

# C++ Programming

Trainer : Rajiv K

Email: [rajiv.kamune@sunbeaminfo.com](mailto:rajiv.kamune@sunbeaminfo.com)



# Agenda

---

- **dynamic memory allocation**
- **Copy constructor**
- **shallow copy**
- **deep copy**
- **Exception Handling**
- **template**



# dynamic memory allocation

- Dynamic memory allocation in C/C++ refers to performing **memory allocation manually** by a **programmer**.
- Dynamically allocated memory is allocated on **Heap**,
- it is the programmer's responsibility to deallocate memory when no longer needed.
- In C++, to **allocate memory dynamically** we should use **new operator** and to **deallocate that memory** we should use **delete operator**.
- If new operator fails to allocate memory then C++ runtime **implicitly throws bad\_alloc exception**.
- We can also **initialize** the memory for **built-in data types** using a new operators as well as **user defined data types** including **structure** and **class**, a constructor is get called.

MEMORY FROM	:HEAP SECTION.
TO ALLOCATE MEMORY	:NEW OPERATOR
TO DEALLOCATE MEMORY	:DELETE OPERATOR
WHEN NEW OPERATOR FAILS:	THROWS BAD_ALLOC EXCEPTION



# new operators

## new operator:

- The new operator denotes a **request** for **memory allocation**.
- A **new** operator **allocate the memory** and **returns** the **address** of the newly allocated and initialized memory **to the pointer variable**.

## Syntax to use new operator:

data-type pointer-variable= new data-type;

## Example:

**int\* ptr=new int;**

### ALLOCATION:

```
<return type> * p = new <return type>{DECLARATION}  
int * p = new int(10);{INITIALIZATION}
```

### DEALLOCATION:

```
delete p;
```

### FOR ARRAY

#### ALLOCATION:

```
<return type> * arr = new int [SIZE];
```

#### DEALLOCATION:

```
delete [ ] arr;
```

### FOR OBJECT:

#### ALLOCATION:

```
<class name> * ptr = new <class name>
```

#### DEALLOCATION:

```
delete ptr;
```



# Examples

```
Point* ptr = new Point;    //parameterless constructor will call
```

```
Point* ptr = new Point(); //parameterless constructor will call  
                          //Recommended
```

```
Point* ptr = new Point( 10, 20 ); //parameterized constructor will call
```

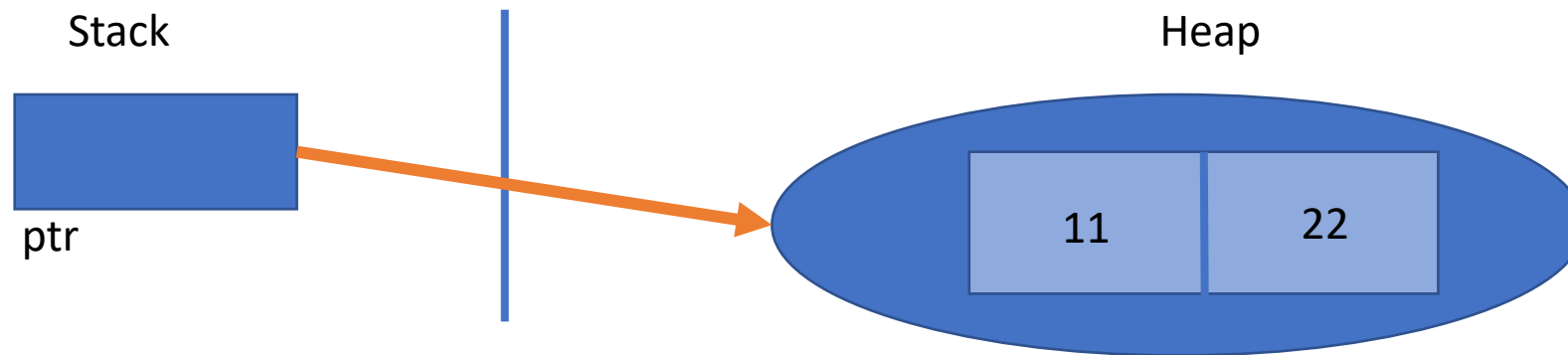
```
delete ptr; //destructor will call
```



# Heap Based object

```
Point *ptr=new Point(11,22) ;  
delete ptr;
```

- By using new we are allocating dynamic memory for complex class object .
- Object get created on heap section hence this object is call Heap based or dynamic object.
- ptr is point type pointer which is holding the address of that dynamic object.



# memory block(Array) memory allocation using new

- To allocate the dynamic memory for **5 continuous** of type double use following example to under stand it better.
- `double* ptr=new double[5];`
- It would returns a pointer to the first element of the sequence.
- In above diagram ptr[0] would give first element, ptr[1] would give second element so on and so forth.



# delete operator

- In C++ programmer can **delete** the **dynamically allocated memory** using **delete operator**.

**Syntax:**

**delete pointer-variable;**

**Example:**

```
int * ptr=new int; //Dynamic memory allocation
```

```
delete ptr;           //deleting the dynamically allocated  
                      //memory using delete operator
```

```
int * pqr=new int[5]; // Dynamically allocating the memory for block
```

```
delete[ ] pqr; // deleting the block of memory which is been  
              //allocated the memory dynamically.
```





# delete operator

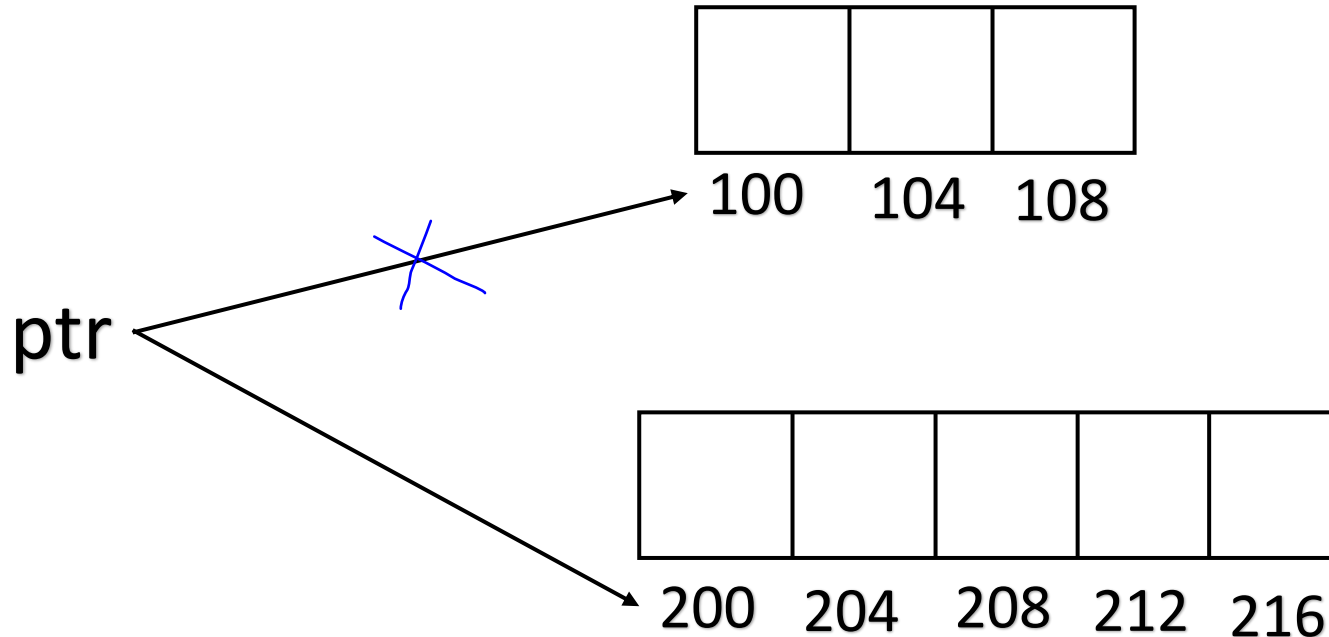
- **Memory leakage:** If we reserve space in memory but no pointer is pointing to it then wastage of such memory is called **memory leakage**.

- **Example:**

```
int* ptr = new int[3];
```

```
ptr = new int[5];
```

//this statement definitely loose 12 bytes.



# Copy Constructor

- A Copy constructor is parameterized constructor of the class used to declare and initialize an object from another object.
- It takes single parameter of same type as reference.
- If developer **dose not write the copy constructor within the class** then in that case **compiler provides the copy constructor** when ever it is required.

- **Syntax:**

```
class_name(const class_name & variable_name)           //Copy constructor
{
    //Body of copy constructor
}
```



# Copy Constructor

- **Copy Constructor Gets called in following Conditions.**

1. If we pass object to the function as value.
2. If we return object from function as value.
3. If we initialize object from existing object.
4. If we throw object
5. If we catch object

- Type of Copy constructor

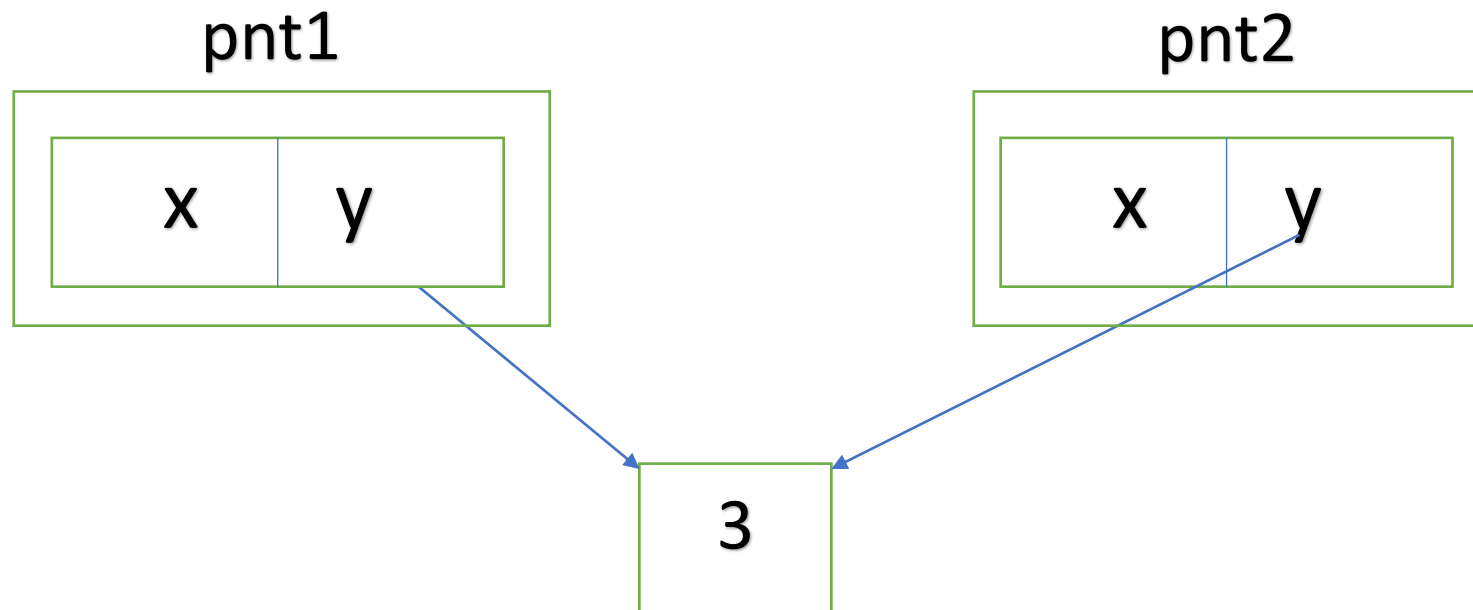
1. **Default Copy Constructor** : When user/programmer does not provide the copy constructor in such case compiler provides the default copy constructor. **SHALLOW COPY**  
`<constructor> p2 = p1;`
2. **User defined Copy Constructor** : When user/programmer provides the copy constructor.  
**DEEP COPY** `<constructor>(<classname> & p1)`  
ex:  
`Point(Point & p1)`



# Shallow Copy

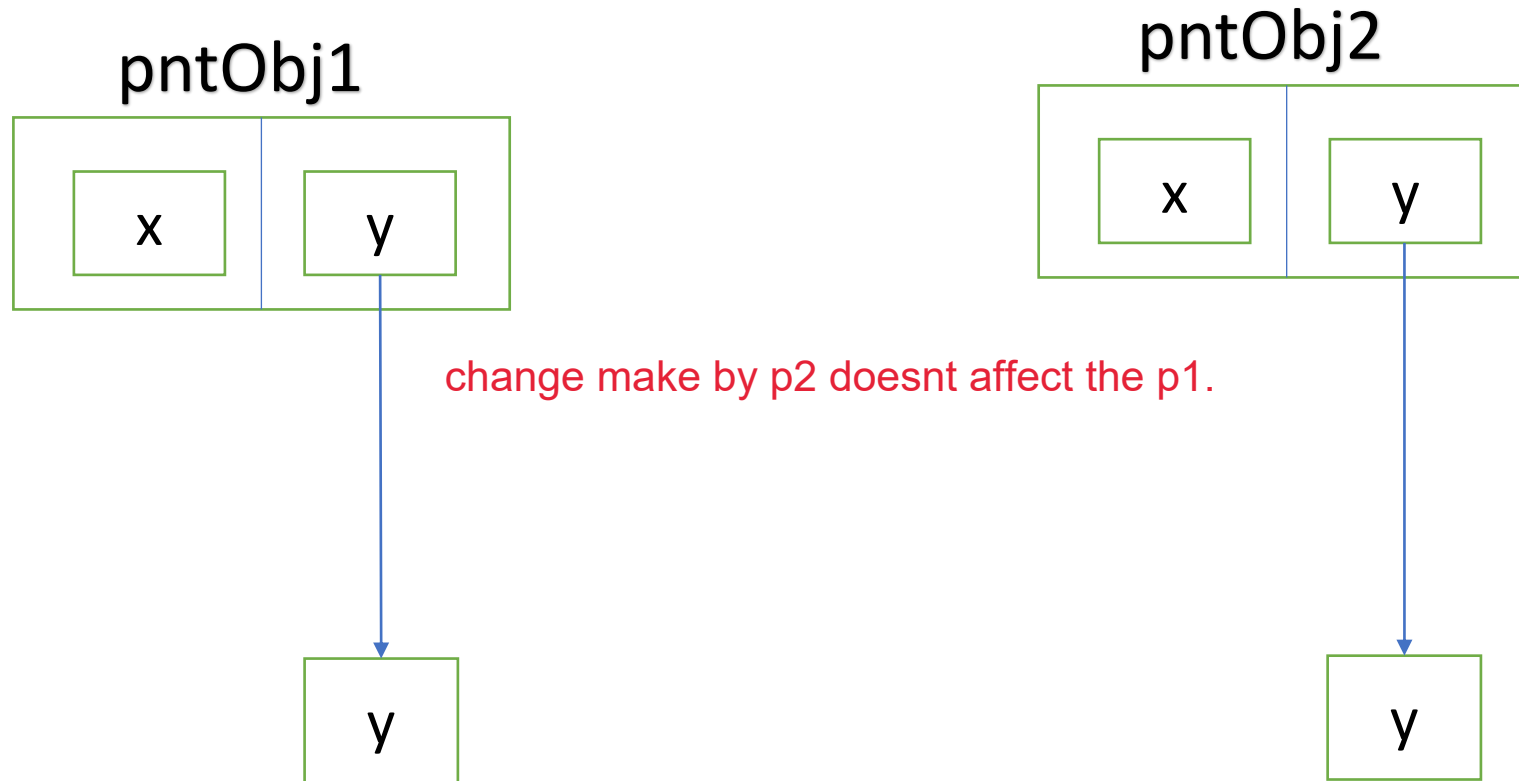
- Shallow copy is a copy of an object by coping data of all the **data members** it is also called as **bit-wise copy**.
- When programmer dose not provide copy constructor then **compiler** provides default copy constructor.
- The default constructor creates the exact copy or **shallow copy** of the existing object.

IF WE MAKE CHANGES BY P2 then THERE WILL BE CHANGE IN THE VALUE OF P1 also.



# Deep Copy

- In case of deep copy memory is dynamically gets allocated for the copy and then only values are copied.
- Deep copy is also called as member wise copy.
- Deep copy requires user/programmer defined copy constructor.



# Difference between malloc and new

## new

new is an operator.

new returns a typecasted operator, so no need to do explicit typecasting.

We must mention the datatype while allocating the memory with new.

When memory is allocated with new, constructor gets called for the object.

## malloc

malloc is a function.

malloc returns void pointer, malloc returns void pointer, cast it explicitly, before use.

malloc accepts only the exact no. of bytes required, so no need to mention datatype.

When memory gets allocated by malloc function, constructor function does not gets called.



# Difference between malloc and new

## new

Memory allocated by new is released by the operator delete.

Destructor is called when memory is released with delete.

To release memory for array syntax is delete[ ]

In case new fails to allocate memory, it raises a Run time exception called as bad\_alloc.

## malloc

Memory allocated by malloc is released by function free.

Destructor is NOT called when the memory is released with free.

To release memory for array syntax is free( ptr );

In case malloc fails to allocate memory it returns a NULL.



# Exception Handling

- One of the **advantages of C++ over C** is Exception Handling. Exceptions are **runtime anomalies/error** or **abnormal conditions** that a program encounters during its execution.
- Exception is an **event or object** which is thrown at **runtime**.
- **Exception Handling** in C++ is a **process to handle runtime errors**. We perform exception handling so the **normal flow** of the application can be **maintained even after runtime errors**.
- Exception need to be handled other wise it would terminate the program abnormally.





# Exception Handling:

- To create an application, most of the times we take help of operating system resources. Following are some the resources we use frequently.
  - 1. Memory
  - 2. File
  - 3. Socket etc.
- In C++ we can handle exceptions with the help of following keywords:
  - 1.try
  - 2.catch
  - 3.throw

} BLOCK  
KEYWORD



# Exception Handling:

- 1. try block/ try handler:

- if we want to **inspect** group of statements for **exception** then we should put such statement **inside try block** or **try handler**.

- Syntax:

```
try{
```

```
    code statement 1;
```

```
    code statement 2;
```

```
    code statement 3;
```

```
}
```

} Code which may possibly generate the exception



# Exception Handling:

- 2. catch block/catch handler:

- If we want to **use try** block then it is **necessary** to provide at **least one catch** block.
- To **handle exception** thrown **from try block** we should **use catch block** or catch handler. Single try block may have multiple catch block but it must have at least one catch block.

- Syntax:

```
try{  
    code statement 1;  
    code statement 2;  
    .....  
    code statement n;  
}  
catch()  
{  
    //Block of code to handle the error  
}
```

**//Code which may possibly generate the exception**



# Exception Handling

- **Generic Catch block/Generic catch handler:**

- A catch block which can handle **all types of exception** is called generic catch block.
- In case if you don't have an idea which error will be thrown by the block of code then in such case you need to give generic block for the try.

- **Syntax:**

```
try{  
    code statement 1;  
    code statement 2;  
    .....  
    code statement n;  
}  
catch(...) GENERIC CATCH BLOCK  
{  
    //Block of code to handle the error  
}
```

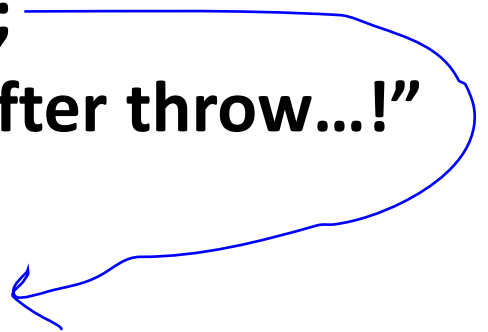
} **//Code which may possibly generate the unknown exception**



# Exception Handling

- **3. throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- Example:

```
try
{
    cout<<"Before throw...!";
    throw 10;
    cout<<"After throw...!"
}
Catch(int n)
{
    cout<<"Error code:"<<n;
}
```



# Template

- While calling the function we **pass the data type** too, so that **repetitive code writing can be avoided to reduce the programmers effort.**
- That means we are trying to write the **generic program** with the help of templates.
- **Example:**
- Write a function to **add two int num** and write a code to **add two double num** for this we need to write the **two separate functions** and we may call it as **function overloading** if they are having **same name but different signatures as follows.**

```
Case1:int sum (int num1,int num 2)           //Function to add 2 int nums  
        {return num1+num2;} //business logic
```

```
Case2:double sum(double num1,double num2) //Function to add 2 double nums  
        {return num1+num2;}//business logic
```



# template

- In above code we can see we have written the repetitive code so to avoid this we use templates.
- Template gets resolve a compile time.
- **template and typename or class are the key word** we use to implement the templates.
- Type of templates
  - 1. function template
  - 2. class template
- **Syntax:**
- **template<typename T>**
- **Template<class T>**

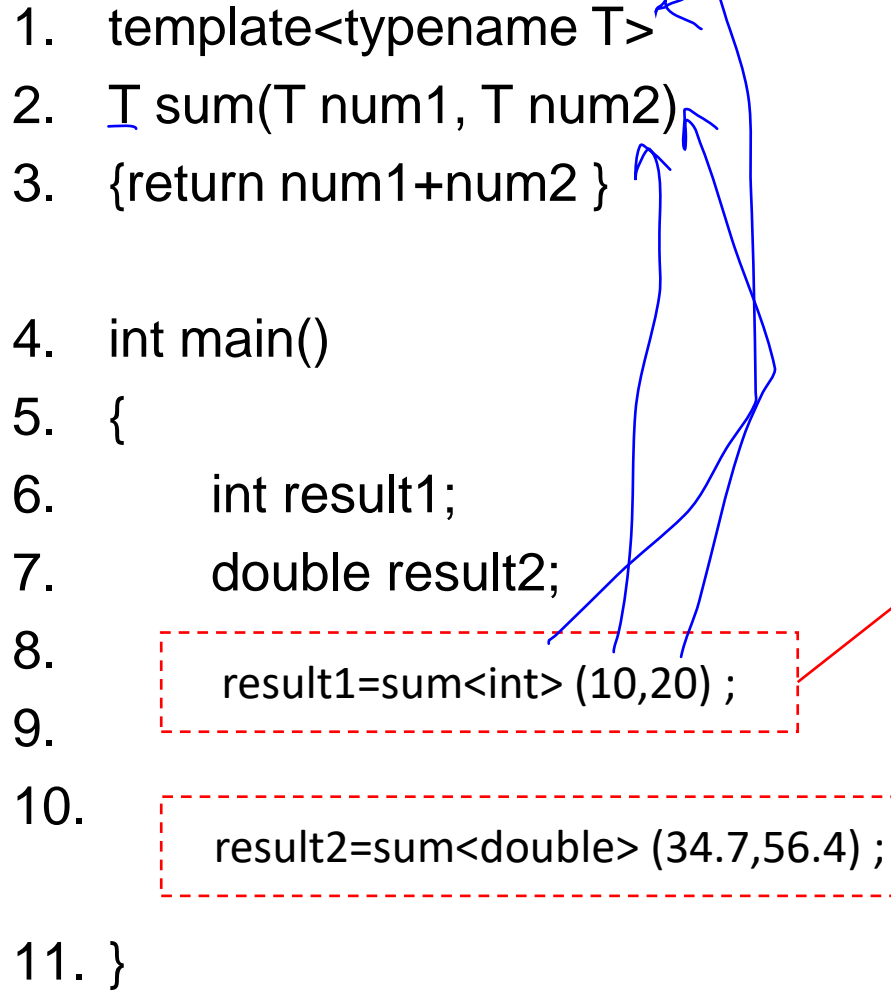


# Template

- **Example: function template**

```
1.  template<typename T>
2.  T sum(T num1, T num2)
3.  {return num1+num2 }

4.  int main()
5.  {
6.      int result1;
7.      double result2;
8.      result1=sum<int> (10,20) ;
9.
10.     result2=sum<double> (34.7,56.4) ;
11. }
```



//compiler will resolve this call  
//as follows  
int sum(int num1,int num2)  
{return num1+num2}

//compiler will resolve this call //as  
follows  
double sum(double num1, double num2)  
{return num1+num2}





---

# Thank You

