Data types:-

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

1. char, unsigned char, signed char → 1 Byte
2. Int, unsigned Int, signed Int → 4 Byte
3. Short, unsigned short, signed short → 2 bytes
4. long int, us long int, s long int, long long int = 8 bytes
5. Float → 4 bytes
6. Double → 8 bytes
7. long double → 12 bytes
8. wchar_t = 2 or 4 bytes.

Typedef declarations:-

→ you can create a new name for an existing type using typedef.

```
typedef type newname;
typedef int feet; // feet is another name of int
    feet distance;
```

Enumerated types:-

Enumeration type declares an optional type name and set of zeros & more identifiers that can be used as value of ~~int~~ type.

```
enum enum-name {list of names} var-list;
enum color {red, green, blue} c;
        c = blue;
enum color {red, green = 5, blue};
```

By default red = 0 green = 1 blue = 2

by green = 5 blue = 6 & red = 0.

Two kinds of expressions:-

lvalue - Expressions that refer to a memory location is called lvalue expression.

rvalue:- The term rvalue refers to a data value that is stored at some address in memory. Rvalue is an expression that can't have a value assigned to it which means an rvalue may appear on right but not at left hand side of assignment.

literals:-

Constants refer to fixed values that the program may not alter and they are called literals.

* #defind identifier value
* const type variable = value;

Type qualifiers in cpp:-

1. const - objects of type const cannot be changed by your program during execution.
2. Volatile:- The modifier volatile tells the compiler that a variable value may be changed in ways not explicitly specified by the program.

(3) restrict :- A pointer qualified by restrict is initially that the only means by which the object it points to can be accessed only (gg adds a new type qualifier called restrict.

Storage Class :-

It defines the scope and life-time of variables and/or functions within a cpp program.

(1) Auto storage Class :-

- ~~Aut~~ It is the default storage class for all local variables.

```
{ int mount
  auto int mount;
}
```

In above example there are two variables with same storage class but auto can only use within function i.e. local variable.

(2) ~~Ifre~~ Register Storage Class :-

- It is used to define local variables that should be stored in a register instead of RAM.
- This means variables max. size = the register size (usually one word) and can't have the

unary '&' operator applied to it.

- It doesnot have a memory location.

{ register int miles; }

- It should only be used for variables that require quick access such as counters.

- defining register doesnot mean that the variable will be stored in a register.
  It only means that it might be stored in a register depending on h/w & implementation restrictions.

The static storage class:-

- The static storage class instruct the compiler to keep a local variable in existance during the life time of the program instead of creating & destroying it each time it comes into and goes out of scope.

- Making local variable static allows them to maintain their values between function call.

- The static modifier may also be applied to global variable.
  When this is done it get restricted

- Static is used on a class data member it causes only one copy of that member to be shared by all objects of its class.

The extern storage class :—

- It is used to give a reference ~~variable~~ ~~of~~ of global variable that is visible to all the program files.

- When you use 'extern' the variable ~~or function~~ cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

main.cpp

```
#include<iostream>
  int count;
extern void write_extern();
main() {
  count = 5;
  write_extern();
}
```

support.cpp.

```
# include
 extern int count;

Void write_extern (void){
std:: cout<< count<<
    std::endl;
 }
```

O/p - 5.

Mutable storage class:—

The mutable specifier applies only to class objects, ~~which are discussed later in~~

It allows a member of an object to override const member function.

Mutable member can be modified by a const member function.

OPERATORS:—

(I) Arithmetic operators:—

+, -, *, /, %, ++, --;

(II) Relational:—

==, !=, >, <, >=, <=,

(III) logical:—

&&, ||, !

(IV) Bitwise:—

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

128 64 32 16 8 4 2 1
1 1 1 1 0 0
1 1 1 1 0 0 0 0

~ - compliment.

<< - left sift -

A = 60 ~~60~~ 60<<2 = 240.

>> - right shift.

A = 60 60>>2 = 15

Assignment operator:-

=, +=, -=, *=, /=, %=, <<=, >>=
&=, ^=, |=

Misc.

⇒ Size of()
⇒ Condition ? X : Y
⇒ ,
⇒ .(dot) and ->(arrow)
Member operators are used to reference individual members of classes, structures and unions.

⇒ & - address
⇒ * - pointer.

Only - ~~&~~ ? :, =, +=, -=, *=, /=, %=, >>=, <<= &=, ^=, |=

Conditional & Assignment operater precedence is Right to left
~~otherwise~~ any other operator has left to right

Function :-

⇒ strcat() - to concatenate two strings.

⇒ memcpy() - to copy one memory location to another location.

* Passing arrays to a function :-

(i) void myfunc(int *param) {
}

(ii) void myfunc(int param[]) {
}

*. Strings :-

char greeting[6] = {'H', 'E', 'L', 'L', 'O', '\0'};
char greeting[] = "Hello";

(1) strcpy(s1, s2);
(2) strcat(s1, s2);
(3) strlen(s1);
(4) strcmp(s1, s2);
(5) strchr(s1, ch); - returns a pointer to the first occurrance of char ch in string s1.
(6) strstr(s1, s2); returns a pointer to the first occurance of string s2 in s1.

XYZ

Pointers:-

```
int main() {
    int var1;
    int var2[10];
    cout <<&var1 << var2; //o/p- Adress of both.
```

```
int main() {
    int var=20;
    int *ip;
    ip = &var;
    cout << var; // o/p = 20
    coup<< ip ; // o/p = Address
    cout<< *p ; // op = 20.
```

```
int main() {
    int var[3] = {10, 100, 200};
    int *ptr;
    ptr = var
    for (i=0 ; i<3 ; i++) {
        cout<<ptr;      // Address; Ad, Ad
        Cout << *ptr    // Value; 10, 100, 200
        ptr++;
    }
```

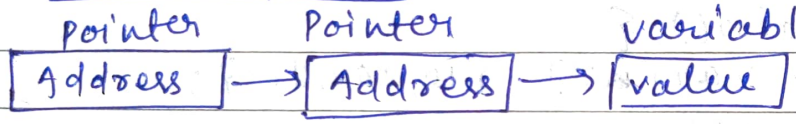
```
int main() {
    int var[3] = {10, 100, 200};
    int *ptr;
    ptr = var;
    int i = 0;
    while (ptr<= &var[3-1]) {
        cout<<ptr; // Address
        cout<<*ptr; // value
```

* int var[3] = {10, 100, 200};
for (int i = 0; i < 3; i++) {
*var = i; // correct.
var++; // This is not correct
}

~~*(var+2)~~

* int main() {
int arr[3] = {1, 2, 3};
cout << *(arr); = 1
cout << *(arr+1); = 2
cout << *(arr+2); = 3

* Pointer to pointer :-



int ** var;

* int main(){
int var;
int *ptr;
int **pptr;
var = 3000;
ptr = &var
pptr = &ptr
cout << var // o/p - 3000
cout << *ptr // o/p - 3000
cout << **ptr // op - 3000.

* Return pointer through function:-

int *getRandom() {
static int r[10];
srand((unsigned) time(NULL));

```
for(int i=0; i<10; ++i){
        r[i] = rand();
        cout<<r[i]<<endl;
    }
return r; } //
int main(){
  int *p;
    p= getrandom();
  for (int i= 0; i<10; i++){
       cout<<(p+i)<<endl
  }
```

Smart pointer;

⇒ Smart pointer is a wrapper class over a pointer with an operator like * and -> overload.

⇒ It can deallocate and free destroyed object memory.

```
* Class smartptr{
        int *ptr;
    Public:
        explicit smartptr(int * p =NULL){ptr= p;}
        ~smartPtr(){delete(ptr);}
        int & operator*(){return *ptr;};

int main()
{smartptr ptr(new int());
      *ptr = 20;
      cout<<*ptr;
     return 0;
}
```

O/P = 20.

Types of smart pointer:-

① unique_ptr:-

- It stores one pointer only.
- We can assign a different object by removing the current object from the pointer.

② shared_ptr:-

- By using shared_ptr more than one pointer can point to this one object at a time and it will maintain reference counter using use_count() method.

③ weak_ptr:-

- Its much more similar to shared_pointer except it will not maintain a Reference counter.

* Structure:-

- User defined data types
- Used to combine different data types.

```
Struct abc {
    int a1;
char int b1;
  float c1;
int main() {
   Struct abc ab;
    ab.a1 = 85
    ab.b1 = 'A'
```

cout<< abc.a1 << abc.b1 << abc.c1;

o/p = 85 A 85.8

Structure doing typedef :—

```
typedef struct abcd {
    int A1;
Char ~~int~~ B1;
    float C1;
} Ank;

int main(){
    Ank ankit;
    ankit.A1 = 85;
    ankit.B1 = 'A';
    ankit.C1 = 85.85;
cout << ankit.A1 << " " << ankit.B1 << " " << ankit.C1;
```

o/p = 85 A 85.85

* Unions :- It is a type of structure that can be used where the amount of memory used is a key factor.

→ This is most useful when the type of data being passed through functions is unknown, using a Union which contains all possible data types can remedy this problem.

```
union ank {
    int A1
    char A2
    float A3
};
int main(){
    union ank a1;

    a1.A1 = 34;
    Cout << a1.A1;  //o/p = 34

    a1.A2 = 34
    Cout << a1.A2;  //o/p = "

    a1.A3 = 34.34
    Cout << a1.A3;  //op = 34.34;
```

References:-

- Another name for already existing variable.

References vs pointer:-

- You can't have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed

to refer to another object.
Pointer can be pointed to another object at anytime.

- A reference must be initialized when it is created. Pointers can be initialized at any time.

* Creating reference.

```
int a = 27;
int &r = a;
```

*
```
int main(){
    int i;
    double d;

    int &r = i;
    double &s = d;

    i = 5;
    cout << i << " " << r; // op = 5 5
    d = 50.7;
    cout << d << " " << s; //op = 50.7 50.7
```

passing parameter by references in cpp.

```
void swap (int &x, int& y);
int main() {
int a= 100;
int b= 200;
 cout<< a <<" " << b; // 100 200

swap(a,b);

Cout << a <<"  " << b; //200 100

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

    return;
}
```

Returning values by Reference in Cpp

- when a function returns a reference, it returns an implicit pointer to its return value.

- This way, a function can be used on the left side of an assignment statement.

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

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};
double & setValues(int i) {
    return vals[i]; // return reference to ith element
}
int main() { for(int i=0; i<5; i++) {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
}
  setValues(1) = 20.23 // change 2nd element.
  setValues(3) = vals[i] << endl;

  cout << " Value after change " << endl;
  for(int i=0; i<5; i++) {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
  }
  return 0;
}
```

Values before change
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
Value after change
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50.

65-90 , 97-122

- int& func() {
  int q;
  //! return q; // compiletime error
  static int x;
  return x; // safe, x lives outside
  // this scope.
  }

- when returning a reference, be careful that the object being reffered to doesn't go out of scope.

- It is not legal to return a reference to local var. But you can always return a reference on static variable.

CASTING Operators :-

- one data type to be converted into another.

- It is a unary operator & same predilence as unary operators.

(type) expression.

① const_cast<type>(expr) -

The const_cast operator is used to explicitly override const and volitile in a cast.

The target type must be the same as source type except for the alteration of its const or volatile attributes.

(Char ch = char(rem) + 0)

This type of casting manipulates the const attributes of passed object, either to be set or removed.

(2) dynamic_cast<type>(expr):-

It perform a runtime cast that verifies the validity of the cast.

If the cast cannot be made, the cast fails and the expression evaluates to null.

It performs casts on polymorphic typyes & can cast A* to B* only if object being pointed to actually is a B object.

(3) reinterpret_cast<type>(expr):-

It changes a pointer to another type of pointer.

It allows casting from pointer to an integer type & vice versa.

(4) Static_cast<type>(expr):-

It performs a nonpolymorphic cast.

ex- it can be used to cast a base class pointer into a derived class pointer.

All 1-4 casting operators are used while working with classes & objects.

```
main(){
    double a = 21.09399;  char d = 'A';
    float b = 10.20;
    int c;
  c = (int) a;
  cout << c << endl;  // 21
  c = (int) b
  cout << c << endl;  // 10
    c = (int) d
  cout << d << endl;  // 65.
```

C++ preprocessor :—

- Preprocessors are the directives, which gives instructions to the compiler to preprocess the information before actual compilation starts.
- Start with #.
- The macro is used to include a header file into the source file.
- Preprocessors supported by C++ like #include, #define, #if, #else #Line etc.

#define preprocessor :—

#define preprocessor create symbolic constants,
The symbolic constant is called a macro

#define macro-name replacement-text

```
using namespace std;
#define PI 3.14159
int main() {
    cout << PI;      // 3.14159.
return 0;
}
```

* Function like macros:—

```
#define MIN(a,b) ((a)<(b)) ? a : b)
int main() {
    int i, j;
    i = 100;
    j = 30;
  cout << MIN(i,j) ; // 30.
return 0;
}
```

* Conditional Compilation:—

- used to compile selective portions of your programs source code

```
#ifndef NULL
  #define NULL 0
#endif
```

- We can compile a program for debugging purpose.
We can turn on or off the debugging using a single macros:—

* ifdef DEBUG
  cerr << " x << endl;
  #endif.

— This causes the cerr statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG.

— we can use #if 0 statement to comment out a portion of a program.

* #if 0
  code prevented for compiling
  # endif.

* #define DEBUG

```
#define MIN(a,b),((a)<(b)) ? a : b)
int main(){
    int i, j;
      i = 100;
      j = 30
#ifdef DEBUG
   cerr << "Trace: Inside main func" << endl;
#endif

#if 0
   /* This is commented part */
    cout << MKSTR(HELLO C++) << endl;
# endif.
```

```
cout << MIN(i,j) << endl;

#ifdef DEBUG
   cerr << "Trace: Coming out of main func"
                                    << endl;
#endif
   return 0;
}
```

O/p- 30
Trace: Inside main func
Trace: Coming out of main func.

The # and ## operator :-

- The # operator causes a replacement text token to be converted to a string surrounded by quotes.

```
#define MKSTR(x) #x
int main(){
    cout << MKSTR(HELLO C++) << endl;
  return 0;
}
```

O/p - HELLO C++

Here MKSTR(HELLO C++) turned into "HELLO C++";

- The ## operator is used to concatenate two tokens

```
#define CONCAT(x, y) x##y
CONCAT(HELLO, C++) x##y
```

"HELLOC++"

```
#define concat(a,b) a##b
int main(){
    int xy=100
    cout << concat(x,y);
    return 0;
}
```

o/p = 100.

concat(x,y) = (xy)

* Predefined C++ Macros :-

1. __LINE__ → It contains the current line number of the program when it is being compiled.

2. __FILE__ → This contains the current file name of the program when it is being compiled.

3. __DATE__ :- Contain month/date/year that is date of translation of source file into object code.

4. __TIME__ hour:min:sec at which the program was compiled.