Scope Resolution Operator

1) C++ provides :: SRO

2) It has fellowing uses

1) To access members of a class

For eg :- While defining Member fun of a class, :: is used on fun def

```
Void student :: set (int x, int y, int));
```

2) To access global var

There can be two var with same name - one local & other global.

- In that case, the global var becomes hidden due to local var.
- To access global var, use ::

eg

```
    int a=10;
    int main()
    {
          int a=20;
          Cout<<a; // Local var =20
          Cout<<::a; // global var =20

    }
```

3) To access members of a namespace

:: can be used to access member of a namespace.

for eg to use namespace std, use

```
std:: cout<< 10;
```

Pointer & Reference in C++

1) In C++ you can create pointer in the same manner as in C

2) eg
```
int a = 10;
int *b = &a;
*b = 20; // Change a too
```

Note: This ~~sa~~ line can be written as
```
int *b;
b = &a;
```

```
cout<<a<<b<<*b; // 20, 0Xaab10f, 20.
```

a ↓ Print value of a

b ↓ Print ptr. ↓ Print add of a

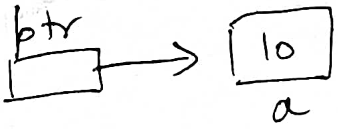
*b ↓ Print value at ptr ↓ Print value of a.

How to use pointers

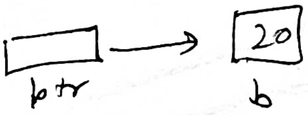
~~1) To point ptr to any variable use this~~ code.

1) To declare a ptr, use this code;
```
int *ptr;
```

2) To point ptr to a var, use this code;
```
ptr = &a;
```

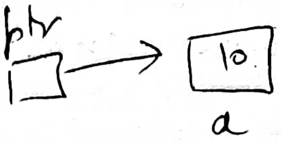


```
ptr = &b;
```



3) To print the value of variable use this code:

```
cout<< *ptr;
// o/p =10.
```



Note:

~~ptr~~ • The value of ptr can be accessed by using this code:

```
cout<< ptr; // This will print
            the add stored
            in ptr.
```

• The value of ptr is ~~surely~~ ~~the~~ the address stored in ptr.

• Difference b/w

*p → print value of a

p → print add of a.

References in C++

Q What are references?

A . C++ provide a feature called reference.

- Ref r an alternative to ptr.
- A ref is another name for ~~an~~ ~~alreado~~ a var.
- Ref r declared ~~like~~ using & operator like this

```
int a =10;
int &ref = a;
```

it means ref is a referenceto a

or " " ref is another name for a

$\boxed{10.}$

$\underbrace{a, ref}$

Same var will have two names.

- If you change ref, value of a will also change

```
b=20;
Cout<<a<<ref; //o/p =20, 20
```

- **Note** * Ref not available in C.

- References r also called alias or implicit pointer.

Q How are refer implemented internally?

A - Internally refer r implemented using constant ptr with automatic indirection.

- Automatic indirection means you donot need to use * operator like ptr. Compiler will apply * operator.

- ∴ They r called implicit ptr.

Q Why are ref used if they work like ptr & we already have ptr?

A. They make it easier to code.

- With ptr, we need to use * operator, but with ref we donot need * operator.

Q where can ref be used?

A. They can be used ~~via~~ ~~obj~~ ~~passing~~. when passing values to a fun.

- If you **pass var** to a fun, then any changes you made inside fun will **not be** permanent.

  eg If you swap values inside fun, it will not be permanent

- But, if you **pass ref** to a fun, then any changes you made inside fun **will be** permanent.

  eg If u swap values, then they will be permanently swapped.

NO ref

```
swap(int x, it y)
{
        int temp;
        temp = x;
          x = y;
          y = temp;

}


swap(a,b);

//a & b will not
    be swapped
```

using references

```
swap(int &refx, int &refy)
{
      int temp
       temp = refx;
    refx = refy;
     refy = temp;
}  //Here refx & refy
   //are references to a&b;


swap(a,b);
//a & b will be
   swapped.
```

Rules for ref

1) Ref must be initialized when creating them

eg
```
int a=10;
int &ref= a;
// ref is initialized to a
```

But, this is wrong:-
```
int a=10;
int &ref;// ref is not initialized
ref = a;
```
X (wrong)

2) Ref can't be null. (ptr can be null)
```
int a=10;
int &ref= NULL; X (wrong)
```

3) Ref can't be reassigned to other var. (ptr can be reassigned)
```
int a= 10, c=20;
int &ref= a;
&ref= c; // ref can't be reassigned to c.
```
X

Ptr vs ref

ptr	Ref
1) ptr stores add of another var	1) ref is another name for same var

2) ptr can be uninitialized when declaring
int a=10;
int *p; //uninitialized
p = &a; // initialize later
	2) Ref must be initialized when declaring
	int a=10;
	int &ref= a;
	//must be initialized here.

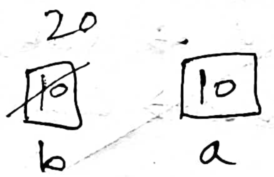
3) ptr can be reassigned. It can ~~pt~~ point to same other var
eg int a=10;
int *p=&a;
int b=20;
p = &b;
//P is reassigned to b
*p // print value of b
	3) Ref can't be reassigned It can't become ref of some other var.
	eg int a=10;
	int &ref= a;
	int c=20;
	ref=c;
	//ref is not reassigned to c. ref will copy the value of c.
	ref is still a ref of a
	# cout<<ref<<a<<endl;
	Both will print same value.

ptr	Ref
4) * operator is needed for ptr.	4) * operator not needed for ptr.
5) ptr can have null	5) Ref can't have null.
6) ptr can be made to point to some other var	6) Ref can't be made to refer some other var.

Examfle of a variable, Ptr & reference

1) int a=10;
int b = a;
Cout << a << b
b = 20; //Change value of b
Cout << a << b;

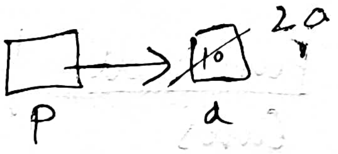
//OP is 10, 20



a & b r Separate var.

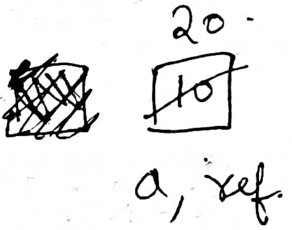
2) int a= 10;
int *p = &a; // assign add of a to p.
~~Cout << a << *p~~
*p = 20; // Change value of a.
Cout << a << *p;
↓ ↓
Print value of a.

//O/P is 20, 20.



3) int a=10;
int &ref= a; //ref is a reference to a.
ref = 20; // Change both a & ref.
cout << a << ref;
↓ ↓
Print same value

//O/P is 20, 20.



a & ref r same var.

Call by value (CBV)
" " Address (CBA)
" " Ref (CBR)

CBV → var is passed.
~~CLASS~~ → copy of var is used

CBA → add of var is passed
→ stored in ptr.

CBR → var is passed
→ stored in ref.

<u>Fun call for all these three cases</u>

CBV: swap(a, b);

var r passed during fun call.

CBA: swap(&~~ref~~a, &~~ref~~b);

add of var r passed. during fun call.

CBR: swap(a, b);

var r passed during fun call.

Fun def for these three cases :-

CBVS $$\text{void swap(}\underbrace{\text{int a, int b}}_{\text{var r used here.}}\text{);}$$

CBA : $$\text{veid swap(}\underbrace{\text{int *a, int *b}}_{\text{ptr r used here.}}\text{);}$$

CBR : $$\text{Veid swap(}\underbrace{\text{int \&refa, int \&refb}}_{\text{ref r used here.}}\text{);}$$

Summary

	CBU	CBA	CBR.
arguments : "Passed" in fun	var(a,b)	add (&a, &b)	var (a,b)
	↓	↓	↓
Parameters : "taken" fun.	var(x,y)	ptr(*x, *y)	~~Ref(&a, &b)~~. Ref(&refa, &refb

NOTE** Take care of what needs to be "passed" & what needs to be "taken" as parameter. See next code.

CBV

```
void swap(int x, int y)
{
    int temp;
    temp = x;
     x = y;
      y = temp;
}

int main()
{
   int a=10, b=20;
   swap(a,b);
   Cout<<a<<b;
   //values r not
        swapped
}
```

Var r taken as param

Var are passed

O/P = 10,20

CBA

```
void swap(int *ptra,
          int * ptr b)
{
     int temp;
    temp = *ptra;
    *ptra = *ptrb;
    *ptrb = temp;
}

int main()
{
   int a=10, b=20;
   swap(&a,&b);
   Cout<<a<<b;
   //Values r swapped
}
```

pointers r taken as param

Addresses r Passed

There r not references

O/P = 20,10.

CBR.

```
void swap(int&ref a,
          int&ref b)
{
   int temp;
   temp = ref a;
   ref a = ref b;
   ref b = temp;
}

int main()
{
   int a=10, b=20;
   swap(a,b);
   cout<<a<<b;
 //values r swapped
}
```

References r taken as param

these r references, not address

Var r Passed

O/P = 20,10

CBV	CBA	CBR.
• Values r not swapped	• Values r swapped	• Values r swapped.
• Parameters :- Var r taken as param.	• Pointer r taken as param	• Ref r taken as param.
• Arguments (What r passed to fun) Var r passed to fun	• Addresses r passed to fun	• Var r passed to fun.
• Fun def No * operator used inside fun definition	• * operator is used by pointer inside fun def.	• No * operator used by references inside fun def.