**INTERNAL FUNCTIONING**

Member functions arent duplicated. Member functions are created and plced in memory only once when they are defined in the class definition. Data items however will hold different values, so there must be a separate instance of each data item for each object.

**FUNCTIONS**

To Pass by reference any type(primary or objects) you just have to mention & in the function parameters. no need to use \*

void myFunction(int& num){

}

A const member function doesnt modify any of its class’s member data.

class Foo{

int data;

Foo():data(5){}

void constFunction(const DataType& data) const{

data=99; //throws an error

}

}

To make sure that constFunction doesnt change the variables, the arguments passed also are const

We can also return reference variables in a function which can be used on the left hand side. The folowing is the example.

int x;

int& setx();

int main(){

setx()=5;

}

int& setx(){

return x;

}

Inline functions:

Usually, whenever the function call happens, there is some eextra time involved in jumping to the function. instructions for saving registers, instructions for pushing arguments onto the stack in the calling program, removing them form the stack, restoring registers and instruction to return to the calling program. This slows down the program. To save execution time in short functions, you may elect to put the code in funciton body directly inline with the code in the coalling program.

To make a function inline we just need to add the inline keyword before the functin declaration:

inline return\_type function\_type(arguments){

}

Default arguments

Default arguments apply only to trailing arguments. That is, if arguments are missing, it is assumed that the arguments at the end are missing

void myFunction(char= “\*”, int=25);

void myFunction(char ch,int k){

for(int i=0;i<k;i++)

cout<<ch;

}

(or)

void myFunction(char ch= “\*”,int k=25){

}

Such default arguments can be used for constructors too!

**CONSTRUCTORS**

class foo{

private:

int data;

public:

foo():data(0){}

}

constructor():initialization\_list

{

//function body

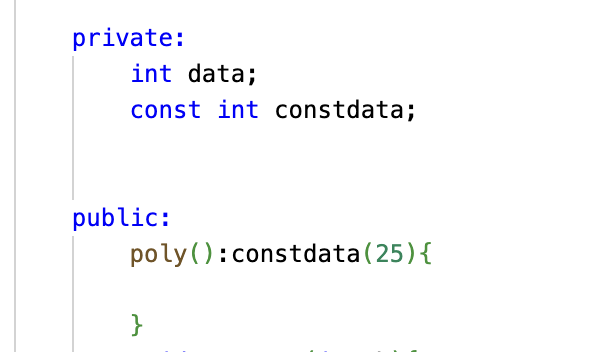
}

This must be the preferred way to write constructor. The member data must be initialized before the constructor executes. You can initialize them in the constructor body too but its a good practice to initialize before the constructor body is executed. Also const member data can onl be initialized there.

If there is no constructor compiler provides with a default constructor. Default constructor doesnt perform any intialization.

However, if there are const or reference data members, they must be explicitly initialized in a constructor. else c++ compiler will throw an error.

When initializing const variable in the constructor, we must initialize in the initialization list. We cant do it in the constructor body. We should do it as shown below:



Destructors dont take any arguments. they have exact same syntax like constructors except that they are preceeded by a ~.

Default Copy constructor

You can initialize an object with another object of same type. You dont have to create a special constructor. Its called the default copy constructor.

class Distance{

int length,width;

Distance(int a,int b):length(a),width(b){}

};

Distance dis1(5,6);

//In both of the following statements a copy constructor is invoked.

Distance dist2(dist1);

Distance dist3=dist1;

Default copy constructor uses member by member copy. It is provided by the compiler.

**Member function declaration outside the class**

We only declare the function in the class but implement/define it outside the class

class Foo{

public:

void foo\_function();

}

void Foo::foo\_function(){

cout<< “we are in foo function”;

}

return\_type class\_name::function\_name(arg1,arg2){

}

Advantages:

1) readability of code, if there are too many member functions declaring some of them outside will help in readability

2) encapsulation: Defining member functions outside the class can help enforce encapsulation by hiding the implementation details of class

3) Reusability: function can be compiled separately from the class allowing it to be used in different parts of the program

4)

**STRUCTS**

The difference between structs and classes is that, in class members are private by default. In structs, members are public by default. If you dont mention any keyword in the class its private.

struct foo{

void func();

private:

int data1;

};

**STATIC**

If a static data item is declared in the class, only one such item is created for entire class. Static variables exist even if there are no objects created. Memory space for static data is assigned only once. Just like the member functions.

class Foo{

private:

static int data;

}

int Foo::data=5;

static data is declared in class definition but it is assigned outside the class definition so that it doesnt violate the idea that a class definition is only a blueprint.

**OPERATOR OVERLOADING**

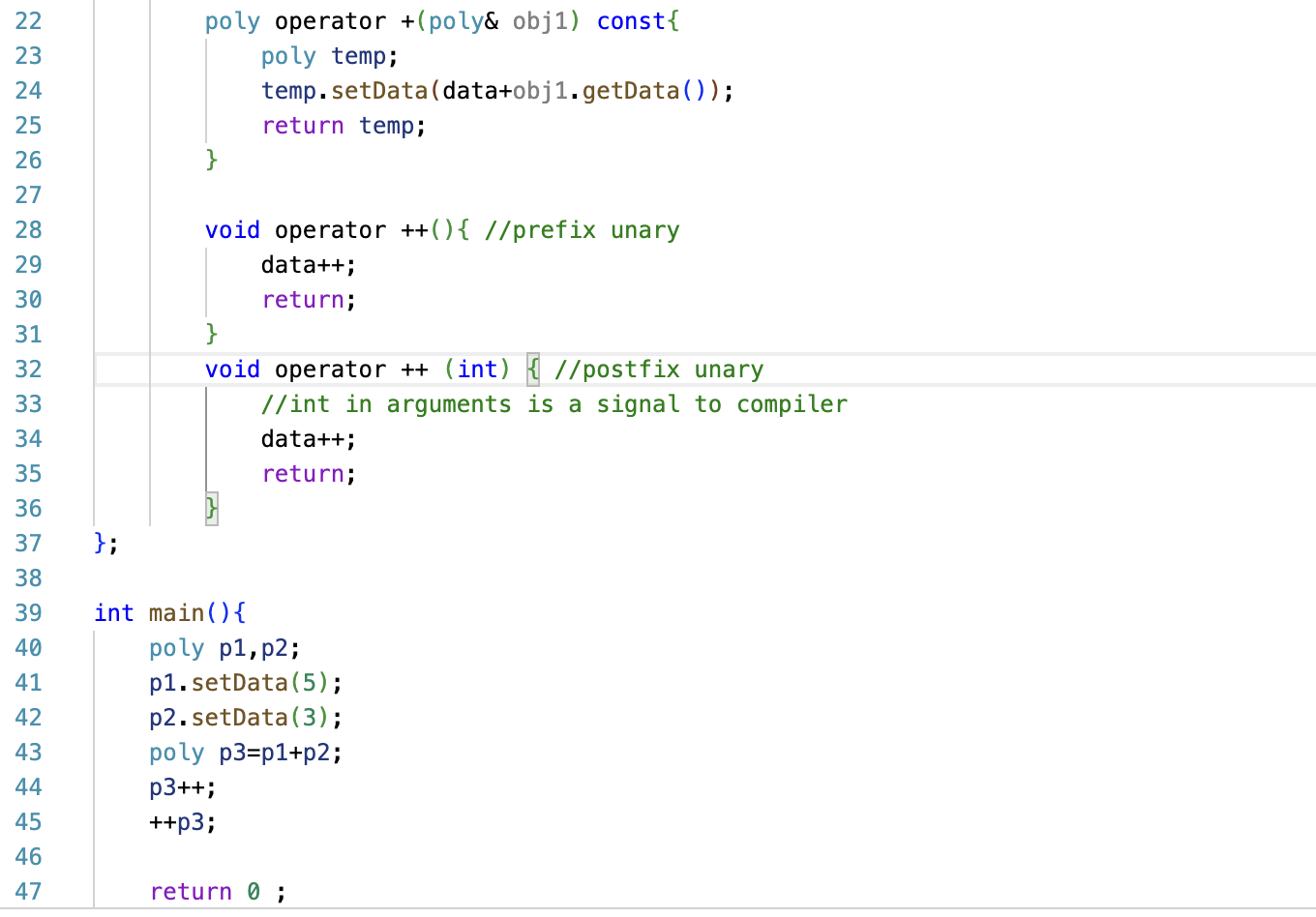
In case of unary operator there are two types: prefix and postfix

In case of binary operator the operator () of the left side is invoked where as the object on the right of the operator is sent as arguments.

In the below example:

p3=p1+p2;

p1’s operator +() is invoked. p2 is sent as argument.



**FUNCTION OVERLOADING**

Number of parameters, type of parameters.

We cant solely change the return type.

The function signature must be different (Signature includes function name and the arguments).

**VARIABLES**

Global variables if not initialized are by default assigned 0. This is in contrast to local variables which when left uninitialized have garbage values. Memory space is set aside for global ariables when the program begins. The storage class is static which means they exist for the life of the program. there is no need to use the keyword static. Global variables are visible in the file in which they are defined starting at the point where they are defined.

There are also static local variables. Static local variables have the visibility/scope of other local variables. However, they are created (memory is assigned) only when that function/block of code is first executed. After it is executed for the first time, it remains until the end of llife of the program. Static local variables are used in cases where functions need to remember a value when it is not being executed, that is in between the function calls. Static variables defined in the function are initialized only once for the program when those functions are called for the first time.

Local variables and functiona rguments are stored on stack where as global and static variables are stored in heap.

Local variables are stored on the stack which grows downward in the memory in descending order. Global variables are stored in the heap which grows upward in the memory in ascending order.

**INHERITANCE**

Base class is the super class. Derived class is the subclass.

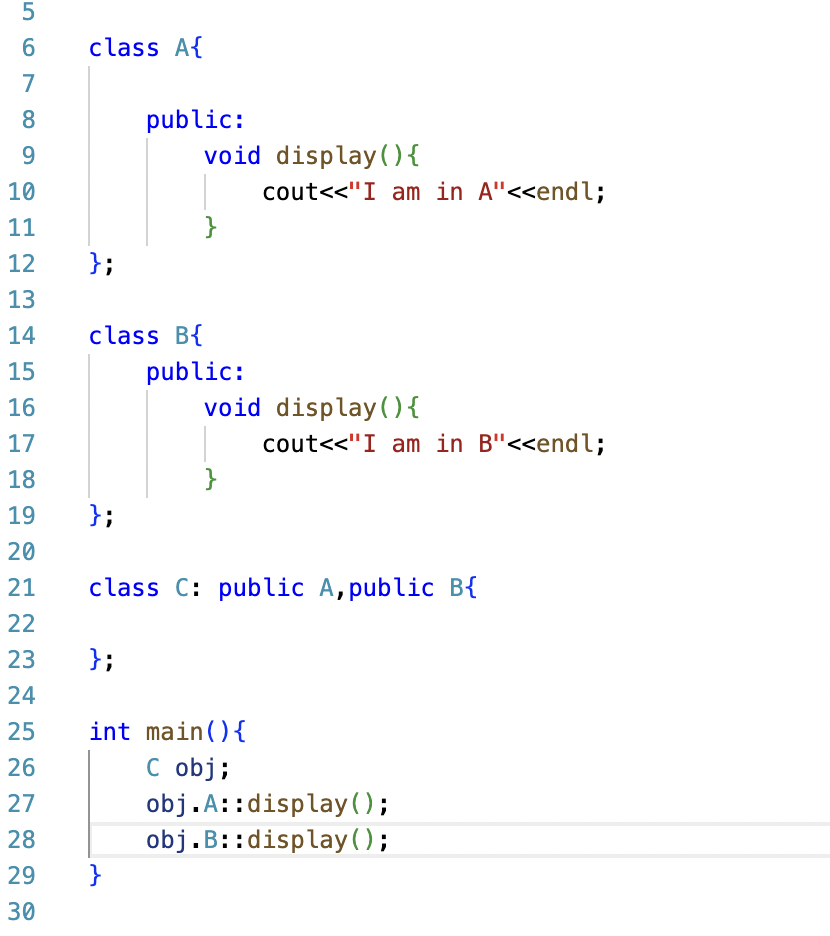
Abstract base classes are the ones in which objects are not created at all. Objects of only derived classes are created.

Using protected access specifier for member data in base class maybe risky because the user must just inherit the class to access the data. Instead keep the base class data as private, and use the scope resolution operator to access base class functions and access the data in the derived class.

The following is the example:



In case of multiple inheritance where two base classes have functions with the same name while derived class not having a function with that name, a scope resolution operator must be used to specify which function is to be used. this is called disambiguation. For example:



**AGGREGATION**

In inheritance, a derived class B *is kind of* a base class A. In aggregation we use *has a* relationship. For example, a library *has a* book. Aggregation is a part-whole relationship. Aggregation might occur in object oriented programming when one object of class A is an attribute of class B.

**COMPOSITION**

Composition is a stronger form of aggregation. It has all the features of aggregation plus two more:

1) the part may belong to only one whole

2) lifetime of the part is the same as the lifetime of the whole

Example: a car is composed of doors, the doors cant belong to some other car and they are born and die along with the car. A room is composed of a floor, ceilig and walls. While aggregation is a “has a”relationship, composition is a “consists of” relationship.

**POINTERS**

int var=2;

cout<<&var<<endl;

this gives the address of var.

How to store this address in a variable?

We use a pointer variable.

int \*ptr=&var;

The asterisk is a part of the variable, not part of the type. That is, it is a part of ptr, not part of int.

int \***ptr1,\***ptr2,\*ptr3;

when we just print ptr, it gives the address of the variable.

But when we print \*ptr, it will give the value held by the variable.

\* is called the dereference operator. It means the value of the variable pointed by.

The asterisk used in the derereference operator means *value of variable pointed to by.*(cout<<\*ptr)

The asterisk used in the declaration of pointer means *pointer to int/data\_type* (int \*ptr)

There is a general purpose pointer that can point to any type of data.

void \*ptr;

This ptr can be used to point to functions which are independent of the data type.

int k=5;

float p=25.65;

void \*ptr;

ptr=&k;

ptr=&float;

The above code works.

In arrays, the name of the array is its address.

int arr[5]={1,2,3,4,5};

arr is the address here.

for(int i=0;i<5;i++)

cout<<\*(arr+i)

We cant increment an address. For example we cant increment arr.

cout<<\*(arr++);

This is wrong.

It is equivalent to saying 7++ where 7 is a constant. However we can increment a pointer variable that stores this address.

int\* ptr=arr;

for(int i=0;i<5;i++)

cout<<\*(ptr++);

new

new returns a pointer to an area of memory that holds data.

char \*str=new char[50];

delete[] str;

char \*ptr=&var;

delete ptr;

Pointers to objects:

class Distance{

----

---

---

};

Distance \*distptr;

distptr=new Distance;

Distance \*disptr=new Distance[50];

Distance \*distptr[50]; //this is an array of pointers to objects

Distance dist[50]; //this is an array of objects.

When we create a pointer for an object there are different ways to access member data and functions.

class Foo{

public:

int data;

void display(){}

}

Foo \*ptr=new Foo;

(\*ptr).data=5;

(\*ptr).display();

(or)

ptr->data=5;

ptr->display();

-> this is the membership access operator.

**STRINGS**

Strings terminate with null character \0

char str1[]= “hi this is anirudh”;

char \*str2= “hi this is anirudh”;

char str3[]={‘a’, ‘b’ , ‘c’};

char \*str6=new char[5];

str1++; //cant do this because str1 is a constant

str2++; //can do this because str2 is a pointer

cout<<str2; //this will give i this is anirudhkaluri

char str4[50];

cin>>str4; //lets enter -> Hi this is anirudhkaluri@gmail.com

cout<<str4; //this will output-> Hi

cin will consider a space to be terminating character. so we use cin.get();

cin.get(str4,50);// this will take string including psaces.

//but this wont allow multiple lines to be read

cin.get(str4,50 ‘$’);

//this will read until it encounters $ character. by default if there is no third argument, the terminating character will be \n

//the following wont work because you will get a bus error. memory size should be specified

char \*str;

cin>>str;

**VIRTUAL FUNCTIONS**

**POSIX**