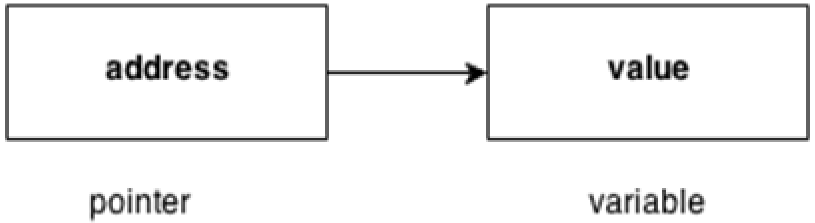
Dr. D. Y. Patil Pratishthan's
Institute for Advanced Computing and Software Development

Sub-c++
Day3

C++ pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

C++ Array of Pointers

- Array and pointers are closely related to each other.
- In C++, the name of an array is considered às a pointer, i.e., the name of an array contains the address of an element.
- C++ considers the array name as the address of the first element.

```
int *ptr[5];        // array of 5 integer pointer.
```

we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

```
int a; // variable declaration.
ptr[2] = &a;
```

Array of Pointer to Strings

```
int main()
{
  const char *names[5] = {"john",
                          "Peter",
                          "Marco",
                          "Devin",
                          "Ronan"};
for(int i=0;i<5;i++)
    {
        cout << names[i] << endl;
    }
    return 0;
}
```

C++ Void Pointer

A void pointer is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

void *ptr;

In C++, we cannot assign the address of a variable to the variable of a different data type. Consider the following example:

int *ptr; // integer p

float a=10.2; // floating variable initialization

ptr= &a; // This statement throws an error.

C++ has overcome the above problem by using the C++ void pointer as a void pointer can hold the address of any data type.

```
 void *ptr;   // void pointer declaration
  int a=9;   // integer variable initialization
  ptr=&a;   // storing the address of 'a' variable in a void pointer variable
```

Difference between void pointer in C and C++-
-In C, we can assign the void pointer to any other pointer type without any typecasting,
 -whereas in C++, we need to typecast when we assign the void pointer type to any other pointer type.
.

Memory Management in c++

What is Memory Management?
Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

C++ also supports malloc() and calloc() functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory dynamically.

New operator
A **new** operator is used to create the object while a **delete** operator is used to delete the object.

Advantages of the new operator

The following are the advantages of the new operator over malloc() function:

It does not use the sizeof() operator as it automatically computes the size of the data object.

It automatically returns the correct data type pointer, so it does not need to use the typecasting.

Like other operators, the new and delete operator can also be overloaded.

It also allows you to initialize the data object while creating the memory space for the object.

Differences between the malloc() and new

-The new operator constructs an object, i.e., it calls the constructor to initialize an object
while **malloc()** function does not call the constructor.
-The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object.
This is the biggest difference between the malloc() and new.

-The new is an operator, while malloc() is a predefined function in the stdlib header file.

The operator new can be overloaded while the malloc() function cannot be overloaded.
If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer

Continue...

In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.

In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the memory.

Delete operator

-It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically.

- This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax-

delete pointer_name ;

Points to remember-

-It is either used to delete the array or non-array objects which are allocated by using the new keyword.

-To delete the array or non-array object, we use delete[] and delete operator, respectively.

-The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap

-It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

Class

- A template for creating similar objects.
- Maps real world entities into classes through data members and member functions.
- A user defined type.
- An object is an instance of a class.
- By writing a class and creating objects of that class, one can map two concepts of object model, abstraction and encapsulation, into software domain.

Class Components

- A class declaration consists of following components
    - Access specifiers: restrict access of class members
        - `private`
        - `protected`
        - `public`
    - Data members
    - Member functions
        - Constructors
        - Destructors
        - Ordinary member functions

- Programs related to class ,object, member function,data members and access specifiers.

Constructor

- Special member function of the class with same name as its class name.
  - Used to initialize attributes of an object
- Implicitly called when objects are created.
- Without any input parameter is no-argument constructor.
- Rules for implementing constructor:
  - No return type for constructor. Not even void.
  - Multiple constructors can be written - different number, types and order of parameters.
  - Must have same name as that of the class.

Constructors in `class cDate`

```
class cDate
{
    ...
public:
//No-argument Constructor
    cDate()
    {
        mDay = 1;
        mMonth = 1;
        mYear = 2000;
    }
//Parameterized Constructor
    cDate(int d, int m, int y)
    {
        mDate = d;
        mMonth = m;
        mYear = y;
    }
```

```
int main()
{
    cDate d1;        .
    cDate d2(25, 8,
2014);
    return 0;
}
```

Calls no-argumet constructor

Calls parameterized constructor.

Copy Constructor

Copy Constructors is a type of constructor which is used to create a copy of an already existing object of a class type.

It is usually of the form X (X&), where X is the class name.

The compiler provides a default Copy Constructor to all the classes.

used to copy data of one object to another.

Implicitly-declared copy constructor

If no user-defined copy constructors are provided for a class type the compiler will always declare a copy constructor as a non-explicit inline public member of its class.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
   . . . .

}
```

e.g.

```
class A
{
   A(A &x) //  copy constructor.
  {
     // copyconstructor.
  }
}
```

//gives call to copy constructor

Calling copy constructor as-

- A a2(a1);
- A a2 = a1;

a1 initialises the a2 object.

```
Class A
{
    int n;
    A(int n1 = 1) {
         n=n1;
    }
    A(const A& a)  {
         n=a.n;
    } // user-defined copy constructor
};
int main()
{
    A a1(7);
    A a2(a1); // calls the copy ctor
}
```

Types of Member Functions

- Following are the types of member functions of a class:
    - **Mutator** - Changes contents of instance members
    - **Accessor** - Accesses instance members
    - **Facilitator** - Helps to view the values of attributes of object
    - **Helper** - A private function, accessed from public member functions of same class to help in their implementation

`class cDate`: Member Functions

```
class cDate
{
    ...
public:
    void display(); //Facilitator Function
    int getDay(void);   //Accessor Function
    void setDay(int);   //Mutator Function
    ...
};
```

```
//Function definition of display member function
void cDate :: display()
{
  cout<<"Date:"<<mDay<<"/"<<mMonth<<"/"<<
mYear<<endl;
}
```

Invoking a Member Function

```
int main()
{
    cDate d1(2,3,4);
    cDate d2(5,6,7);
    d1.display();
    d2.display();
    return 0;
}
```

`this` Keyword

- **`this`** is a keyword in C++.
- It is a pointer
- **`this`** always holds a reference of an object which invokes the member function.
    - **`this`** points to an individual object.
- It is a hidden parameter that is passed to every class member function.

Using `this` Keyword

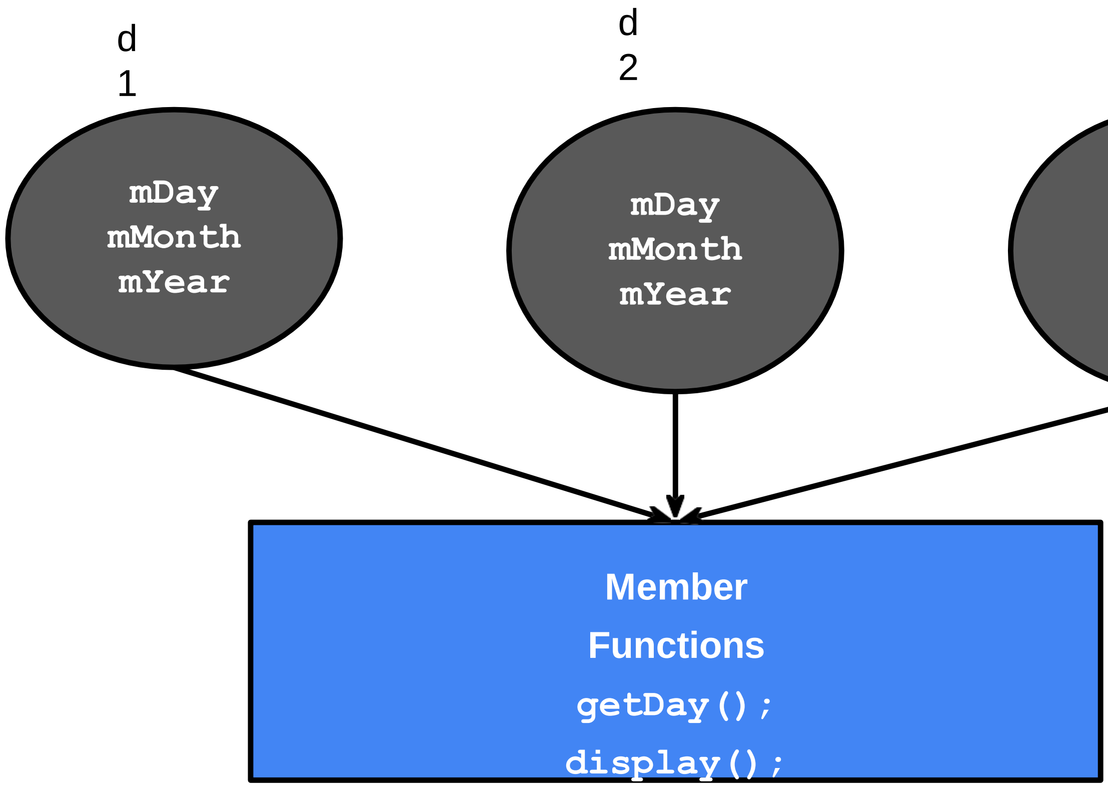
```
int cDate::getDay(void)
{
     return this->mDay;
     //OR return mDay;
}
```

```
void cDate::setDay(int d)
{
     this->mDay = d;
}
```

Access current object

```
int main()
{
  cDate d1(4,5,6), d2;
  int day = d1.getDay();
  cout<< "Day is = " << day;      // displays 4
  d2.setDay(5);      // sets day of d2 to 5
  return 0;
}
```

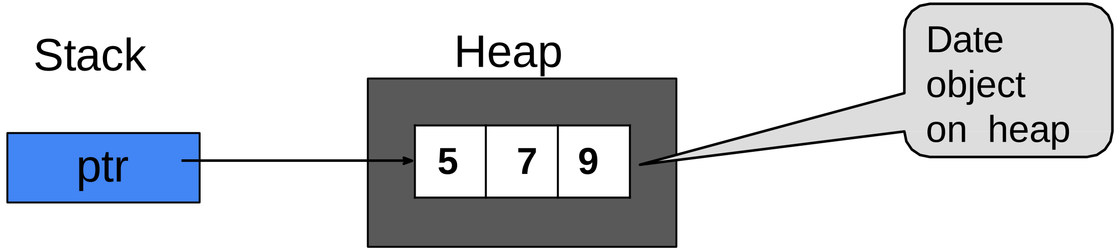
Memory Allocation to Objects



Code
segment

Creating an Object on Heap

```
int main()
{
    cDate* ptr = new cDate (5,7,9);
    ...
    delete ptr ;
    return 0;
}
```



Creating `const` Objects

- To create a constant object use `const` keyword:

```
const cDate d1(2,3,4);     // statement in main
```

- `const` objects invoke `const` member functions only.
- `const` functions are ‘read only’ functions.

`Const` member function

- Const function can be called on any type of object, const object as well as non-const objects.
- object is declared as const, it needs to be initialized at the time of declaration.
- not allow to modify the object on which functions are called.
- Accidental changes to objects are avoided.

`const` Member Functions

```
//In Class Declaration
class cDate
{
    ...
public:
    ...
    int getDay(void) const;
    ...
};

//Function Definition
int cDate::getDay(void)const
{
    return this□ mDay;
}
```

```
int main()
{
  const cDate d1(5,7,9);
  int day = d1.getDay();
  cout<<“Day is = ”
   <<day;

  return 0;
}
```

Static Variables

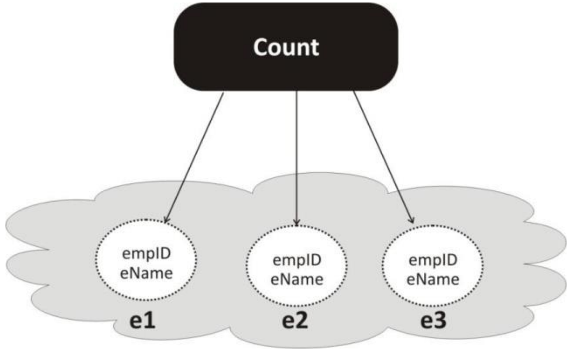
- Some characteristics or behaviors belong to the class rather than a specific instance
  - `interestRate`, `CalculateInterest` method for a `SavingsAccount` class
  - `count` variable in `Employee` to automatically generate employee id
- Such data members are static for all instances
  - Change in static variable value affects all instances
  - Also known as class variable.

Application

.To keep track how many objects created

Static Variables in Memory

```
class Employee
{
   int empId;
   int eName;
   static int count;
}
```



- Data to be shared by all objects is stored in static data members.
- Only a single copy exists.
- Class scope and lifetime is for entire program.
- How can they be accessed?

Static Member Functions

- Can access static data members only.
- Invoked using class name as:

```
class name :: functionName();
```

- **this** pointer is never passed to a static member function.

```
public class Employee
{
   . . .
   static int count;
   static int showCount()
   {
      return count;
   }
}
```

```
main()
{
   int number =
   Employee::showCount();
   cout<< "Number
   employees are:" <<
   number;
}
```