Input Buffer

- when `cin>>someInteger;` is followed by `cin>>str;` on the next line and the enter key which is pressed after typing the integer value gets saved as an input buffer and later is accepted by `cin>>str;`
  - to avoid this use `cin.ignore();` between `cin>>someInteger;` and `cin>>str;`

decltype

- `decltype(x) y;` will declare a new variable y which has the data type of variable x which already exists

enum

- used to alias a constant integer with a commonly used name to improve readability
  - `enum day {mon,tue,wed,thr,fri,sat,sun};` is same as `const int mon = 0, tue = 1, wed = 2, thr = 3, fri = 4, sat = 5, sun = 6;`
- day is a user defined data type which can only store values mon, tue, wed, thr, fri, sat, sun and the variable automatically gets an int value aliased with the name

```
#include <iostream>
using namespace std;
int main()
{
    enum day
    {
```

```
        mon,
        tue,
        wed,
        thr,
        fri,
        sat,
        sun
    } d = mon;
    // or day d = mon;

    cout << d << endl; // 0
    // d = jan; error

    cout << tue; // 1

    return 0;
}
```

- `enum day {mon=4,tue,wed,thr,fri,sat,sun};` assigns mon =4 tue=5 wed=6 thr=7 fri=8 sat=9 sun=10
- `enum day {mon=4,tue,wed,thr,fri=13,sat,sun};` assigns mon=4 tue=5 wed=6 thr=7 fri=13 sat=14 sun=15
- `enum day {mon=4,tue,wed,thr,fri=6,sat,sun};` assigns mon=4 tue=5 wed=6 thr=7 fri=6 sat=7 sun=8

Type Definitions

- makes variables more readable

- so lets say a school has `int m1,m2,m3,h1,h2,h3;` where `mX` are your marks in subject X and `hX` is the highest marks in that subject
- to make it more readable we use `typedef` to alias the data type of `mX` from int to marks and `hX` from int to highest to add more context

```
typedef int marks, highest;
marks m1, m2, m3;
highest h1, h2, h3;
```

For Each Loop

```
int A[] = {1, 2, 3, 4, 5};
for (int x : A)
        cout << x;
```

- `for (auto x : A)` where the compiler infers the data type of x on its own

```
int A[] = {1, 2, 3, 4, 5};
for (auto x : A)
    cout << x;
```

- modifying x in the loop doesn't change the corresponding element in array because x is a copy
  - to avoid this you can use `int &x : A` instead of `int x : A`
    - `auto &x : A` is also allowed

Binary Search

```cpp
#include <iostream>
using namespace std;
int main()
{
    const int A[] = {4, 8, 24, 42, 101, 404, 1234};
    int key, l = 0, h = (sizeof(A) / sizeof(int)) - 1;

    cout << "Enter Key: ";
    cin >> key;

    while (l <= h)
    {
        int mid = (l + h) / 2;

        if (key == A[mid])
        {
            cout << "Found at " << mid;
            return 0;
        }
        else if (key < A[mid])
            h = mid - 1;
        else
            l = mid + 1;
    }

    cout << "Not Found";
    return -1;
}
```

Finding Min and Max of an Array

```cpp
#include <iostream>
#include <climits>
using namespace std;
int main()
{
    const int A[] = {4, 8, 24, 42, 101, 404, 1234};
    int min = INT_MAX, max = INT_MIN;

    for (auto x : A)
    {
        min = x < min ? x : min;
        max = x > max ? x : max;
    }

    cout << "Min = " << min << endl;
    cout << "Max = " << max;

    return 0;
}
```

Note on Arrays

- when an array of length n is created and when m elements are hard coded then remaining n-m elements are automatically assigned with 0(provided the length of the array is not an user input)

- `A[i]` is always equal to `i[A]` where `A` is an array and `i` is an integer counter variable of a for loop because `*(A+i)` is same as `*(i+A)`

- when `int *p = A` and `A` is an array then we can used `p[i]` instead of `A[i]` without having to dereference

- all the elements of a 2-D array are contiguous in the memory just like a 1-D array

- need to use reference of x when A is 2-D array is iterated over using a for each loop, but you still have to use nested for each loop

```cpp
#include <iostream>
using namespace std;
int main()
{
    const int A[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

    for (auto &i : A)
    {
        for (auto j : i)
            cout << j << " ";
        cout << endl;
    }

    return 0;
}
```

  - `const int A[][] = {{1, 2, 3, 4}, {5, 6, 7, 8}};` is invalid
  - prefer using auto in a nested for each loop iterating over a 2-D array

- passing an array to a function by value decays it into a pointer to the first element of the array but if the array is a member of a struct and the struct is passed by value then the array doesn't decay

Pointers

- size of a pointer is independent of data type of the variable it is pointing to and always takes 8 bytes

- *Memory Layout*
  - `| HEAP |`
  - `| STACK|` ← declarations like `int i = 0;` are stored in STACK and is automatically deleted when goes out of scope
  - `| CODE |` ← read-only part of the memory where the code is loaded after launching the program and global variables are stored here
    - the CODE section can access STACK and itself but not the HEAP
    - to access the HEAP from the CODE section you need to create a pointer to a memory address in HEAP and the pointer is created in the STACK from the CODE section
      - thus HEAP can only be accessed using pointers
      - accessing Files and hardware devices is also done using pointers
    - accessing HEAP using `new`
      - example: `int *p = new int[5];`
      - if not freed at the end, we get a Memory Leak so use `delete[] p;` and then `p = nullptr;`
        - don't do `p = nullptr;` first as you won't be able to free HEAP later

- a 2-D array in HEAP is an array of pointers in stack such that every pointer of the array points to an array in the HEAP
- once an array is created in the STACK you cannot change its size but it is possible if it is in the HEAP

```cpp
#include <iostream>
using namespace std;
int main()
{
    int *p = new int[5];

    for (int i = 0; i < 5; i++)
    {
        p[i] = i + 1;
        cout << p[i] << " ";
    }

    delete[] p;       // to avoid leaking of {1, 2, 3, 4, 5}
    p = new int[10]; // new memory gets allocated pointing to {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

    delete[] p;
    p = nullptr;

    return 0;
}
```

- Pointer Arithmetic
    - dereferencing is not needed

- subtracting 2 pointers pointing to different indices of an integer array gives how many indices far are the 2 pointers without dividing by 4 (size of an int)

- doing `p+=2` moves the pointer by 8 bytes instead of 2 bytes if it is pointing to integer array

- this is

```
#include <iostream>
using namespace std;
int main()
{
    int A[] = {1, 2, 3, 4, 5};
    int *p = A; // Same as int *p = &A[0]

    for (int i = 0; i < sizeof(A) / sizeof(int); i++, p++)
        cout << *p;

    return 0;
}
```

- same as this

```
#include <iostream>
using namespace std;
int main()
{
    int A[] = {1, 2, 3, 4, 5};
```

```
    for (int i = 0; i < sizeof(A) / sizeof(int); i++)
        cout << *(A + i);

    return 0;
}
```

- Runtime gotchas w/ pointers
  - memory leak when HEAP allocations are not freed
  - uninitialized pointer
    - `int *p = 25;` where 25 is not stored in a variable
      - you can use `int *p = &n;` or `int *p = new int;`
  - dangling pointer
    - If a pointer is having an address of a memory location which is already deallocated
    - if a pointer is passed to a function and freed at the end of that function and if the pointer is accessed again after the control returns to the calling function then we have a runtime error but not a memory leak
- References
  - `int x = 10;` allocates a box in the memory named x which stores value 10
  - `int &y = x;` creates an alias of x which is y
    - So x and y will always have the same value and changes to one can be reflected on the other variable as well
  - doesn't consume memory
  - `&x` is always same as `&y`
  - declaration without initialization is an error

- later on you cannot reassign to reference so if `int z = 12;` and `&y = z;` is invalid since y is already a reference of x

Strings

- 2 ways to create a string
    1. using string class
    2. using array of char
        - without pointer: `char str[] = "Hello";`
            - `char str[10]` can store 9 letters as 10th one has to be `\0` to identify the char array as a string
        - with pointer: `char *s = "Hello";`
- 2 ways to get user input
    - if using string class
        1. `cin>>str;` accepts one word
        2. `getline(cin,str);` accepts words with white spaces
    - if using array of char
        1. `cin>>str;` accepts one word
        2. `cin.getline(str,50);`
            - where 50 is the max length of string which is same as the length of the array of char
- 2 ways to find length(doesn't count `\0`)
    1. if using string class, use `str.length();`
    2. if using char array, use `strlen(str);` and `#include<cstring>`

- cstring
  - strcat(destination, source)
    - removes `\0` from source and merges them
    - strncat(destination, source, length)
      - the length of the source to be concatenated
  - strcpy(destination, source) or strncpy(destination, source, length)
  - strstr(str,subStr)
    - used to check for sub string `subStr` in string `str`
      - if found returns sub string from the first occurrence to the end of the string including the sub string itself
      - else it returns `NULL`
        - `NULL` cannot be printed by `cout` so write `cout` in an if block that checks if `strstr(str,subStr)` doesn't return `NULL`
  - strchr(str, ch)
    - used to check occurrence of `char ch` in `string str` and return the string ahead of it including ch itself
    - strrchr(str, ch) checks from Right to Left instead
  - strcmp(s1,s2) returns -ve, 0, +ve value

- `strtol(str,NULL,10)` converts string to long
    - 10 means decimal system, 2 means binary system and so on
- `strtof(str,NULL)` converts string to float
- tokeniser function strtok(str,str2)
    - `strtok(str,"=;")`

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char str[20] = "x=10;y=20;z=35";
    char *token = strtok(str, "=;");

    while (token != NULL)
    {
        cout << token << endl;
        token = strtok(NULL, "=;");
    }

    return 0;
}
/**
x
10
```

```
y
20
z
35

*/
```

- `strtok(str,";")`

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char str[20] = "x=10;y=20;z=35";
    char *token = strtok(str, ";");

    while (token != NULL)
    {
        cout << token << endl;
        token = strtok(NULL, ";");
    }

    return 0;
}
/**
x=10
y=20
z=35
```

```
*/
```

- string class
  - creates an array bigger than the number of characters to maintain some buffer capacity like vectors
  - if any modification that adds characters more than the available buffer capacity then it internally creates another copy of that array, makes the modification and still leaves some buffer capacity in the newly copied array
  - str.length() gives length of string excluding `\0`
    - str.size() also does the same
  - str.capcity() gives the actual capacity of the string including `str.length()` + 1(due to `\0`) + buffer capacity
  - str.resize(newCapacity) changes the capacity of `str` from `str.Capacity()` to `newCapacity` + buffer capacity
  - str.max_size() gives the max size of a string the compiler supports which depends on the CPU architecture and OS
  - str.clear() to empty a string

    - empty string contains only `\0`
        - so `str.length()` is 0

- `str.empty()` checks if the string is empty or not

- `str.erase()` is same as `str.clear()` but you can also pass index and can be used for trimming

- `str.append(anotherStr)` appends `anotherStr` at the end of string `str` by removing `\0` at the end of `str` before appending

- `str.insert(strIndex, snippetStr)` inserts `snippetStr` in `str` at `strIndex`
    - 0th index of `snippetStr` becomes `strIndex`th index of `str`
    - `str.insert(strIndex, snippetStr, n)` inserts substring of `snippetStr` which is till (n-1)th index

- `str.replace(strIndex, howMany, snippetStr)` replaces `howMany` letters from `strIndex`th index with `snippetStr`

- `str1.swap(str2)`

- stack operations
    - `str.push_back(ch)` adds char ch at the end of `str`
    - `str.pop_back()` removes last char of `str`
    - `str.front()` returns the first char of `str`
    - `str.back()` returns the last char of `str`

- `str.copy(chArray,str.length())`
  - copies string `str` into char array `chArray`
  - 2nd arg is the length of `str` to be copied
  - doesn't add `\0` so need to do it manually

```cpp
#include <iostream>
using namespace std;
int main()
{
    string str = "Welcome";
    char chArray[10];
    str.copy(chArray, 3); // `Wel` is stored in `chArray`
    chArray[3] = '\0';    // now `chArray` can be printed by `cout` as it identifies as a string
    return 0;
}
```

- `str.find(subStr)` or `str.find(ch)`
  - returns the index of first occurrence of `ch` in `str` if char is provided as argument
  - if string is provided then if the string `subStr` exists in `str` then the index of first occurrence of first letter of `subStr` in `str` is returned
  - `str.rfind(subStr)` or `str.rfind(ch)` does the same thing but from Right to Left
    - if not found then it returns an invalid index that doesn't lie between 0 to `str.length()`

- `str.find_first_of(ch)` or `str.find_first_of(ch, startingIndex)`

    - `startingIndex` is the index from where searching starts
    - gives the first occurrence of char `ch` in string `str`
    - str.find_last_of(ch) or str.find_last_of(ch, startingIndex)
    - if a string is given instead its not a syntax error
        - it traverses `str` char by char and as soon as a char at `str` also exists in the string argument then it returns the index for that char in `str`(which will always be the first occurrence)

- str.substr(lowerLim, upperLim)
    - returns sub string of `str` from index `lowerLim` till index `upperLim`
        - `upperLim` is not mandatory, when not provided, it is equal to `str.length()`
    - returned string includes the char at index `lowerLim` of `str`

- s1.compare(s2) same as `strcmp(s1,s2)` of `cstring`

- str.at(ind) returns char at index `ind` of string `str` and throws an exception when out of bounds
    - hence same as writing `str[ind]` which cannot throw an exception when out of bounds
        - string is a class so `[]` operator is overloaded
-
    - is an overloaded operator that concatenates strings

- = is an overloaded operator that copies the content of r value to l value

- string iterations
    - 2 ways: `iterator` and `reverse_iterator`

- the iterator is pointer for the array of chars in the string class and we use pointer arithmetic to iterate using the string class

```cpp
#include <iostream>
using namespace std;
int main()
{
    string str = "Hello World";

    for (string::iterator it = str.begin(); it != str.end(); it++)
        cout << *it;
    cout << endl;
    for (string::reverse_iterator it = str.rbegin(); it != str.rend(); it++)
        cout << *it;

    return 0;
}
```

- `rbegin` and `rend` are same as `begin` and `end` respectively but are used with `reverse_iterator` instead
- similar to `for(int i = 0; str[i] != '\0'; i++`

Functions

- avoid user input in functions, input should be passed from the calling function

- use tuples to return multiple values and if you only want to return array use object of array from STL which will not decay into pointer

- when a function has a local variable and a global variable of the same name then local variable is preferred as it has a smaller scope
    - use `::<nameOfVariable>` to access the global variable in a function that also has a local variable of the same name
    - even after a function ends the changes made to the global variable are preserved
        - this is useful, but I also want to make sure that the variable is not visible to all the functions of my program so the changes are only preserved for one function
            - so instead of declaring it as a global variable make it a static local variable of that function
                - they are located in the code section so their content is not lost after a function ends

- return types are not a deciding factor in function overloading

- function templates (generics)
    - overloaded functions with the same logic might just deal with different data type so we need to overload function for every possible data type which makes code verbose

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

int getmax(int a, int b) { return a > b ? a : b; }
float getmax(float a, float b) { return a > b ? a : b; }

int main()
{
```

```
    cout << getmax(4, 8) << endl;
    cout << getmax(4.8f, 8.4f); // without specifying f they are double as a default

    return 0;
}
```

- to avoid this we can create a generic data type using function templates

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

template <class T>
T getmax(T a, T b) { return a > b ? a : b; }

int main()
{
    cout << getmax(4, 8) << endl;
    cout << getmax(4.8f, 8.4f);

    return 0;
}
```

  - here the data type of T is decided dynamically
  - T can also be an object
  - data types of both a and b should be same here
    - if we pass an int and double then we get an error

- when generic functions are outside a generic class

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

template <class T>
class maths
{
    T a, b;

public:
    maths(T a, T b);
    T add();
    T sub();
};

template <class T>
maths<T>::maths(T a, T b)
{
    this->a = a;
    this->b = b;
}

template <class T>
T maths<T>::add()
{
    T c = a + b;
    return c;
}

template <class T>
T maths<T>::sub()
{
```

```
    T c = a - b;
    return c;
}

int main()
{
    maths<int> obj(1, 2);
    cout << obj.add() << endl;
    cout << obj.sub();
    return 0;
}
```

- default arguments of a function
  - a template function cannot have default arguments

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

int add(int x, int y, int z = 0) { return x + y + z; }

int main()
{
    cout << add(1, 2) << endl;
    cout << add(1, 2, 3);

    return 0;
}
```

- here the default arguments of variable z is 0 so if the value of z is not provided in the function call then its value fall backs to the default value
- this way using one function we can add 2 or 3 numbers
- allows to combine otherwise overloaded functions
- rule default arguments should be declared from right to left and you cannot have a variable with no default argument in between 2 variables which have default arguments
- all variables can have default arguments, but if one of them doesn't have one then it should be the leftmost variable and cannot lie between any 2 variables
  - `int add(int x = 0, int y, int z = 0)` is an error
  - `int add(int y, int x=0, int z = 0)` is correct
- it is a good practice to use `nullptr` as the default argument for functions that accept pointers, as it clearly indicates that the pointer is not pointing to a valid memory location
  - This can help prevent issues such as dereferencing a null pointer, which can lead to undefined behavior and crashes

- call by reference functions are handled differently by the compiler(provided pointers are not involved)
  - machine code of function called by reference gets appended to machine code where the function is called by reference
    - the function called by reference acts as if its content was in the calling function at the line where the function call happens
    - all functions that are called by reference become inline functions because their machine code get copied to function call
  - avoid loops in functions that are called by reference as the loop or any other complex code can change the nature of the call and it may not be by reference anymore

- return by address

- functions perform operation in HEAP and return an address without freeing it to the calling function which can be used to read and print by dereferencing the address

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

int *retByAdd(int size)
{
        int *p = new int[size];
        for (int i = 0; i < size; i++)
            p[i] = i + 1;

        return p;
}

int main()
{
        int *ptr = retByAdd(5);
        for (int i = 0; i < 5; i++)
            cout << ptr[i] << " ";

        delete[] ptr;
        ptr = nullptr;
        return 0;
}
```

  - you cannot return the address of a local variable as it will be destroyed after the function ends so only HEAP memory address can be returned
  - used to return arrays declared in HEAP

- return by reference
  - you cannot return the reference to a local variable as it will be destroyed after the function ends

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

int &retByRef(int &x) { return x; }

int main()
{
        int a = 10;
        cout << a << endl;
        retByRef(a) = 20;
        cout << a; // 20

        return 0;
}
```

- pointer to a function
  - rule: the name of the pointer variable whenever used with asterisk sign should be enclosed in round brackets

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

void hello() { cout << "hello world" << endl; }

int main()
```

```
{
    // 1st way
    void (*funptr)() = &hello;
    (*funptr)();

        // 2nd way
    void (*anotherfunptr)();
    anotherfunptr = hello;
    (*anotherfunptr)();

    return 0;
}
```

- using with arguments to a function

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

int min(int x, int y) { return x > y ? y : x; }

int max(int x, int y) { return x < y ? y : x; }

int main()
{
    int (*funptr)(int, int) = &min;
    cout << (*funptr)(10, 20) << endl;

    funptr = max;
    cout << (*funptr)(10, 20);
```

```
    return 0;
}
```

- passing array to function
  - when an array is passed as a parameter to a function, it decays into a pointer to the first element of the array

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

int search(const int A[], int len, int key)
{
    for (int i = 0; i < len; i++)
        if (key == A[i])
            return i;
    return -1;
}

int main()
{
    const int A[] = {2, 4, 5, 7, 10, 9, 13};

    int k;
    cout << "Enter an Element to be Searched:";
    cin >> k;

    int index = search(A, 7, k);

    if (index == -1)
    {
```

```
        cout << k << "not found";
        return -1;
    }

    cout << "Element found at index :" << index << endl;
    return 0;
}
```

OOPS

- size of a object = size of data types only as functions don't occupy any memory
- class doesn't occupy any space, object does as class is just a blueprint
- multiple objects have multiple copies of data members in the stack but all objects share the functions between them which are located in the code section
- pointer to object in STACK

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

class rect
{
public:
    float len, brd;
```

```
    void area() { cout << "Area is " << len * brd << " sq. units" << endl; }

    void peri() { cout << "Perimeter is " << 2 * (len + brd) << " units" << endl; }
};

int main()
{
    rect r1;
    rect *rectptr = &r1;
    rectptr->len = 123;
    rectptr->brd = 2;
    rectptr->area();
    rectptr->peri();

    return 0;
}
```

- pointer to object in HEAP

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

class rect
{
public:
    float len, brd;

    void area() { cout << "Area is " << len * brd << " sq. units" << endl; }
```

```
    void peri() { cout << "Perimeter is " << 2 * (len + brd) << " units" << endl; }
};

int main()
{
    rect *rectptr = new rect();
    rectptr->len = 123;
    rectptr->brd = 2;
    rectptr->area();
    rectptr->peri();

    delete rectptr;
    rectptr = nullptr;
    return 0;
}
```

- note that when using scope resolution operator `::` to write functions outside the class then don't need to rewrite the default values for parameters outside the class
    - it is considered re declaration of the function
- getters and setters / accessor and mutator
    - setter functions of a class are public functions that get the value as a function parameter and assign it to initialize private variables of their class
    - getters are public functions that just return the value of the private variable
    - if a variable has getter and no setter it becomes read only
    - if a variable has a setter and no getter then it becomes write only (example passwords)

- we should use constructor to assign default value instead of expecting that setters will always be used before getters and parameterized constructor can call setters as well
    - if getters are used before setters we might see garbage value and we thus we should assign default values
- ideally a class should have a copy constructor and a parameterized constructor with default values

- prefer writing functions outside class
    - functions declared outside the class have their machine code outside the main function's machine code
    - if they are declared in the class they become inline functions and their machine code is inside the machine code of main
        - inline function cannot have complex logic, so if you have complex logic use scope resolution operator for those functions
- so all functions defined in a class are inline and all functions defined outside it with scope resolution operator are not inline

- to make a function inline explicitly write `inline` before the return type of the function
    - inline functions improve performance as the function is not called but the code of the function is inserted where the function was called
    - inline functions should not have complex logic like loops or pointers and should be simple
- struct and class are same in c++ as both of them can store variables and functions unlike c where structs can only store variables.
    - the difference is that in a struct everything is public

- operator overloading

```
#include <iostream>
using namespace std;
class complex
{
public:
    int r, i;

    complex(int r = 0, int i = 0)
    {
        this->r = r;
        this->i = i;
    }

    complex operator+(complex c)
    {
        complex temp;
        temp.r = c.r + r;
        temp.i = c.i + i;
        return temp;
    }

} c1(1, 2), c2(3, 4), c3;

int main()
{
    c3 = c1 + c2;
    cout << c1.r << " + " << c2.r << " = " << c3.r << endl;
    cout << c1.i << "i + " << c2.i << "i = " << c3.i << "i ";
```

```
    return 0;
}
```

- friend operator overloading

```
#include <iostream>
using namespace std;
class complex
{
public:
    int r, i;

    complex(int r = 0, int i = 0)
    {
        this->r = r;
        this->i = i;
    }

    friend complex operator+(complex c1, complex c2);

} c1(1, 2), c2(3, 4), c3;

complex operator+(complex c1, complex c2)
{
    complex temp;
    temp.r = c1.r + c2.r;
    temp.i = c1.i + c2.i;
    return temp;
}
```

```
int main()
{
    c3 = c1 + c2;
    cout << c1.r << " + " << c2.r << " = " << c3.r << endl;
    cout << c1.i << "i + " << c2.i << "i = " << c3.i << "i ";

    return 0;
}
```

- reason for using friend: the friend function accepts 2 objects c1 and c2 of class complex
  - objects c1 and c2 are not available in the scope of the class

- insertion operator loading using `cout<<c`
  - where c is a complex number and we will overload insertion operator `<<` to print both real and imaginary parts
  - we will have to use a friend function and also pass `cout` to it

```
#include <iostream>
using namespace std;
class complex
{
public:
    int r, i;

    complex(int r = 0, int i = 0)
    {
        this→r = r;
        this→i = i;
    }
```

```
    friend ostream &operator<<(ostream &out, complex &c);
} cnum(1, 2);

ostream &operator<<(ostream &out, complex &c)
{
    out << c.r << " + " << c.i << "i";
    return out;
}

int main()
{
    cout << cnum;
    return 0;
}
```

- we are returning `ostream` so that insertion operator can be chained
  - `cout<<cnum<<endl;` is valid
  - if we replace `ostream` return type with `void` then we cannot chain insertion operator

Inheritance

- example of well designed class and inheritance(with copy constructor missing)

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

class rectangle
{
```

```
    int len, brd;
public:
    rectangle(int len = 0, int brd = 0);
    int getLen();
    int getBrd();
    int validator(int qty);
    void setLen(int len);
    void setBrd(int brd);
    int area();
    int peri();
};
rectangle::rectangle(int len, int brd) // default values are not repeated again
{
    setLen(len);
    setBrd(brd);
}
int rectangle::getLen() { return len; }
int rectangle::getBrd() { return brd; }
int rectangle::validator(int qty)
{
    if (qty < 0)
    {
        cout << qty << "is a non positive number, invalid input";
        exit(-1);
    }
    return qty;
}
void rectangle::setLen(int len) { this->len = validator(len); }
void rectangle::setBrd(int brd) { this->brd = validator(brd); }
int rectangle::area() { return len * brd; }
int rectangle::peri() { return 2 * (len + brd); }
```

```
class cuboid : public rectangle
{
    int hgt;
public:
    cuboid(int len = 0, int brd = 0, int hgt = 0);
    int getHgt();
    void setHgt(int brd);
    int volume();
};
cuboid::cuboid(int len, int brd, int hgt)
{
    setHgt(hgt);
    setLen(len);
    setBrd(brd);
}
int cuboid::getHgt() { return hgt; }
void cuboid::setHgt(int hgt) { this→hgt = validator(hgt); }
int cuboid::volume() { return getLen() * getBrd() * hgt; }
```

- if you are inheriting a class then you cannot rely on the constructor of that class
    - so you need to make sure that the constructor of the child function can also initialize variables of parent class
- constructor precedence in inheritance
    - if a child class is inheriting from the parent class in public mode such that a default constructor in parent class exists then when the object of the child class is created then the default constructor of parent class is

called first followed by a constructor of child class

- how to call the parameterized constructor of the parent class? when the constructor of the child class is called then its not immediately executed, instead the parameterized parent class constructor is called by giving it one parameter of the parameterized child class constructor. After the parameterized parent class constructor executes the control goes back to the parameterized base class constructor and then it gets executed

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

class papa
{
public:
    papa(int x) { cout << "Papa has " << x << endl; }
};

class son : public papa
{
public:
    son(int y, int z) : papa(z) // we write this becuase the constructor cannot be called
like a setter function.
    {
        cout << "son has " << y << endl;
        cout << "papa has " << z;
    }
};

int main()
{
```

```
    son baccha(10, 20);

    return 0;
}

// Papa has 20
// son has 10
// papa has 20
```

- isA and hasA
  - a class can be used by another class in two ways: one by inheriting it (isA) and one by creating an object of it (hasA)
  - when a class cuboid inherits a class rectangle, the cuboid is a rectangle
  - when a class table uses an object of rectangle as its data member then the table has a rectangle
- virtual parent classes
  - 2 parent classes inherit from a grandparent class and a child class inherits from both the parent classes
    - then the functions of the grandparent class can be accessed by 2 paths. This is multi path inheritance
      - to remove this ambiguity we use virtual parent classes
        - just add keyword `virtual` before inheritance mode for the 2 parents
          - virtual functions are preferably called compared to non-virtual functions
- inheritance modes:
  - if a class is inherited in public mode
    - private variables cannot be accessed but public remains public and protected remains protected

    - further inheritance is possible
- protected mode
    - public and protected variables and functions of the parent class become protected in the child class and private variables of parent are not accessible
    - further inheritance is possible but the protected members cannot be accessed outside the parent and child class
- private mode
    - public and protected variables and functions of the parent class become private in the child class and private variables of parent are not accessible
    - further inheritance is not possible as all variables of child are private

- 2 ways to use inheritance
    - to generalize : you have square, rectangle, parallelogram and trapezium so you can create a parent class quadrilateral and generalize the shapes
    - to specialize : you have parallelogram, you can create child classes square and rectangle to specialize the shape parallelogram

- parent class pointer and child class object `beta b; papa *papaptr = &b` or `papa *papaptr = new beta();`
    - if you use have a child class publicly inheriting from the parent class such that the child class object is created first and the parent class pointer points to the object of the child class (usually both should be same data type) and then you cannot call functions of the child class
    - suppose a class cuboid is the child class which inherits from parent class rectangle such that the pointer points to rectangle papa and is allocated heap address using new cuboid which is the child instead of rectangle papa then using pointer we can only call the function of rectangle papa and not the function of cuboid beta

- the opposite: child class pointer and parent class object is not possible and is an error
- the constructors are called from child to parent

- function overriding
  - if the child class beta inherits from parent class papa such that there is a function which has the same name and parameters in both the classes and an object of beta is created and that function with common name is called, then how do we get to control which of the 2 functions are called?
    - example if a beta is a new car and papa is the old car which has manual windows and the new car has power windows then the function is same but there is variation
  - without making any effort to address this, if an object of beta is created and that function is called with a common name the compiler calls the one in the child class as it is closer in scope
  - if the name of the two functions are same and there is a variation in the parameters it takes then whenever you call the papa's functions then it becomes function overloading instead of overriding
  - if we have the same situation (same named functions in 2 classes that have inheritance relationship) with parent class pointer and child class object and call that function the parent class function is called and not the one nearer to the scope which is in the beta class
    - if 2 functions have common name in separate classes having a inheritance relationship with parent class pointer and child class object then if we make on such function virtual by adding virtual keyword before return type then the non virtual function is called.

- so we can demonstrate run time polymorphism by using virtual functions with function overriding with parent class pointer and child class object

```
#include <iostream>
using namespace std;
class anyCar
{
```

```
public:
    virtual void start() = 0;
    // {cout << "started a generic car" << endl;}
    virtual void stop() = 0;
    // {cout << "stopped a generic car" << endl;}
};

class swift : public anyCar
{
public:
    void start() { cout << "started swift" << endl; }
    void stop() { cout << "stopped swift" << endl; }
};
class innova : public anyCar
{
public:
    void start() { cout << "started innova" << endl; }
    void stop() { cout << "stopped innova" << endl; }
};

int main()
{
    anyCar *ptr = new swift();
    ptr->start();
    ptr->stop();
    ptr = new innova();
    ptr->start();
    ptr->stop();

    return 0;
}
```

- generalization of swift and innova so we can get away by just declaring empty virtual functions as we will never start a generic car, it has to have some model but when you do this make virtual functions = 0 then they are called pure virtual functions
  - a class with pure virtual function is called an abstract class
  - we cannot create objects of abstract classes but you can make pointer objects to them
- 3 possible use cases of inheritance:
  - if all functions of the class are non virtual then the its use is re-usability
  - if some functions are virtual then both uses re-usability and polymorphism
  - and if all virtual functions then only use is to achieve polymorphism and such a class is called an interface

- Friend functions
  - functions outside the class which want to access the private and protected variables/functions can be declared as friend functions inside the class to give them the read only access
  - we add `friend` before the return type of the function in the class
  - a friend relationship cannot modify data and is read only

- friend class
  - if an external class uses an object of another class as its data members then if it wants to access the private and protected variables/functions we use a friend class
  - declare the class which wants to access private and protected variables/functions as friend class inside the class whose object its creating
  - add line `friend <name of the class that wants to access the private and protected variables/functions using objects of the class this line is in>`
  - a friend relationship cannot modify data and is read only

- static variables
  - memory allocated by declaring a variable as static is shared between objects and changes to them are persistent across the lifetime of objects
  - static members can be used outside the class using scope resolution operator without having to create an object of the class
- static functions are similar but cannot access non static variables and can be called without creating an object of the class they belong to
- nested class can access members of outer class only if they are static
  - but it can create an object of the outer class to access its members
  - a nested class acts independently as if it were an outer class

exception handling

- used during runtime errors
- if any exception is thrown in the try block then the control moves to the catch block
  - the remaining lines of try block are not executed
  - catch block will not execute if no exceptions thrown in try block
- `throw var` then `var` needs to be a parameter of the catch block
  - `var` usually used to throw codes

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

```cpp
int divider(int x, int y)
{
    if (y == 0)
        throw 404;
    return x / y;
}

int main()
{
    try
    {
        divider(10, 0);
        cout << "I am bout to divide by 0";
    }
    catch (int err)
    {
        cout << "Oops, ";
    }
    cout << "After Try Catch";

    return 0;
}
// Oops, After Try Catch
```

- the main use of try catch is when a function is expected to return a value but is not able to then it should throw an exception
    - if we know that a function can throw an exception then call it in try block
- you can throw any data type including object of a class
    - when throwing an object of a class, a good practice is to make sure that the class that might throw exception is inheriting publicly from c++'s built in class called exception

    - throwing a constructor will create the object and then throw
- you can declare what a function throws by appending `throw (<data type thrown>)` at the end of the function signature but this is optional
    - if you keep this round bracket empty means that the function throws no exception

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

int divider(int x, int y)
{
    if (y == 0)
        throw string("DivZer0");
    return x / y;
}

int main()
{
    try
    {
        divider(10, 0);
        cout << "Lets Divide!" << endl;
    }
    catch (const string &err)
    {
        cout << "Oops, Error:" << err << endl;
    }
    cout << "After Try Catch";

    return 0;
}
```

```
// Oops, Error:DivZer0
// After Try Catch
```

- you can write multiple catch blocks for single try block that handle different data types of thrown variables
  - if you do `catch(...)` then the catch block can catch all thrown exceptions regardless of their data type
  - `catch(...)` should be the last catch block and not the first one if you are using multiple catch blocks
    - otherwise if the catch all block is at the top then it will handle all exceptions and the catch blocks below it will be unused
  - if you are throwing objects of multiple classes in a try block such that the classes have an inheritance relationship then the catch block at the top should deal with the child class and the next catch block should deal with the parent class

Template functions and classes

- used for generic programming

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

template <class T>
T adder(T x, T y) { return x + y; }

int main()
{
    cout << adder(10, 5) << endl;
    cout << adder(3.14, 6.023) << endl;

    return 0;
}
```

- the above function can only accept variables that are of the same data type

- for a function to accept 2 generic variables of two different types

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

template <class T, class R>
R adder(T x, R y) { return x + y; }

int main()
{
    cout << adder(10, 3) << endl;
    cout << adder(10, 3.14) << endl;

    return 0;
}
```

  - we can still pass two variables of the same data type

- useful if you are going to create generic data structures and perform operations on them

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

template <class T>
class Stack
{
```

```
    T *stk;
    int top;
    int size;

public:
    Stack(int sz)
    {
        size = sz;
        top = -1;
        stk = new T[size];
    }
    void push(T x);
    T pop();
};
template <class T>
void Stack<T>::push(T x)
{
    if (top == size - 1)
        cout << "Stack is Full";
    else
    {
        top++;
        stk[top] = x;
    }
}
template <class T>
T Stack<T>::pop()
{
    T x = 0;
    if (top == -1)
        cout << "Stack is Empty" << endl;
```

```
    else
    {
        x = stk[top];
        top--;
    }
    return x;
}
int main()
{
    Stack<float> s(10);
    s.push(10);
    s.push(23);
    s.push(3.14);
    s.push(6.023);
    return 0;
}
```

pointers and constants

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;

    int *ptr = &x;
    (*ptr)++;

    int const *ptr2 = &x;
```

```
    // (*ptr2)++; error
    int y = 20;
    ptr2 = &y;
    // (*ptr2)++; error

    int *const ptr3 = &y;
    (*ptr3)++;
    // ptr3 = &x; error

    const int *const ptr4 = &y;
    // (*ptr4)++; error
    // ptr4 = &x; error

    cout << x << " " << y << " " << *ptr2 << " " << *ptr4;
    return 0;
}
// 11 21 21 21
```

- if variable is const and a pointer `ptr` is pointing to it then the pointer `ptr` should be const of the data type not as a pointer which it can also be at the same time but will require 2 const in declaration
- you can still point to the const of a data type of a variable without it having to be a constant
- if we write `const` at the end of a function signature in a class then the compiler will not allow that functions to modify values of the class
  - you can also make a parameter in call by reference const to add some restrictions on what a function can do with the variables
  - if you still want to change a private variable in a constant function of a class then you can declare the variable with `mutable` keyword
- if a class is marked final by adding keyword final after the class name then the class cannot be inherited further
  - in C++, the `final` keyword can be used to prevent a virtual function from being overridden in child classes

- smart pointer
  - deallocates itself when pointer goes out of scope

preprocessor directives

- if a constant is needed across all files

```
#define x 10
#define c cout
#include <iostream>
using namespace std;
int main()
{
    c << x;
    return 0;
}
```

- we can also define functions

```
#define sqr(x) (x * x)
#define show(x) (cout << x);
#include <iostream>
using namespace std;
int main()
{
    show(sqr(5));
    return 0;
}
```

- we use ifndef to def only if not defined already

```
#ifndef sqr
#define sqr(x) (x * x)
#endif

#ifndef show
#define show(x) (cout << x);
#endif

#include <iostream>
using namespace std;
int main()
{
    show(sqr(5));
    return 0;
}
```

Destructors

- called when the object goes out of the scope
- if the object is created in HEAP and deleted then the destructor gets called
- if a file is opened in a class then it should be closed in its destructor
- or any HEAP allocations made in the class should be deleted in its destructor
- during inheritance
    - first the parent constructor is called
    - then the child constructor is called
    - then the child destructor is called

- then the parent destructor is called
- if we have a parent pointer and child object and used delete pointer then
  - only parent class destructor will be called
  - to fix this make the parent class destructor virtual
    - then parent class destructor is called then child class destructor

Streams

- for input and output
- `iostream` has `istream` for input and `ostream` for output
- `cin` is a class of `istream` and `cout` is a class of `ostream`
- insertion and extraction operator are used and sometimes overloaded
- File handling
  - writing

```
#include <fstream>
using namespace std;
int main()
{
    ofstream outfile("hello.txt");
    // if the file doesn't exist, it is created anf if it does then it becomes empty
    // if you dont want to lose the pre existing data and want to append instead specify append mode app or truncate mode trunc(default)
    // ofstream outfile("hello.txt",ios::app);
    outfile << "HI" << endl;
    outfile << 123 << endl;
    outfile.close();
```

```
        return 0;
}
```

- reading

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream infile;
    infile.open("hello.txt");
    if (!infile.is_open())
        cout << "Cannot open file, maybe invalid name";
    string str;
    int n;
    infile >> str;
    infile >> n;
    cout << str << endl << n;
    infile.close();
    return 0;
}
```

Lambda functions

- they are unnamed functions
- `[](){cout<<"Hello World";}();`
- `[](int a , int b){cout<<"Sum is = "<<a+b;}(10,5);`

- `int x = [](int a , int b){return a+b;}(10,5);`
- `auto f = [](){cout<<"Hello World";};` and f() can be used later
- we can specify return type but not compulsory `int x = [](int a , int b)->int{return a+b;}(10,5);`
- accessing variables outside the lambda expression

```cpp
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c = 30;
    [&a, &b](){ cout << a << " " << b<< endl; }();
    [&](){ cout << a << " " << b << " " << c; }();
        return 0;
}
```

ellipsis `...`

- allows a function to take multiple args