Pointers in C++

Branch: S6 ECE

Faculty: SREEDIVYA

Pointers

- A pointer is a variable that holds the memory address of another variable of same type
- Derived data type
- A pointer that stores the address of some variable x is said to point to x.
- A pointer is declared by putting a star (or '*') before the variable name.
- To access the value of x, we need to dereference the pointer.

Reference Variable

- A reference is an alias(another name) for another variable.

- In C++, the address of a variable is obtained using **reference operator '&'(or address operator)**

Syntax

```
type &reference_name=variable_name;
```

Dereference operator/Indirection operator(*)

-It is possible to obtain the value of a variable directly from a pointer variable

-The expression *pointer_variable gives the value of the variable

-This evaluation is called dereferencing the pointer.

Need for pointers

- To return more than one value from a function.
- To pass arrays and strings more conveniently from one function to another.
- To manipulate arrays more easily by moving pointers to them.
- To create complex data structures, such as linked lists and binary trees
- For dynamic memory allocation and de-allocation

Sample program using pointers Example1

```
#include<iostream>
using namespace std;
void main()
{
        int a,*ptr;          //declaration of variable a and pointer variable ptr
        ptr=&a;              //initialization of ptr
        cout<<“The address of a:”<<ptr<<“\n”;
}
```

OUTPUT

The address of a: 0x8fb6fff4

Example 2

```
#include<iostream>
using namespace std;
int main()
{
 int n=44,*rn;
 int &rn=n;
 cout<<“Value of variable n=“<<n<<endl;
 cout<<“Value of pointer variable rn=“<<&rn;
}
```

OUTPUT

```
Value of variable n=44
Value of pointer variable rn=0x8fb6fff4
```

Pointer Expressions and Pointer Arithmetic

- A pointer can be incremented(++) or decremented(--)

  int ptr++; or int ++ptr;

  int ptr--; or int - -ptr;
- Any integer can be added or subtracted from a pointer
- One pointer can be subtracted from another

Example

```
#include<iostream>
using namespace std;
void main()
{
        int num[]={56,75,22};
        int *ptr,i;
        cout<<“Array values: “<<“\n”;
```

```
        for(i=0;i<3;i++)
                cout<<num[i]<<"\n";
        ptr=num;
        cout<<"Value of ptr: "<<*ptr<<"\n";
        ptr++;
        cout<<"Value of ptr++: "<<*ptr<<"\n";
        ptr--;
        cout<<"Value of ptr--: "<<*ptr<<"\n";
        ptr=ptr+2;
        cout<<"Value of ptr+2: "<<*ptr<<"\n";
        ptr=ptr-1;
        cout<<"Value of ptr-1: "<<*ptr<<"\n";
}
```

OUTPUT

Array values are:

56

75

22

Value of ptr: 56

Value of ptr++:75

Value of ptr--:56

Value of ptr+2:22

Value of ptr-1:75

Pointers with Arrays and Strings

Differences between pointers and arrays

- Arrays refer to a block of memory space whereas pointers do not refer to any section of memory
- Memory address of array cannot be changed whereas content of pointer variable such as memory addresses can be changed.
- We can declare the pointers to arrays as follows:

```
int *nptr;
nptr=number[0];
```

Program to print the sum of even numbers using pointers to arrays

```
#include<iostream>
using namespace std;
void main()
{
        int numbers[50],*ptr,i,n;
        cout<<“Enter the count: “;
        cin>>n;
        cout<<“\nEnter the numbers ”<<“\n”;
        for(i=0;i<n;i++)
        cin>>numbers[i];
```

```
        ptr=numbers;
        int sum=0;
        for(i=0;i<n;i++)
        {
                if(*ptr%2==0)
                        sum=sum+*ptr;
                ptr++;
        }
        cout<<“\nSum of even numbers : ”<<sum;
}
```

OUTPUT

Enter the count : 5

Enter the numbers

10

16

23

45

34

Sum of even numbers: 60

Array of Pointers

- Represents a collection of addresses
- Save a substantial amount of memory space
- Array of pointers point to an array of data items
- Each element of the pointer array points to an item of the data array
- Data items can be accessed either directly or by dereferencing the elements of pointer array

Example

```
int *inarray[10];
```

Example

```
#include<iostream>
using namespace std;
void main()
{
    int *p = new int[5];
    for (int i = 0; i < 5; i++)
            p[i] = 10 * (i + 1);
    cout << *p << endl;
    cout << *p + 1 << endl;
    cout << *(p + 1) << endl; // similar to p++
    cout << p[2] << endl;
}
```

OUTPUT

10

11

20

30

Pointers to objects

- A pointer can point to an object created by a class.

  eg: item x; // *item* is the class and *x* is the object

  Then,

  item *it_ptr; // *it_ptr* is pointer of type *item*

- Object pointers are useful in creating objects at run time
- Object pointers can be used to access public members of an object

Example

```
#include<iostream>
using namespace std;
class item
{
        int code;
        float price;
 public:
        void getdata( int a, float b)
        {
                code = a;
                price = b;
        }
        void show(void)
        {
                cout<<“Code: ”<<code<<“\n”;
                cout<<“Price: ”<<price<<“\n”;
        }
};
const int size = 2;
```

```
int main()
{
        item *p = new item[size];
        item *d = p;
        int i,x;
        float y;
        for(i=0;i<size;i++)
        {
                cout<<“Enter code and price for item  “ <<i+1;
                cin>>x>>y;
                p->getdata(x,y);
                p++;
        }
```

```
                for(i=0;i<size;i++)
                {
                        cout<<“Item: “<<i+1<<“\n”;
                        d->show();
                        d++;
                }
        return 0;
}
```

OUTPUT

```
Enter code and price for item1 40 500
Enter code and price for item2 50 600
Item: 1
Code: 40
Price: 500
Item: 2
Code: 50
Price: 600
```

this pointer

- C++ uses a unique keyword called **this** to represent an object that invokes a member function
- **this** is a pointer that points to the object for which this function was called.
- **this** pointer is automatically passed to a member function when it is called
- **this** pointer acts as an implicit argument to all the member functions

eg:

```
class ABC
{
        int a;
        - -- -
        ---- -
};
```

The private variable ‘a’ can be used directly inside a member function like,

```
a=123;
```

or

```
this->a = 123;
```

Used in:

- Overloading the operators using member function
- Returning the object it points to (eg: return *this;)
- When local variable's name is same as member's name

Example (name of local variable and member name are same)

```
#include<iostream>
#include<string>
using namespace std;
class Test
{
  int x;
```

```
public:
        void setX (int x)
        {
           this->x = x;
        }
        void print()
         {
           cout << "x = " << x << endl;
        }
};
```

```
int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

OUTPUT

x =20

Dynamic Allocation operators

- Dynamic memory allocation is the process of allocating memory during runtime
- C++ defines two unary operators '**new**' and '**delete**' for this.
- An object can be created using 'new' and destroyed using 'delete' operator
- So a data object created inside a program using 'new' will remain in existence until it is explicitly destroyed by using 'delete'
- If sufficient memory is not available for allocation, 'new' operator returns a null pointer

Dynamic objects

- For object creation using ‘new’, syntax will be,

  datatype pointer_variable = new datatype;

  The **new** operator allocates sufficient **memory** to hold a data object of that datatype and returns the address of the object . The **pointer_variable** holds the **address** of the memory space allocated.

  eg: int *p = new int;

  float *q = new float;

- We can also initialize a memory using new operator

  Syntax is:

  pointer_variable = new datatype(value);

  Example:

  int *p = new int(25);

- For arrays,

  Syntax is:

  pointer_variable = new datatype[size];

  Example:

  int *p= new int[10];

- When a data object is no longer needed it is destroyed to release the memory space for reuse and it is done using '**delete**' operator.

Syntax:

```
delete pointer_variable;
delete [size] pointer_variable;      //For array
```

Example:

```
delete p;
delete q;
```

where p and are pointer variables

```
delete []p;        //deletes the entire array pointed by p
```

Advantages of new over malloc

- Automatically computes the size of data objects. No need to use the operator sizeof()
- Automatically returns the correct pointer type. No need to use typecasting
- It is possible to initialize the object while creating the memory space
- Both ‘new’ and ‘delete’ operator can be overloaded.