→ Dynamic memory allocation :-

Before learning about how dynamic memory allocation works we should learn about how our programme data, codes, variable etc. stored in memory.

So, basically part of memory, where a programme data, codes, variable etc. are stored, devided into 4 parts.

1) machine code / text :-
where your executable code stored

ii) data part :-

when those variables are stored whose scope / life span is for whole programme but not local. ex - static / global variable.

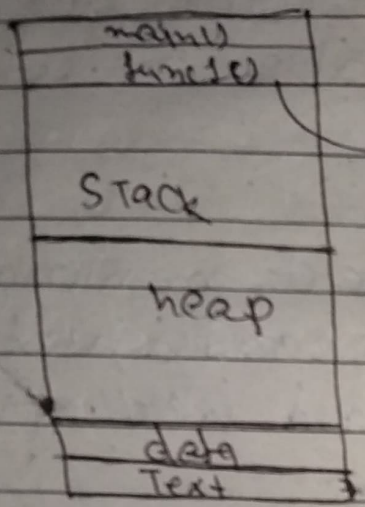
iii) heap :-

Part of memory from where dynami-cally allocated memory alloated.

iv) stack :- Part of memory in which function & it's local variables

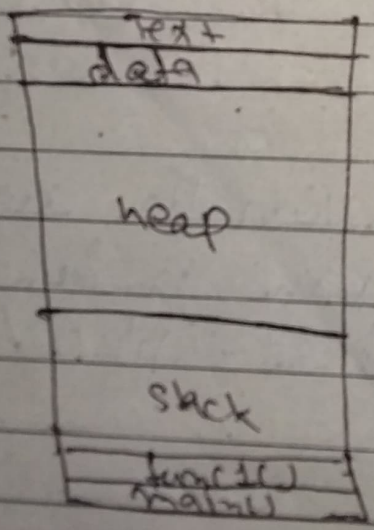
are stored in LIFO (Last In First Out) manner).

We can visually understand it as



→ if func1() call is over it will go out of scope with its local (nonstatic) variables.

OR.



We use 'new' keyword to allocate and 'delete' keyword to delete/free DAM.

After deallocating any dynamic memory using delete, we should initialize that memory with NULL.

ex-

```
int * Ptr = new int[5];
delete[]Ptr;
Ptr = NULL;
```

NOTE:- To avoid memory leak we should always free dynamically allocated memory if it's use is over.

→ Exception handling in C++ :-

For any good programmer it is necessary that his written code works fine for any input and for any exceptional case.

So, we should learn exception handling. So that we can avoid any problem in the output due to some exception in input or some other exception.

Some exceptions are like

- devinding by zero.
- no heap memory available
- accesing array elements outside the allowed index range.
- poping from empty stack or pushing to full stack

etc.

We will learn to handle these cases in exception handling.

try, throw and catch keyword in C++:-

try keyword is used to make a block in which we write those codes who might throw an exception.

throw keyword is used to throw exception using variables/data types.

catch keyword is used to make a block which do some work when get exception thrown by throw keyword of same data type.

Catch block used to handle the exceptions. There can be multiple catch block for different datatype thrown by throw keyword, if one catch block is executed other will not be executed. But there must be one catch block for one thrown datatype. Or we can give a catch block which will execute for any datatype by default.

NOTE :- ~~If is not~~ In multiple function call or for nested function if try block is used in first function

→goes out of scope and if

and exception is thrown by last function then this thrown exception looks for catch block in the same function, if it is not there it passes it to its called and that called function also don't have catch block it passes it to his caller, this process goes scope out goes until suitable catch block not found.

Some examples:-

```
#include<iostream>
using namespace std;

int main(){
    int x, y;
    cin>>x>>y;
    try{
        if(y==0)
            throw 0;
        cout<<"Result is"<<x+y<<endl;
    }
    catch(int x){
        cout<<" error: dividing with zero"<<
                                    endl;
    }
    return 0;
}
```

```
#include<iostream>
using namespace std;

int main(){

    double x,y;
    cin>> x >>y;
try
  {
     if(x==0.0)
        throw 0;
     if(y==0)
        throw string("Y is 0");
     if(x+y>0.0)
        throw (x+y);
  }

catch(int e1){
        cout<<e1;
  }
catch (string &e2){
        cout<< e2<<endl;
   }
catch(...){
     cout<<" x+y"<<;
  }
  cout<<"\n bye.\n";
return 0;
}
```

Catch(...) means catch all, used as default

```
#include<iostream>
using namespace std;

void average(int arr[], int n) throw(string)
{
    if(n==0)
        throw string("Array size is 0");
    for(int i=0;i<n;i++){
        cout<<i+5<<endl;
    }
}

int main(){
    int arr[]={7,8,9,5};
    int n=0;

    try{
        average(arr,n);
    }
    catch (string &e)
    {
        cout<<"e"<<endl;
    }
    cout<<"In bye \n";

    return 0;
}
```

Note:- writing throw(data) with function declaration is very useful to know this function throw which-which type data.

```
#include <iostream>
using namespace std;

void f1() throw(int)
{
    cout << "F1 begins \n";
    throw 100;
    cout << "F1 ends \n";
}

void f2() throw(int){
    cout << "F2 begins \n";
    f1();
    cout << "F2 ends \n";
}

int main(){

try
{
    f2();
}

catch (int x)
{
    cout << "caught exception \n";
}
cout << "Bye ... \n";
return 0;
}
```

NOTE:- To operator overloading to work, at least one of the operands must be an user defined class object.

→ Operator overloading :-

Operator overloading useful when you want an operator to do operation which is not predefined. For example An addition operator (+) used to add two floating point value but you want to add your custum datatypes values, for this you need operator overloading.

We can overloading almost all operators except :, ::, ., ?: and Size of operator.

To do operator overloading we define an operator function which defines all operations that overloaded operator will perform.

Syntax :-

keyword

{{return_type_of_function}} operator {{operator}} (arguments)

{

// Code;

}

ex:- Here is an example of add operator(+) overloading.

```
#include<iostream>
using namespace std;

class vec{

    int real;
    int imaginary;
public:
    vec(){}
    vec(int r, int i): real(r), imaginary(i){}

    void show(){
    cout<<"vec imaginary number is"<<
        real<<"+"<<imaginary<<"i"
                <<endl;
    }

vec operator+(vec m){
    vec temp;
    temp.real = this->real + m.real;
    temp.imaginary = this->imaginary + m.imaginary;
    return temp;
}
};
```

```
int main(){
    vec m1(5, 10), m2(15, 27);
    vec m3 = m1 + m2;
    m3.display();
    return 0;
}
```

NOTE :- here operator function has only one parameter while it is working on two objects. It is because it is member function of a class, in this case first object (here m_1) is treated as class object and second object (here m_2) will be passed to operator function.

But if operator function is a friend function we have to pass two arguments.

- overloading of short hand operators (+=/ -=/*=/!=).

```
#include<iostream>
using namespace std;
class mark{
    int marks;
public:
    mark(){}
    mark(int a){
        marks=a;
    }
    void show()
    { cout<<"marks is:"<<marks<<endl;
    }
    void operator+=(mark &m){
        this->marks=this->marks+m.marks;
    }
};
int main(){
    mark initial(0), math(99), Phy(89);
    initial+=math;
    initial+=Phy;
    initial.show();
    return 0;
}
```

• Overloading • increment / decrement operator in prefix form :- (++x / --x)

As we know increment or decrement operator in prefix form increament the value by one then return the value.

```
#include<iostream>
using namespace std;

class mark{
    int data;
  public:
    mark(){}
    mark(int d){
      data = d;
    }
    void show(){
      cout<<" data is :" << data<<end
    }
    mark operator ++(){
      data += 1;
      return data;
    }
    friend mark operator --(mark &);
};
```

```
mark operator--(mark &m)
{
    m.data -= 1;
    return;
}

void operator ++()
{
    data += 1;
}
friend void operator--(mark &m);
};

Void operator --(mark &m){
    m.data -= 1;
}
int main(){
    mark obj(57);
    Obj.show();
    ++Obj;
    Obj.show();

    --Obj;
    --Obj;
    Obj.show();
return 0;
}
```

There is another methode for operator function

```
mark operator ++( ){
    data += 1;
    return *this;
}
friend mark operator --(mark &);
};

mark operator --(mark &m)
{
    m.data -= 1;
    return m;
}

int main(){
    mark obj(85);

    (++obj) obj.show();
    (++obj).show();
    (--obj).show();

    return 0;
}
```

- overloading increment/decrement operator in postfix form (x++ / x--) :-

As we in postfix form, current value is incremented/decremented by 1 but returned initial value.

Also write to differe Prefix & and postfix operator while overloading extra int datatype passed as parameter to operator function.

```
#include <iostream>
using namespace std;

class mark{
    int mark;
    public:
      mark(){}
      mark(int a)
       {
          mark=a;
       }
      void show()
      {
          cout<<" value of mark is:"<<mark
                                    <<endl;
      }
```

to specify postfix

```
mark operator ++(int)
{
    mark demo(*this);
// this operator is a pointer which contains
// address of current object, by dereferencing
// this we get that object. copy constructor

    mark += 1;
    return demo;
}

friend mark operator --(mark&, int);
};

mark operator --(mark &m, int)
{
    mark demo(m);
    m.mark -= 1;
    return demo;
}

int main()
{
    marks anil(68);
    anil.show();

    (anil++).show();
    anil.show();
```

```
(anil--).show();
anil.show();

return 0;
}
```

→ Overloading of special ~~funct~~ operators [], () and → :-

Main thing to know before overloading these functions are ~~they~~ their operator functions can't be static member function or friend function.

- Overloading operator [] :-

As we know this square bracket operator used with array to traverse/ go to a specific index in that array.

But in this example we will use this operator on an object to get value stored in the array of that object.

Also while overloading we pass an integer to the operator function

which indicates the position/index.

```
#include <iostream>
using namespace std;

class arr{
    int arra[3]={24,56,87};
  public:
    int operator[](int x)
     {
        return arra[x];
     }
};

int main()
 {
    arr obj;
   cout<<obj[0]<<endl;
    cout<<obj[1];
  return 0;
}
```

NOTE:- We don't need to over assignment operator(=) it is automatically overloaded by compiler but coe can also overload it.

- overloading operators → :-

```
#include<iostream>
using namespace std;

class marks{
  int mark;
  public:
    marks(){}
    marks(int m){
      mark = m;
    }
    void print(){
    cout<<" mabs obtained ="<<mark<<endl;
    }
    marks * operator->(){
      return this;
    }
};
int main()
{
    marks ami(100);
    ami -> print();
    return 0;
}
```

NOTE:- By overloading you can't alter the precedence, associativity of operator.

except (=) assignment operator function, other all operator function can be inherited.

- Difference b/w new keyword and operator new and overloading operator new and delete:-

new keyword or new operator is an operator which denotes a request for memory allocation on the heap. If sufficient memory is available, new operator/new keyword initializes the memory and returns the address of newly allocated and initialized memory to the pointer variable. So, when a class object is created using new it invokes the constructor to initialize the memory.

But operator new is function that only allocates memory but not initialize its (i.e, no constructor will be called) but after overloaded new returns then compiler automatically calls the Constructor.

It's possible to overload operator new either globally or for specific class.

If we overload new and delete for specific class it's operator function will be invoked at time of object initialization.

If we overload new and delete for global scope, it's operator function will be invoked whenever new and delete is used in that programme.