1. Tokens:
2. Keywords

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bool | asm | catch | char16\_t | char32\_t | const\_cast | friend | template | this | try |
| virtual | volatile | throw | enum |  |  |  |  |  |  |

**CASE SENSITIVE!!**

1. Identifiers

Allowed: Non-keywords, Non- Literals MyName; BREAK. . . .

But:

* Not keywords break
* No special character except “\_” Data-D; My.File
* No beginning with digits 29Cat
* No space My Name
* No Single/Double quotes ‘ABC’; ”ABC”

1. Literals (or Constants)
2. Integers Constants
3. Decimal Any number not beginning with zero
4. Octal Octal system; PRECEDED BY 0
5. Hexadecimal Hexadecimal system; PRECEDED BY 0X or 0x
6. Character Constants

Single Character enclosed in ‘ ‘

1. Floating Constants(real constants)

Numbers with fractional parts 2.0; 3.008; -9.6; 0.00045

Or exponents 3.4E45 or 3.12e46

Invalid if:

* No decimal 7; 6e3
* No digit after decimal 7. ; 7.E3
* Symbols other than ‘. ‘ 3/5
* More than one decimal points 5.654.54
* Comma present 34,665.76
* No digits after e/E 3.4e

1. String literals

Sequence of characters enclosed in “ “

1. Punctuators
2. Operators
3. Data Handling:
4. Data types

unsigned Range same but from 0: int is -32768 to 32767 unsigned int is 0 to 65535  
signed All same

**Long float same as Double**

Size:

short 2

unsigned short 2

signed short 2`

int 2

unsigned int 2

signed int 2

long 4

unsigned long 4

signed long 4

char 1

unsigned char 1

signed char 1

float 4

double 8

long double 10

Derived :

* + Arrays
  + Functions
  + Pointers
  + Reference
  + Constant

User-Defined Derived:

* + Class
  + Structure
  + Union: Two or more variables share location; declaration similar to structure
  + Enumeration: Alternative to declare integer constants---enum{START, PAUSE, GO}
    - **Values automatically declared from 0**
    - Enumeration can be named---enum status{START, PAUSE, GO}

**const int should be initialised while declaring**

1. Formatting the Output

setw()

has to be used for every line

cout<<setw(5)<<2; => \_ \_ \_ \_ 2

setfill()

to fill spaces in setw()

cout<<setfill(‘0’);

cout<<setw(5)<<2; => 0 0 0 0 2

setprecision()

to set number of digits displayed

for entire number:

cout<<setprecision(5)<<12.5424512; => 12.542

for decimal places only:

cout.setf(ios::fixed);

cout<<setprecision(5)<<12.5424512; => 12.54245

other ios flags:

left left justify

right right justify

showpoint show decimal point and trailing zeroes

showpos display plus sign

uppercase make e as E

scientific display in E notation

fixed normal notation-no trails, no scientific notation

cout.setf(ios::fixed); cout.setf(ios::showpoint);

cout<<setprecision(5)<<5.8;

=> 5.80000

expression1?:expression2:expression3; equivalent to. . . if(expression1==1)

{

expression2;

}

else

expression3;

comma can also be used for sequencing. . .

b=(a=3, a+1); equivalent to a=3; b=a+1;

order of preference

++,--

Sizeof,!,+,-

\*,/,%

<,>,<=,>=

==,!=

&&

||

?:

= and other assignments

,

a=2,b=5 3 \* a ++ - 3 / 3 + b = 13

Type casting:

* + - 1. Implicit

Automatic by system when expression has more than one data types

* + - 1. Explicit

Done by user

Syntax: (datatype) expression will print the result in given type only

x=x(operation)y; can be written as x(operation)=y(constant);

x=x+y; is x+=y;

Generating random numbers

rand(): any number from 0

(rand()%U)+L: any number from L to U

random(X): any number from 0 to X

random(X)+ L: any number from L to L+X-1 random(61)+40----> from 40 to 100

Structure: Data only

Class: Data and functions

|  |  |
| --- | --- |
| iostream.h | cin, cout, endl |
| conio.h | getch, clrscr |
| iomanip.h | setw, setprecision |
| string.h |  |
| ctype.h |  |
| math.h |  |
| stdio.h | gets, puts, scanf, printf, remove, rename |
| stdlib.h (miscellaneous) | random, cin.getline, cin.get, abs |

endl: flushes output buffer

]Required for file handling

\n: does not

Procedural Programming: Emphasis on function; separates functions and data manipulated by them

Object Based Programming: Emphasis on object; no inheritance

Object Oriented Programming: Emphasis on object; inheritance allowed

Properties of OOP:

…Data abstraction: (class/structure)

Showing essential features and hiding rest

…Data encapsulation: (objects of classes)

Wrapping data and functions working on the data into a single unit called class

* Abstraction focuses on observable behaviour of objects and encapsulation focuses on implementation that gives rise to this behaviour

…Modularity:

Partitioning a program into different components to decrease complexity and to create well-defined, documented boundaries

…Inheritance:

The ability of a class to derive date and properties from another class

Reusable

Transitive: [A->B & B->C => A->C]

…Polymorphism:

Same message, different reactions by different classes

By virtual functions, function overloading

**Advantage of OOP:**

Reusability of data

Ease of comprehension

Ease of designing and maintenance

Ease of redesigning and extension

**Disadvantage of OOP:**

Overly generalised

Tricky to design

At times relations are artificial

Static/Early binding: Arguments of methods/function are resolved in the compilation stage

Function Overloading: A function having different definitions differentiable by the number and types of argument in each definition

Signature of a function: The number, type and order of arguments of a function

int ABC(int a); //original

int ABC(int a); //seen as repeat

char ABC(int b); //seen as error

char ABC(char b); //allowed since diff type of argument

int ABC(char b); //allowed since diff argument though return type same

**Typedef doesn’t change type**

Same array name and diff dimensions are different

int func(int A[2]);

int func(int A[2][3]); //different

Classes:

A data type to bind data and its associated functions

All private members by default (in structures, all public by default)

Functions: (what they do)

Accessor functions…to return values of data members

Mutator functions…to change data members

Manager functions…specific functions like constructors, destructors

Functions: (method of compiling)

Inline…

Replaces function call with function code; compiler doesn’t have to jump from call statement to function as function already available, hence fast

inline <function name>(arguments) { }

Member functions are inline by default if defined inside the class...hence those should be small

Constant…

cannot alter values of date members

<function name>() const { }

Nested…

Called by another member function

Scope resolution:

int x,y;

class A

{

int x; //local to class

void f()

{ x=3; //assigned to local x not global x

y=4; //assigned to global y

::x=5; //assigned to global x

}

};

Static Data members: Data members available to all objects; associated with class not any object

Declared inside

**Defined outside**

Static Member Function: Can access only static data members and invoked by class name not object

class ABC

{ static int x;

public:

static func() {cout<<x;}

};

int ABC::x; //define outside

int main()

{

ABC::func();

}

struct student

{

int roll;

float marks;

}

int main()

{

student s1={0,0.0} ; //allowed

}

class student

{

int roll;

float marks;

}

int main()

{

student s1={0,0.0}; //illegal

}

**Access specifiers:** Private > Protected > Public

**Constructors:**

Default: No parameters

Parameterised/regular: Takes parameters

Copy: To copy values object to object

**Parameterised constructor with default arguments is equivalent to default constructor**

class ABC

{

int x;

public:

ABC(int i=20)

{

x=i;

}

}

int main()

{

ABC obj(34);

ABC obj2; // allowed as default argument present

//same as default constructor

class ABC

{

int x;

public:

ABC(int i)

{

x=i;

}

}

int main()

{

ABC obj(34);

ABC obj2; //not allowed as default constructor

//not present

Try to avoid:

class ABC

{

public:

ABC();

ABC(i=2);

}

…since both same if statement is: ABC obj; //ambiguity hence error

To create array of objects of a class type…default constructor is mandatory

Constructor initialised when object created  
Destructors initialised at the end

Calling a constructor:

Implicitly: *ABC obj(2,3.4,’c’);*

Explicitly: *ABC obj=ABC(2,3.4,’c’);* //name of constructor used

Explicit calling used specially to create Temporary Instance/Object:

They exist only for that statement. Deleted after that

No object created for that

*class ABC*

*{*

*int i,j;*

*public:*

*ABC(int a,int b)*

*{i=a;j=b}*

*void print(){cout<<i<<j;}*

*}*

*int main()*

*{*

*ABC obj(1,2);*

*obj.print();* //gives…12

*ABC(2,3).print();* //gives….23

*}*

Classes are derived data types…have constructors. Fundamental data type also has constructors.

int a,b,c; //default constructor used(garbage value)

int i(2),j(3),k(4); //initialised by given values

Copy constructor:

ABC obj1; //default constructor used

ABC obj2=obj1; //copy constructor used

If no user defined copy constructor given, automatically added

To give user defined copy constructor…

class ABC

{

int i,j;

public:

ABC(int a,int b)

{i=a;j=b;}

ABC(ABC&obj)

{i=obj.i;j=obj.j;}

};

int main()

{

ABC obj1(1,2);

ABC obj2(obj1); //copy constructor called

ABC obj3=obj1; //same

func(ABC obj4); //copy constructor called since called by value

//also if function returns an object

}

Why argument in copy constructor not passed by value?

If passed by value, another copy constructor required to make a copy, hence calls itself again and again till out of memory. Hence, error.

**Base class constructor invoked first; base destructor invoke last**

**Invocation of constructor is in the order of declaration; invocation of destructor in opposite order**

Dynamic initialisation: (initialisation during runtime)

int main()

{

int x,y;

cin>>x>>y;

ABC obj(x,y);

….

}

**Constructors can also be overloaded like functions**

**Default Constructor not same as Overloaded constructor**

Destructor:

~ABC() {}

No arguments taken and no return type specified (not even *void*)

Inheritance:

Single

Multiple

Hierarchy

Multilevel

Hybrid

|  |  |  |  |
| --- | --- | --- | --- |
| Base\Inheritance | Publicly | Protected | Privately |
| Public Member | Public | Protected | Private |
| Private Member | X | X | X |
| Protected Member | Protected | Protected | Private |

Private>Protected>Public

**Derive…when:**

Publicly when **all attributed needed** with extra ones

Privately when inherited attributes are **not to be inherited further**

Protected when inherited **attributes to be hidden but inheritable further**

**IMP:**

*class base{*

*public: int x;*

*};*

*class Derived1:public Base*

*{ public: int y;*

*private:*

*using Base::x;* //not allowed; public into private, i.e., selective denial not allowed

*};*

*class base{*

*int x;*

*};*

*class Derived1:private Base*

*{*

*public: int y;*

*using Base::x;* //allowed, private to public, i.e. selective access allowed

*};*

**Same for member functions**

**To inherit private members, change to public or private:**

To public…..no use as data hiding not done

To protected….better as data hiding retained

Abstract class: Not object of class exists, used only for inheritance

**IMP:**

**Overshadowing/overriding functions of base class:**

*class A{*

*int a,b;*

*public:*

*void func()*

*{*

*cout<<”Base”;*

*}*

*};*

*class B : public A{*

*public:*

*void func(double x);* //func redefined

*};*

*int main()*

*{*

*A obja;*

*Obja.func();* //allowed

*B objb;*

*objb.func();* //not allowed since overshadowed; error

*objb.func(3.4111);* //allowed

*}*

**Overloading overshadowed functions(unveiling):**

*class A{*

*int a,b;*

*public:*

*void func()*

*{*

*cout<<”Base”;*

*}*

*};*

*class B : public A{*

*public:*

*using A::func;* //() not required

*void func(double x);* //func redefined

*};*

*int main()*

*{*

*A obja;*

*Obja.func();* //allowed

*B objb;*

*objb.func();* //allowed now since overloaded

*objb.func(3.4111);* //allowed

*}*

**Remember: If base functions are over-loaded, signatures should not match, if matched, derived class functions preferred**

**If in multiple inheritance base class has parameterised/copy constructor, better to add separate constructor in derived class**

**Dealing with dreaded diamond:**

D takes 2 copies of A, leading to ambiguity. Therefore scope used to specify which version to use.

*class A{*

*public: inta;*

*};*

*class B : public A{*

*int b;*

*};*

*class C : public A{*

*int c;*

*};*

*class D : public B, public C{*

*int d;*

*};*

*int main()*

*{*

*D obj;*

*obj.a=5;* //ambiguous

*obj.B::a==5;* //specifies to use B’s version of A

*}*

**But to prevent multiple copies from being inherited, use virtual class:**

*class A{*

*public: inta; };*

*class B :* ***virtual*** *public A{*

*int b;*

*};*

*class C :* ***virtual*** *public A{*

*int c;*

*};*

*class D : public B, public C{* //only one copy of A inherited

*int d;*

*};*

*int main()*

*{*

*D obj;*

*obj.a=5;* //not ambiguous anymore

*obj.B::a==5;* //still correct, can be used also

*}*

**Ways to have properties of one class in another:**

1. **Inheritance**
2. **Nested class**
3. **Containership/Containment/Aggregation:** Objects of other classes present as data members

Relationship between classes:

1. IS-A: inherits
2. HAS-A: has object
3. HOLDS-A: same as HAS-A but without ownership

Pointers:

Declaration syntax: <data type> \* <name>;

Pointer with data type, d stores address of only that data type

int i=25; //initializing variable

int \* ptr; //declaring pointer

ptr= &i; //initializing pointer with location of i

cout<<ptr; //prints locations of i

cout<<\*ptr; //prints value of i not address

**Never leave pointers uninitialized**

**Use int \* ptr=NULL**

**Better to use nullptr: <datatype> \* ptr=nullptr;**

**NULL makes value zero hence becomes int**

**Not so if nullptr used (helpful when overloading)**

*fun(char \*);*

*func2(int);*

*:*

*char\*ptr=NULL;*

*func(ptr);* //invokes func(int) since ptr becomes int (=0)

*:*

*char\*ptr=nullptr;*

*func(ptr);* //invokes func(char\*)

ptr++ or ptr-- increases/decreases by size of data type and not 1

Base address: The address of the first byte of the memory location

Pointer points to base address by default

char

int

1020

1001

1021

1020

1001

a

Static memory allocation: When memory size to be allotted is predefined

Dynamic memory allocation: When the memory is allotted during runtime

*int x;*  //static

*new obj;* //dynamic

delete is another dynamic memory allocation operator

Using new operator:

*<pointer name>=new <datatype>;*

int\*ptr;

ptr=new int;

pointer data type must be same!!

Assigning values through pointers:

*int\*ptr=new int;*

*\*ptr=23;*

For 1-D arrays:

int\*ptr=new int[20]; //always give size

int\*ptr2=new int[5]{1,5,2,9,4};

**Lifetime of object created by new is not restricted to any scope. It is removed explicitly by delete operator**

**Always check if memory is available or not:**

*int\*ptr=new int;*

*if(!ptr)*

*cout<<”No memory!!”;*

For 2-D arrays:

int\*ptr;

int r,c;

ptr=new int[r\*c]; //not int[r][c]

//to give values…

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

{

cout<<”Enter element of row ”<<i+1<<” “;

for(int j=0;j<c;j++)

{

cin>>val[i\*c+j]; //not v[i][j] since it is pointer

}

}

Memory leaks due to improper use of new and delete:

Forgetting to delete

If code can bypass delete operator

Assigning new statement to pointer already pointing to an object

**Array names are seen as pointers pointing to first element**

**For…**

*int age[5];* //age points to age[0]

Array of pointers:

*int\*ptr[10];*

*ptr[3]=&i;* //4th pointer in array pointing to i

*cout<<\*prt[3]; //to print amt*