is pointing to, till pointer points to **nul** character

```
char x1='S', *cp=&x1;
int x2=20, *ip=&x2;
double x3=6.5, *dp=&x3;
cout<<"cp="<<cp<<endl;
cout<<"ip="<<ip<<endl;
cout<<"dp="<<dp<<endl;
cout<<"*cp="<<*cp<<endl;
cout<<"*ip="<<*ip<<endl;
cout<<"*dp="<<*dp<<endl;
```

Pointer variable **cp** (pointer to **char**), displays a string starting from **'S'** (**cp** is pointing to variable **x1** and **x1='S'**. Pointer variable **cp** does not display the address of **x1**. Displaying pointer to any other data type (not a **char** type) displays the address stored in the pointer variable.

Running of the program segment produces following output:

```
cp=S¶ ↕
ip=0x0012ff84
dp=0x0012ff7c
*cp=S
*ip=20
*dp=6.5
```

Statement **cout<<cp<<endl;** displays **S** and few garbage character because **cout** starts displaying the string starting from **S** and then looks for terminating **nul** character and **cout** encounters **nul** character after few garbage characters.

2. Inputting a value through a **pointer** to **char** is **syntactically correct** statement but it may flag a **warning** and may lead to logical error and that may lead to run-time error

```
char* cp;
int* ip;
double* dp;
cin>>cp;
cin>>ip;
cin>>dp;
```

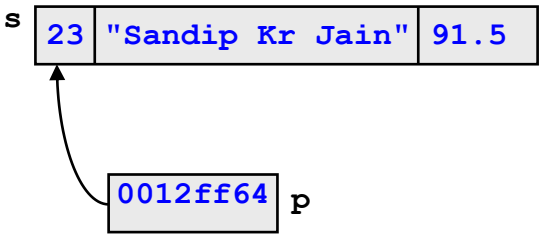
Statement **cin>>cp;** will flag a **warning** but it is **syntactically correct** C++ statement. A string can be inputted through a pointer to **char** because C++ treats pointer to a **char** as a string. Statements **cin>>ip;** and **cin>>dp;** will flag **syntax error** since pointer to other data type (not a **char** type) represents address.

Pointer to 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 structure (class) type. A structure (class) type has to be declared first then pointer to that structure (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 structure (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** (>). Examples are given below:

```
struct student
{
   int roll;
   char name[20];
   double mark;
};
void main()
{
   student s={23, "Sandip Kr Jain", 91.5};
   student *p=&s;
   cout<<"p    ="<<p<<endl;
   cout<<"Roll="<<p->roll<<endl;
   cout<<"Name="<<p->name<<endl;
   cout<<"Mark="<<p->mark<<endl;
}
```

Pointer variable **p** is pointer to **student** (structure type) and it points to **s** (structure variable). Statement **cout<<"p="<<p<<endl** displays the address of **s**. Arrow as dereferencing operator is used between pointer variable name (**p**) and structure member name (**roll** / **name** / **mark**) to display the value stored in the variable **stu**.



Running of the program produces following output:

```
p    =0x0012ff64
Roll=23
Name=Sandip Kr Jain
Mark=91.5
```

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

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

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

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

Expressions `(*p).roll` and `p->roll` are same. But expression `(*p).roll` is more complicated compared to expression `p->roll`. **Star** (*) as a dereferencing operator can be used with pointer to to any data type but **arrow** operator (->) can only be used with pointer to structure (class) type. An example is give on the next page:

```
#include<iostream.h>
class student
{
   int roll;
   char name[20];
   double mark;
   public:
      student(int r, char* n, double m)
      {
         roll=r;
         strcpy(name, n);
         marks=m;
      }
      void display()
      {
         cout<<"Roll="<<roll<<endl;
         cout<<"Name="<<name<<endl;
         cout<<"Mark="<<mark<<endl;
      }
};
void main()
{
   student stu(23, "Sandip Kr Jain", 91.5), *p=&student;
   cout<<"p   ="<<p<<endl;
   p->display();
}
```

Running of the program produces following output:

```
p   =0x0012ff64
Roll=23
Name=Sandip Kr Jain
Mark=91.5
```

A pointer to a class type is exactly similar to pointer to structure 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 `student` and the pointer variable `p` created in the above example, then following C++ statement will flag as syntax error:

```
cout<<p->roll<<p->name<<p->mark<<endl;
```

Compiler will flag syntax errors because `roll`, `name` and `mark` are private members of the class student. There are two ways remove the syntax error:

- Change the visibility labels of data members `roll`, `name` and `mark` from **private** to **public**.
- Add three access functions to return the values stored in private data members `roll`, `name` and `mark`. Pointer variable, dereferencing operator and access function can be used to access the private data members `roll`, `name` and `mark`.

Array and Pointer

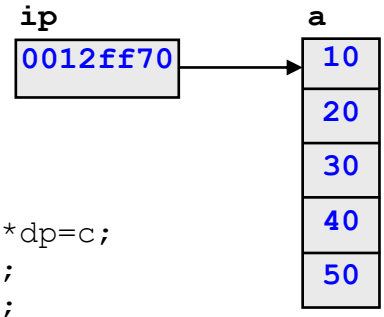
Array name is a constant pointer (address of first element of an array). Displaying an array name (except for array of **char**) will display the starting address of the array. Displaying an array of **char** will display the string stored in the array. An example is given below:

```
#include<iostream.h>
void main()
{
   int a[5]={10, 20, 30, 40, 50};
   char b[6]="APRIL";
   double c[5]={1.2, 2.3, 3.4, 4.5, 5.6};
   cout<<"a="<<a<<" , "<<&a[0]<<endl;
   cout<<"b="<<b<<" , "<<&b[0]<<endl;
   cout<<"c="<<c<<" , "<<&c[0]<<endl;
}
```

Running of the program produces following output:

```
a=0x0012ff70 , 0x0012ff70
b=APRIL , APRIL
c=0x0012ff48 , 0x0012ff48
```

Since array 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. An example is given below:




```
#include<iostream.h>
void main()
{
   int a[5]={10, 20, 30, 40, 50}, *ip=a;
   char b[6]="APRIL", *cp=b;
   double c[5]={1.2, 2.3, 3.4, 4.5, 5.6}, *dp=c;
   cout<<"iarr="<<iarr<<" , ip="<<ip<<endl;
   cout<<"carr="<<carr<<" , cp="<<cp<<endl;
   cout<<"darr="<<darr<<" , dp="<<dp<<endl;
}
```

Running of the program produces following output:

```
a=0x0012ff70 , ip=0x0012ff70
b=APRIL , cp=APRIL
c=0x0012ff48, dp=0x0012ff48
```

If pointer variable's data type and array variable's data type do not match then the compiler will flag a warning. An example is given below:

```
int a[5]={10, 20, 30, 40, 50};
double b[5]={1.2, 2.3, 3.4, 4.5, 5.6};
int* ip=b;
double* dp=a;
cout<<"ip="<<ip<<endl;
cout<<"dp="<<dp<<endl;
```

Statements `int* ip=b;` and `double* dp=a;` will flag warning because `ip` (pointer to **`int`**) stores address `b` (array of **`double`**) and `dp` (pointer to **`double`**) stores address of `a` (array of **`int`**). A pointer to **`void`** will be able to store address of any variable. A pointer to **`void`** is known as **generic** pointer or **type less** pointer. An example is given below:

```
int a[5]={10, 20, 30, 40, 50};
char b[6]="APRIL";
double c[5]={1.2, 2.3, 3.4, 4.5, 5.6};
```

```
void* p=a;
cout<<"Address of a="<<p<<endl;
p=b;
cout<<"Address of b="<<p<<endl;
p=c;
cout<<"Address of c="<<p<<endl;
```

Running of the program segment will produces following output:

```
Address of iarr=0x0012ff70
Address of carr=0x0012ff84
Address of darr=0x0012ff48
```

Pointer variable `p` is a pointer to **`void`** (**generic** pointer). Pointer variable `p` is used to display the address of all the three arrays including array of **`char`** (`b`). It is also possible to display address of an array of **`char`** by type casting the address to pointer to some other data type. One disadvantage of **generic** pointer is that, **generic** pointer **cannot** be **dereferenced**. Trying to **dereference** a **generic** pointer will flag a **syntax error**. An example is given below:

```
double x=20;
int a[5]={6,8,3,8,9};
void* p=&x;
cout<<*p<<endl;
p=a;
cout<<*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 **`a`**. When compiling the program statement:

```
cout<<*p<<endl;
```

will flag syntax error.

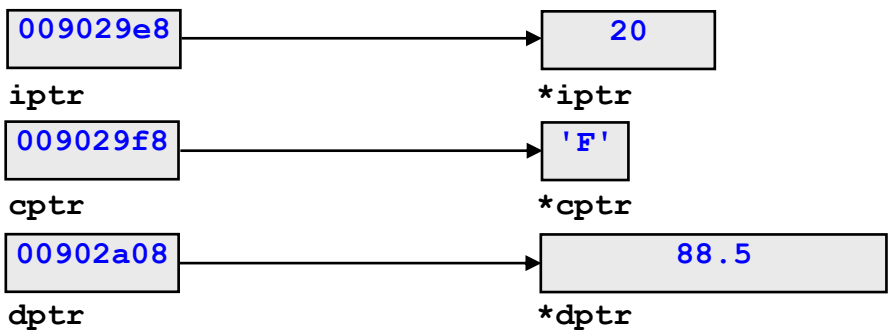
Dynamic Variable

To every variable – variables of fundamental type, array variables, structure variables, objects (variables of the type class) and pointer variables, memory is allocated during the compilation time and memory will be deallocated when a program comes to an end. Variables whose memory is allocated during compilation time is called **static** variable. But there is a special type of variable whose memory is allocated and deallocated during run-time (during program execution) is called **dynamic** variable. Address of a dynamic variable is stored in a pointer variable. Operator **`new`** is used to **allocate** memory dynamically. Operator **`delete`** is used to **deallocate** memory dynamically. It new and delete are **unary** operator and **keywords**. Operators **`new`** and **`delete`** are also called memory management operator.

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 and address is stored in `PtrVar`. Operator **`delete`** deallocates memory pointed to by `PtrVar`.



```
#include<iostream.h>
void main()
{
   double* dp=new double;
   char* cp=new char;
   int* ip=new int;
   *ip=20;    //cin>>*ip;
   *cp='F';   //cin>>*cp;
   *dp=8.5;   //cin>>*dp;
   cout<<"ip="<<ip<<" , *ip="<<*ip<<endl;
   cout<<"cp="<<(void*)cp<<" , *cp="
       <<*cp<<endl;
   cout<<"dp="<<dp<<" , *dp="<<*dp<<endl;
   delete ip;
   delete cp;
   delete dp;
   cout<<"*ip="<<*ip<<endl;
   cout<<"*cp="<<*cp<<endl;
   cout<<"*dp="<<*dp<<endl;
}
```

Pointer variable **ip** points to ***ip**, ***ip** is the dynamic variable (memory is allocated to ***ip** during run-time using operator **new**. Value can be stored in ***ip** either by using assignment operator (=) or taking input from keyboard by using **cin**. Pointer variable **cp** points to ***cp**, ***cp** is the dynamic variable. Pointer variable **dp** points to ***dp**, ***dp** is the dynamic variable. To display address stored in **cp**, **cp** is type casted to **void***. Operator **delete ip** deallocates ***ip**, **delete cp;** deallocates ***cp** and **delete dp;** deallocates ***dp**. Displaying ***ip**, ***cp** and ***dp** after deallocation, shows garbage values.

Running of the program produces following output:

```
ip=0x00902a08 , *ip=20
cp=0x009029f8 , *cp=F
dp=0x009029e8 , *dp=8.5
*ip=4241876
*cp=┕
*dp=1.86082e-307
```

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:

```
double* dp=new double (8.5);
char* cp=new char ('F');
int* ip=new int (20);
cout<<"ip="<<ip<<" , *ip="<<*ip<<endl;
cout<<"cp="<<(void*)cp<<" , *cp="<<*cp<<endl;
cout<<"dp="<<dptr<<" , *dp="<<*dp<<endl;
delete ip;
delete cp;
delete dp;
```

Running of the program segment will produces following output:

```
ip=0x00902a08 , *ip=20
cp=0x009029f8 , *cp=F
dp=0x009029e8 , *dp=8.5
```

Value that is to be assigned to the newly created memory location is written within a pair of parenthesis. Statement `int* ip=new int (20);` does **three** things:

- Creates a pointer variable `ip`
- Address of dynamic variable is stored in `ip`
- Dynamic variable (newly allocated memory location) is initialized with a value `20`

As mentioned earlier, operators **new** and **delete** can be used derived type (structure type / class type). Examples are given showing usage of **new** and **delete** with pointer to structure type and pointer to class type.

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

Running of the program produces following output:

```
sp  =0x009029e8
Roll=23
Name=Sandip Kr Jain
Mark=88.5
```

```
#include<iostream.h>
class student
{
   int roll;
   char name[20];
   double mark;
   public:
      void assign(int ro, char* na, double ma)
      {
         roll=ro;
         strcpy(name, na);
         marks=ma;
      }
      void display()
      {
         cout<<"Roll="<<roll<<endl<<"Name="<<name<<endl
             <<"Mark="<<mark<<endl;
      }
};
```

```
void main()
{
   student *ptr=new student;
   cout<<"ptr ="<<ptr<<endl;
   ptr->assign(23, "Sandip Kr Jain", 91.5);
   ptr->display();
   delete ptr;
}
```

Running of the program produces following output:

```
ptr =009029e8
Roll=23
Name=Sandip Kr Jain
Mark=91.5
```

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 constant. Any attempt to create an array where array size is a variable, compiler flags syntax error. But with dynamic memory allocation it is possible to create a dynamic array whose size can be decided during run-time. Dynamic array is created during run-time by using the operator **new** and it is deallocated during 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 be a **positive integer constant/variable/expression**. Operator **new** allocates `Size` number of contiguous memory locations and starting address of array is stored in `PtrVar`. Operator **delete** deallocates contiguous memory block pointed to by `PtrVar` (dynamic array name).

Example 1:

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
   int n;
   cout<<"Positive integer? "; 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:

```
Positive integer? 15
14  20  22  26  34  36  37  42  66  67  76  85  90  96  97
```

Example 2:

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
   int n;
   cout<<"Positive integer? "; 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:

```
Positive integer? 10
Displaying sorted array
2.2  2.4  4.1  5  6  6.3  7.3  8.6  8.9  9.2
```

Example 3:

```
#include<iostream.h>
void main()
{
   char* arr=new char[20];
   cout<<"Input a string ? "; cin>>arr;
   cout<<"Inputted string= "<<arr<<endl;
   delete []arr;
}
```

Running of the program produces following output:

```
Input a string ? Friday,Saturday
Inputted string= Friday,Saturday
```

Example 4:

```
#include<iostream.h>
struct student
{
   char name[10];
   double mark;
};
```

```
void main()
{
   cout<<"Positive integer? "; cin>>n;
   student* a=new student [n];
   for (int x=0; x<n; x++)
   {
      cout<<"Name? "; cin>>a[x].name;
      cout<<"Mark? "; cin>>a[x].mark;
   }
   for (int k=1; k<n; k++)
      for (int j=0; j<n-k; j++)
         if (a[j].mark<a[j+1].mark)
         {
            student t=a[j];
            a[j]=a[j+1];
            a[j+1]=t;
         }
   for (int c=0; c<n; c++)
      cout<<a[c].name<<" , "<<a[c].mark<<endl;
   delete []a;
}
```

Running of the program produces following output:

```
Positive integer? 5
Name? Ankita
Mark? 90.5
Name? Biresh
Mark? 78.5
Name? Sooraj
Mark? 93.5
Name? Dahlia
Mark? 88.5
Name? Farida
Mark? 82.5
Displaying array sorted on Marks
Sooraj , 93.5
Ankita , 90.5
Dahlia , 88.5
Farida , 82.5
Biresh , 78.5
```