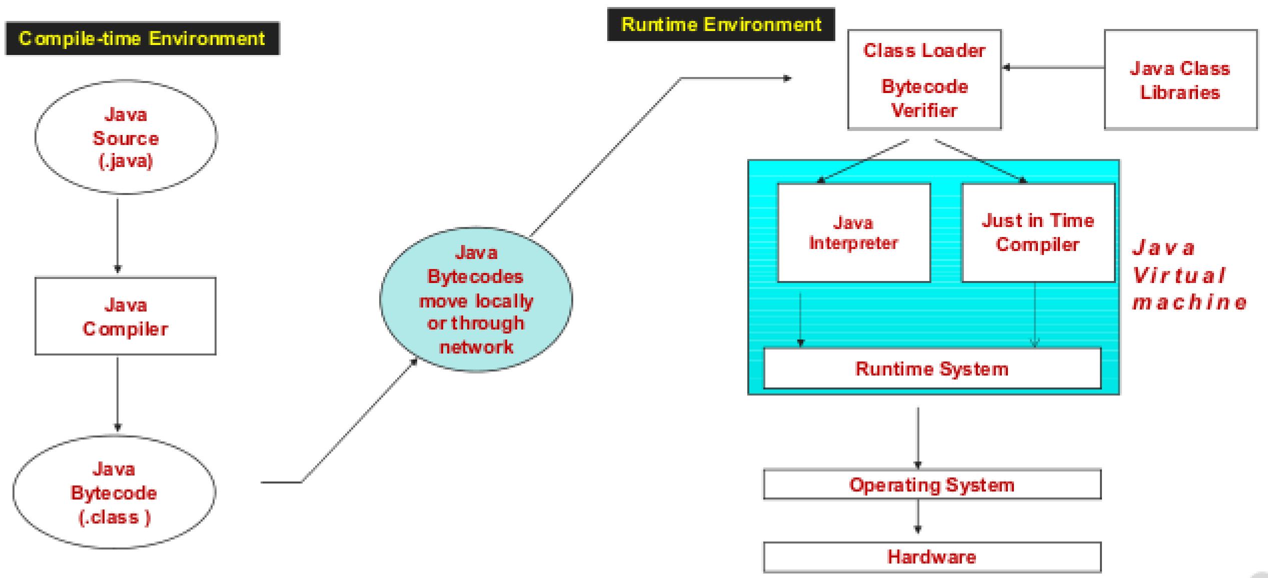
JAVA C C++

JAVA

- December '90 by Patrick Naughton, Mike Sheridan and James Gosling
- January 27, 2010, Oracle - Sun Microsystems
- Java is case-sensitive ; maxval, maxVal, and MaxVal are three different names



- Simple and Powerful
- Object Oriented
- Portable
- Distributed
- Multi-threaded
- Robust, Secure/Safe
- Interpreted
- High Performance
- Dynamic programming language/platform

```
import java.Math.*;        // all classes from java.Math

public class HelloWorld { // starts a class
        public static void main (String[] args) {
            // public = can be seen from any package
        // static = not “part of” an object
                // starts a main method
                // in: array of String; out: none (void)
```

```
                System.out.println("Hello World");
        }
}
```

```
javac HelloWorld.java  // Produces HelloWorld.class (byte code)

java HelloWorld        // Starts the JVM and runs the main method
```

Data types & Variables

- byte 8 bit
- short 16 bit
- int 32 bit
- long 64 bit
- float 32 bit
- double 64 bit
- boolean true or false
- char 16 bit, Unicode
- Use the keyword final to specify a constant
- **Enumeration (enum)**
  - enum in java is a data type that contains fixed set of constants

```
class EnumExample1{
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        public static void main(String[] args) {
                for (Season s : Season.values())
                        System.out.println(s);
                }
                for (Season s : Season.valueOf()){
                        System.out.println(s+" "+s.value);
                }
        }
}
```

- **Annotation**
  - Java Annotation is a tag that represents the metadata
  - Attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler & JVM
  - It is an alternative option for XML and java marker interfaces
    1. @override
    2. @Depricated
    3. @SupressWarning

Enhanced for loop

```
public static int sumListEnhanced(int[] list){
        int total = 0;
        for(int val : list){
                total += val;
                System.out.println( val );
        }
        return total;
}
```

Arrays of objects

```
public void objectArrayExamples(){
        Rectangle[] rectList = new Rectangle[10];
        // How many Rectangle objects exist?
        rectList[5].setSize(5,10);
        for(int i = 0; i < rectList.length; i++){
                rectList[i] = new Rectangle();
        }
        rectList[3].setSize(100,200);
}
```

Object Oriented View

- Class & Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Static Initializer Block (SIB)

- If we wanted some statements to execute before the main function starts, then write the code in SIB

Scanner class

```
Scanner sc = new Scanner(System.in);
int x = sc.nextInt();
```

```java
float y = sc.nextFloat();

String str = sc.next();
```

The protected Modifier

- Allows a child class to reference a variable or method directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible to any class in the same package as the parent class

The super Reference

- If the child constructor invokes the parent (constructor) by using the super reference, it must be the first line of code of the constructor

Method Overriding

- If a method is declared with the final modifier, it cannot be overridden

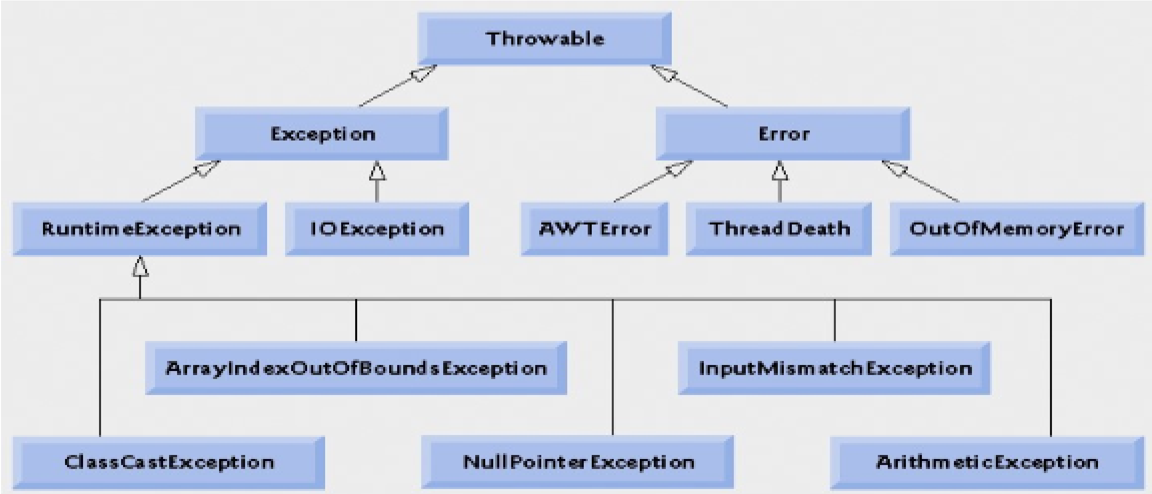
The Object Class

- java.lang package
- Ultimate root of all class hierarchies
- eg methods : toString, equals

Abstraction

- An abstract method cannot be defined as `final` or `static`

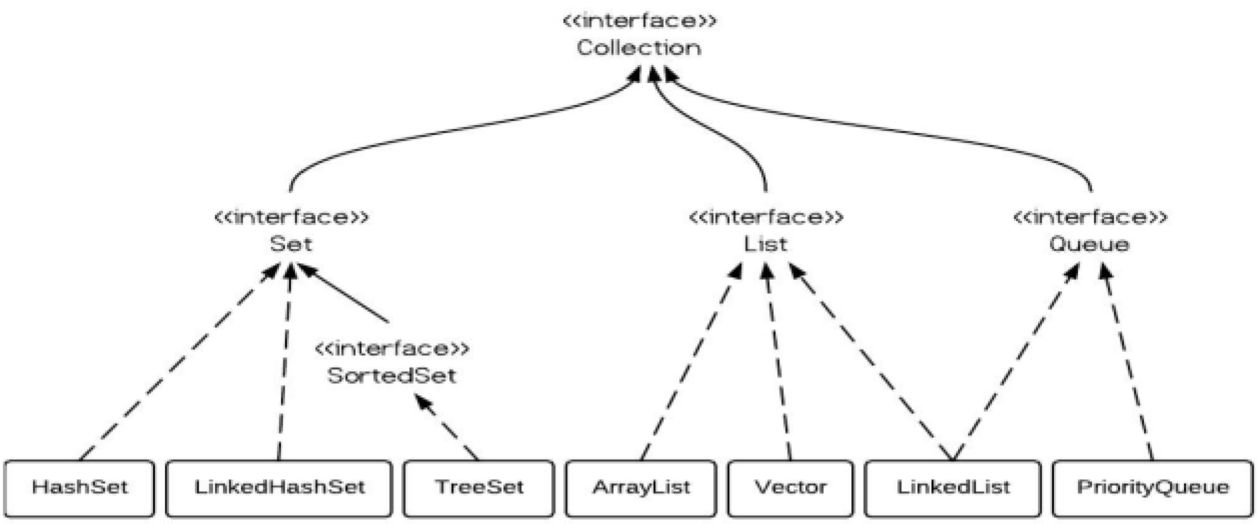
Exception Handling



- **Checked Exceptions**
  - The compiler will issue an error if a checked exception is not caught or asserted in a throws clause
  - eg : IOException - file handling
- **Unchecked Exceptions**
  - Does not require explicit handling
  - RuntimeException or any of its descendants

Collection Framework

- Code to interface: Since the collections framework is coded on the principles of “Code to Interface”, it aids in inter-operability between APIs that are not directly relate



C

Program Development Process

- **Problem solving & Implementation phase**

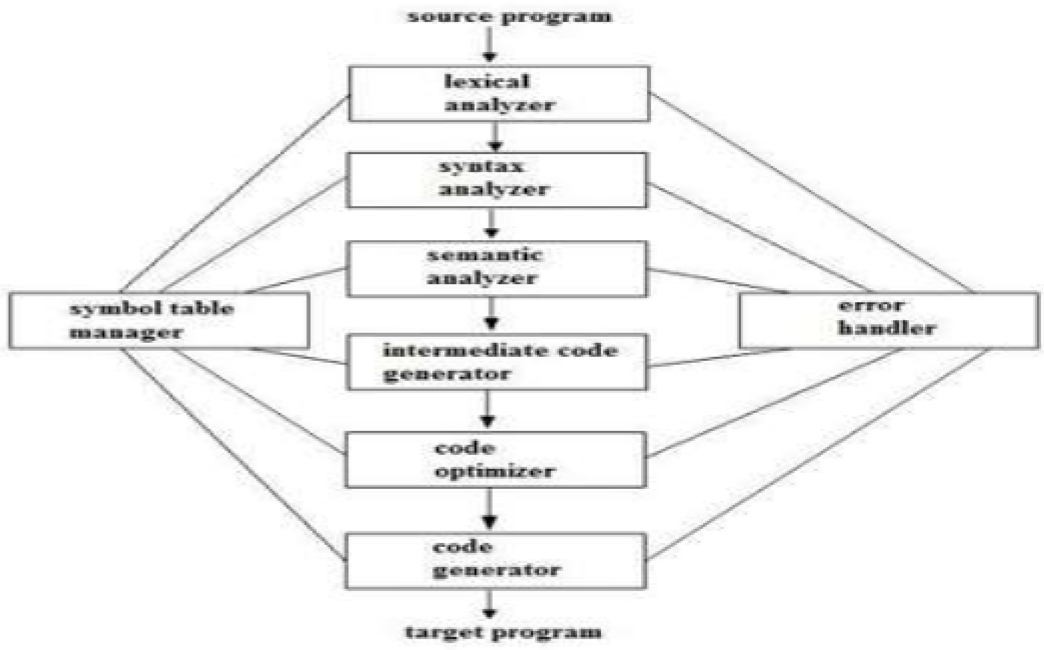
1. Program documentation
2. User needs determination
3. Design & Review program specifications
4. Design the Algorithm
5. Coding
6. Compile, Test & Debug
7. Program deployment

Language-Translators Programs

- **Assembler** : assembly -> machine
- **Compiler** : high-level -> machine
- **Interpreter** : high-level -> machine

Stages of Compilation

1. **Preprocessing** : processes include-files, conditional compilation instructions and macros
2. **Compilation** : takes the output of the preprocessor, and the source code, and generates assembler source code
3. **Assembly** : takes the assembly source code and produces an assembly listing with offsets (object file)
4. **Linking** : takes one or more object files or libraries as input and combines them to produce a single(usually executable) file



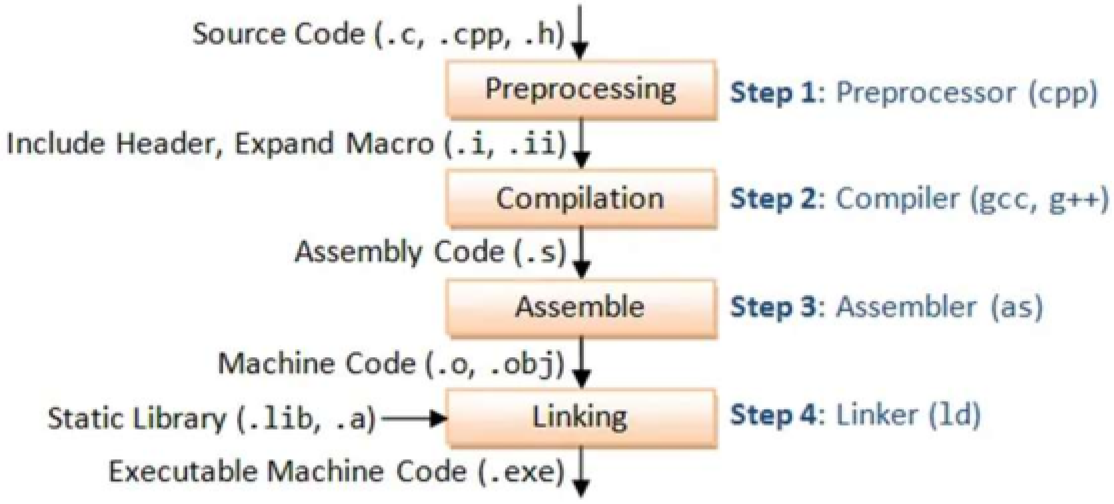
Introduction to C-Programming

- C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T and BELL LABS.

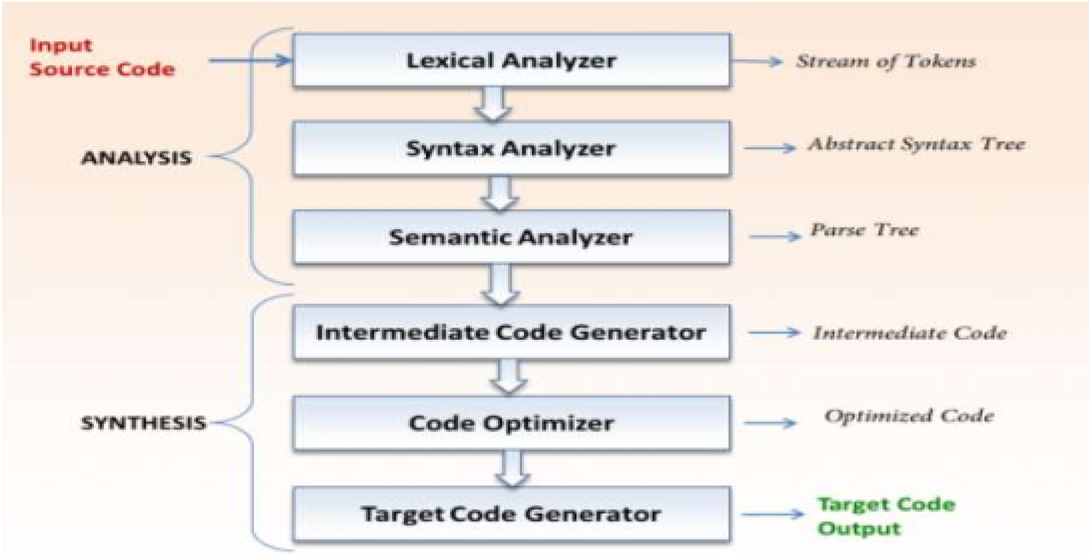
GCC (GNU Compiler Collection)

```
gcc main.c
gcc main.c -o main

gcc -Wall main.c -o main // enable all warnings
gcc -Wall -E print.c // print the preprocessed output to stdout
gcc -S main.c > main.s // produce only the assembly code
gcc -C main.c // produce only compiled code
gcc -save-temps main.c // produce all the intermediate files
```

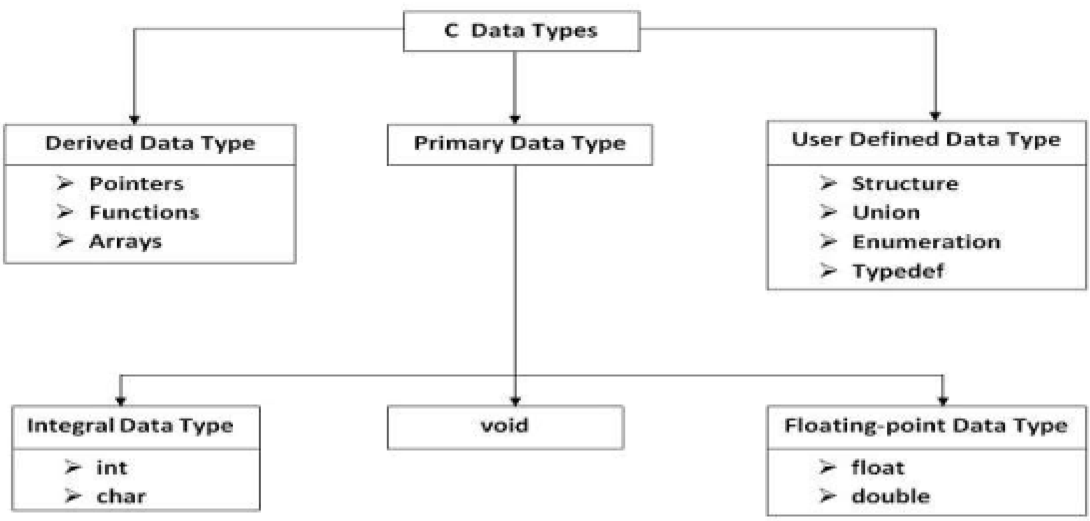
Stages of Compilation



Phases of Compiler



Introduction to C-Language



Lexical Elements

- C - Tokens
  1. Identifiers
  2. Keywords
  3. Constants
     - Literal & Character constants
     - `\b` Backspace character
     - `\f` Form feed
     - `\n` Newline character
     - `\r` Carriage return
     - `\t` Horizontal tab
     - `\o, \oo, \ooo` Octal number
     - `\xh, \xhh, \xhhh` Hexadecimal number
  4. Strings
  5. Operators
  6. Special Symbols

Modifiers

- Modify the meanings of the predefined built-in data types and expand them to a much larger set
- int (2/4 bytes)
- 4 data type modifiers in C
  - **long** (4 bytes)
  - **short** (2 bytes)
  - **signed**
  - **unsigned**

Type Qualifiers

- Can prepend to variable declarations which change how the variables may be accessed
- 2 types type qualifiers in C
  - **const** : causes the variable to be read-only
  - **volatile** : to prevent the compiler from applying any optimizations on objects or variables

Storage Class

- These specifiers tell the compiler how to store the subsequent variable
- `storage_specifier type var_name`

1 auto

- automatic / local variables
- auto variables are stored in stack segment of the process address space

2 static

- Permanent variables within their own function or file
- They are not known outside their function or file, but they maintain their values between calls
- The key difference between a static local variable and a global variable is that the static local variable remains known only to the block in which it is declared

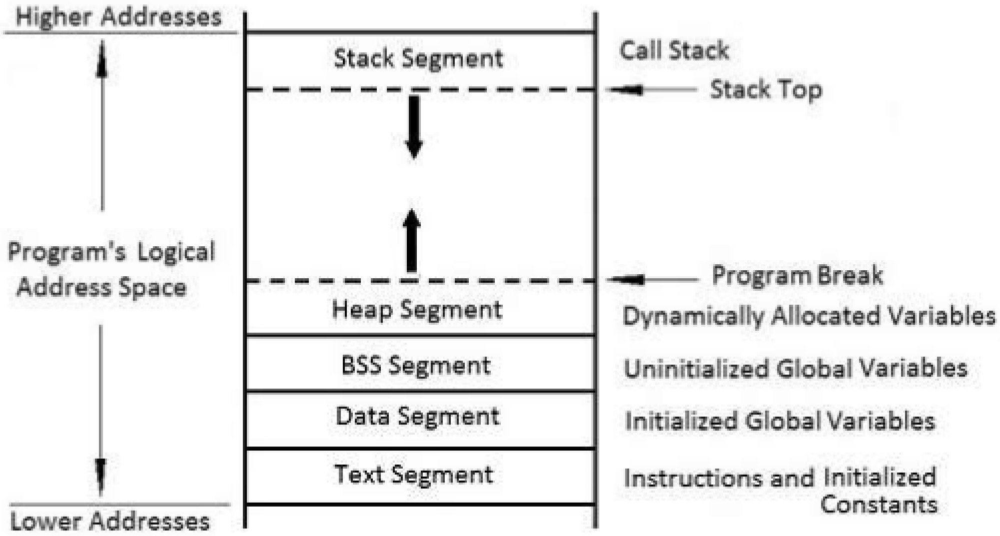
3 extern

- Allows separate modules of a large program to be compiled and linked together
- `extern` informs the compiler of the types and names of global variables without creating storage for them again

4 register

- Originally applied only to `int`, `char`, or pointer types, but now applies to any type
- To keep the value of a variable kept in a CPU register for faster access

Memory Layout in C



Expressions

- An expression consists of at least one operand and zero or more operators

Operators

- Unary operators
- Binary Operators
- Ternary Operators
- Unary `[ + - ! ~ ++ - - (type)* & sizeof ]`
- Arithmetic Operators `[ * , / , + , - ]`
- Relational Operators `[ <,>,==,!=,<=,>= ]`
- Logical Operators `[ &&,||,! ]`
- Bitwise Operators `[ << ,>>,^,&,|, ]`
- Ternary or Conditional Operators `[ ?: ]`
- Assignment Operator `[ = += -= *= /= %=>>= <<= &= ^= |= ]`

swap - logic

num1 = num1 + num2
num2 = num1 - num2
num1 = num1 - num2

Functions have three parts

- **Function Prototype (declarations)** : consists of information regarding the function's name, return types and names of parameters

- **Function invocation** : calling the function
- **Function Definitions** : body of the function

Recursion

```
int factorial(int n){
        if(n == 0)
                return 1;
        return (n * factorial(n - 1));
}
```

```
int fib(int num){
/* Fibonacci value of a number */
        switch(num) {
                case 0 : return(0); break;
                case 1 : return(1); break;
                default: return(fib(num - 1) + fib(num - 2)); break;
        }
}
```

- Stream : a sequence of characters flowing from one place to another
    - input stream : data flows from input device (keyboard, file, etc) into memory
    - output stream : data flows from memory to output device (monitor, file, printer, etc)

Precision Modifier

- `h` the argument is a pointer to type short instead of type int
- `l` the argument is a pointer to type long or double
- `L` the argument is a pointer to type long double
- `#` ensures that there will be a decimal point even if there are no decimal digits
- `*` ensures that minimum field width and precision specifiers can be provided by arguments

Strings & Formatting

sprintf

- Write formatted data to string
- `int sprintf ( char * str, const char * format, ... )`

```
int main (){
        char buffer [50];
        int n, a=5, b=3;
        n = sprintf (buffer, "%d + %d = %d", a, b, a+b);
        printf ("[%s] is a string %d chars long\n",buffer,n);
        return 0;
}
```

sscanf

- Read formatted data from string
- `int sscanf ( const char * s, const char * format, ...)`

```
int main() {
        char* buffer = "Hello";
        char store_value[10];
        int total_read;
        total_read = sscanf(buffer, "%s" , store_value);
        printf("Value in buffer : %s\n",store_value);
        printf("Total items read : %d",total_read);
        return 0;
}
```

strtof

- Convert string to float
- `float strtof (const char* str, char** endptr)`

strtod

- Convert string to double
- `double strtod (const char* str, char** endptr)`

strtol

- Convert string to long integer
- `long int strtol (const char* str, char** endptr, int base)`

strtoll

- Convert string to long long integer
- `long long int strtoll (const char* str, char** endptr, int base)`

strtoull

- Convert string to unsigned long long integer
- `unsigned long long int strtoull (const char* str, char** endptr, int base)`

atoi

- Convert string to integer
- `int atoi (const char * str)`

atol

- Convert string to long integer
- `long int atol ( const char * str )`

atof

- Convert string to double
- `double atof (const char* str)`

fflush

- Flush stream
- The data is forced to be written to disk
- If stream is a null pointer, all such streams are flushed.
- `int fflush ( FILE * stream )`

fpurge

- Data is discarded
- `void __fpurge(FILE *stream)`

getc, fgetc& gets

- getc & fgetc are primarily used to read characters from disk files
- `int getc ( FILE * stream )`
- gets reads characters from the standard input (stdin)
- `char * gets ( char * str )`
- A pointer is a variable that contains the address of another variable

```
int main(){
        int ix = 5;
        int *iPtr = NULL;
        iPtr = &ix;
        printf("ix = %d",ix);        //ix=5
        printf("iPtr = %p",iPtr);    //iPtr=0x7ffe053c3154
        printf("&ix = %p",&ix);      //&ix=0x7ffe053c3154
        printf("*iPtr = %p",*iPtr); //*iPtr=5
        printf("&iPtr = %p",&iPtr); //&iPtr=0x7ffe053c3158
        printf("It is address of x *&p=%p",*&iPtr); //It is address of
*&p=0x7ffe053c3154
}
```

void pointer

- Do not have any type associated with them
- Can hold the address of any type of variable

Pointers & Arrays

- (data == &data[0]) is true : array name is a pointer to the array's first element
- `*(ptr++)`, `*(++ptr)`, `*ptr++`, `*++ptr` - affects the index
- `(*ptr)++`, `++(*ptr)` - affects the content

```
ptr_array = iarray;
ptr_array = &iarray[0];
```

```
int *p, i[10];
p = i;
p[5] = 100;   /* assign using index */
*(p+5) = 100; /* assign using pointer arithmetic */
```

```
int main(){
    int arr[4]={20,30,40,50};
    int *ptr = arr;
    printf("%p %d \n",(ptr+0),*(ptr+0)); // 0x7fff9ba68ff0 20
```

```
    printf("%p %d \n",(ptr+1),*(ptr+1)); // 0x7fff9ba68ff4 30
    printf("%p %d \n",(ptr+2),*(ptr+2)); // 0x7fff9ba68ff8 40
    printf("%p %d \n",(ptr+3),*(ptr+3)); // 0x7fff9ba68ffc 50

    printf("%p %d \n",(arr+0),*(arr+0)); // 0x7fff9ba68ff0 20
    printf("%p %d \n",(arr+1),*(arr+1)); // 0x7fff9ba68ff4 30
    printf("%p %d \n",(arr+2),*(arr+2)); // 0x7fff9ba68ff8 40
    printf("%p %d \n",(arr+3),*(arr+3)); // 0x7fff9ba68ffc 50
    return 0;
}
```

Pointers & 2-D Arrays

- `p` pointer to first row
- `p+i` pointer to ith row
- `p+i+j` pointer to ith row jth column
- `*(p+i)` pointer to first element in the ith row
- `*(p+i)+j` pointer to jth element in the ith row
- `*(*(p+i)+j)` value stored in the cell

Command-Line Arguments

- To pass command-line arguments or parameters to a program when it begins executing\
  - `argc` for argument count
  - `argv` for argument vector

Pointer constant

- `const int *p`
- Defines p as a pointer to a constant integer

Constant Pointer

- `int * const p`
- Defines a constant pointer to an integer

Function Pointer

```
returntype (*ptr-to-fn)(arguments if any)
returntype (ptr-to-fn)(arguments if any)
```

Pointer to Pointer

```
int x = 12;       // x is a type int variable
int *ptr1 = &x; // ptr is a pointer to x
int **ptr2 = &ptr; //ptr2 is a pointer to a pointer to type int
```

Type Conversions

- **Automatic Type Conversions** : lower type is promoted to the higher type before the operation proceeds
- **Integral Promotion** : If an int can represent all the values of the original type, then the value is converted to int, otherwise the value is converted to unsigned int

Dynamic Memory Allocation

1 malloc

- `void *malloc(size_t size)`
- malloc returns a pointer to space(memory) for an object of size size, or NULL if the request cannot be satisfied
- malloc will request space from the OS
- free storage is kept as a list of free blocks - first fit

2 realloc

- `void *realloc(void *p, size_t size)`
- realloc returns a pointer to the new space, or NULL if the request cannot be satisfied
- realloc changes the size of the object pointed to by p to size

3 calloc

- `void *calloc(size_t nobj, size_t size)`
- calloc returns a pointer to space for an array of n obj objects, each of size size, or NULL if the request cannot be satisfied

C Stream

- **Text stream** : it is sequence of characters. C allows a text stream to be organized into lines terminated by a new line character
- **Binary streams**

fopen

- `FILE *fopen(const char *path, const char *mode)`
- **file opening modes**
    - `r` read-only
    - `r+` read & write
    - `w` write/create/over-write
    - `w+` read & write/create/over-write
    - `a` append
    - `a+` read & append/create

fclose

- `int fclose(FILE *fp)`
- Upon successful completion this function returns 0 else end of file (eof) is returned

fgetc

- `int fgetc ( FILE * stream )`
- Get character from stream

fputc

- `int fputc ( int character, FILE * stream )`
- Write character to stream and advances the position indicator

fgets

- `char * fgets ( char * str, int num, FILE * stream )`
- Reads characters from stream and stores them as a C string into str until (num-1)

fputs

- `int fputs ( const char * str, FILE * stream )`
- Writes the C string pointed by str to the stream

fprintf

- `int fprintf ( FILE * stream, const char * format, variable )`
- Write formatted data to stream

fscanf

- `int fscanf ( FILE * stream, const char * format, variable )`
- Read formatted data from stream

fread & fwrite

- for reading/writing data from/to the file opened by fopen function

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

size_t fwrite(void *ptr, size_t size, size_t nmemb,FILE *stream)
```

fseek

- `int fseek(FILE *stream, long offset, int whence)`
- To set the file position indicator for the stream to a new position
- SEEK_SET, SEEK_CUR, SEEK_END

rewind

- `void rewind ( FILE * stream )`
- Set position of stream to the beginning

ftell

- `ftell(FILE *fp)`
- To determine the current location of a file

feof

- `int feof ( FILE * stream )`
- Check end-of-file indicator

ferror

- `int ferror ( FILE * stream )`
- Check error indicator

Structure and Functions

- Structure object as an argument to a function
    - object access by `object.memb`

```
void print_student (struct student)
```

- Structure pointer as an argument to a function
    - object access by `object->memb`
    - If p_str is a pointer to the structure str, the following expressions are all equivalent
        - `str.memb`
        - `(*p_str).memb`
        - `p_str->memb`

```
void read_student_p(struct student *)
```

typedef and Structures

- keyword used to create a synonym for a structure or union type
- To create an object, we need not use key word struct

union

- Derived type of structure
- Provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program
- The union variable u will be large enough to hold the largest of the types in it

enumerations

- user defined types that have integral values, associated with each enumeration is a set of named constants
- Behave like integer constants
- `enum type_name{ value1, value2,...,valueN }`

```
enum day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10,
friday, saturday};

int main(){
        printf("%d %d %d %d %d %d %d", sunday, monday, tuesday, wednesday,
thursday, friday, saturday);
        return 0;
}

1 2 5 6 10 11 12
```

Bit fields

- Specify size (in bits) of structure and union members
- A special unnamed bit field of size 0 is used to force alignment on next boundary
- Cannot be static
- Cannot have pointers to bit field members

```
struct test1{
        double d;
        unsigned int data1;
        char c;
        unsigned long long data2;
};
struct test2{
        double d;
        unsigned int data1;
        unsigned long long data2;
        char c;
};

int main(){
        printf("Size of test1 is %d bytes\n", sizeof(struct test1)); // 24
bytes
        printf("Size of test2 is %d bytes\n", sizeof(struct test2)); // 32
bytes
        return 0;
}
```

Pre-Processor

- `#define`
- `#undef`
- **concatenation operator (##)** : concatenates, or joins, two strings in the macro expansion
- `#if` , `#elif` , `#else` , & `#endif` : control conditional compilation
- **conditional selection using `#ifdef` , `#else` and `#endif`**

GNU gdb

- Debug programs written in C and C++
- Allows you to see inside another program
- Functions like an interpreter
- Start, stop, examine, and change your program during execution

Invoking gdb

```
$ gcc -g debug_me.c -o debug_me // compile your program with the `-g` flag
$ gdb debug_me // start gdb
(gdb) break main // set a breakpoint at main
(gdb) break main // run the program
```

gdb commands

```
(gdb) help // to get a list of commands
(gdb) kill // stop
(gdb) quit // exit
(gdb) continue // continue

gdb programname // start debugger with program name
(gdb) run arg1 "arg2" // start execution

(gdb) kill // kill
(gdb) run  // restart

(gdb) set x = 3
(gdb) print x

(gdb) next // go over function calls
(gdb) step // go into function calls
(gdb) call abort() // call function
```

```
(gdb) backtrace // view backtrace
(gdb) frame 2 // change stack frames

// set breakpoints
(gdb) break 19
(gdb) break test.c:19
(gdb) break func1
(gdb) info breakpoints // list breakpoints

(gdb) watch x // set watchpoint
(gdb) disable 4 // disable watchpoint
```

Breakpoints

- A fundamental debugging tool used to pause the execution of a program at a specific point
- This allows developers to inspect the state of the program, including variable values and the call stack, at that exact location

Watchpoints

- A type of breakpoint that pauses the execution of a program when the value of a specified variable changes
- This is particularly useful for tracking down bugs that occur due to unexpected changes in variable values

GNU - Makefile

- A makefile is a special file, typically named `Makefile`
- It contains a set of directives used by the `make` build automation tool to compile and build programs
- It specifies how to derive the target program from the source files
- **Naming Makefiles**
  - Default names: `GNUmakefile`, `makefile`, `Makefile`
  - Use `-f` or `--file` option for nonstandard names
- **Processing Makefiles**
  - **Phase-1 :** Read files, expand variables/functions, process directives, construct dependency graph.
  - **Phase-2 :** Determine targets to rebuild, invoke rules
- Structure/Components of Makefile

1. **Targets**
    - Usually a file generated by a program
    - The output files or actions to be performed, such as executables or object files
    - A target can also be an action, like `clean`
2. **Prerequisites**
    - Input files for the target
    - Files or conditions that must be satisfied before the target can be built
    - These are usually source files that need to be compiled
3. **Recipes**
    - Commands to create the target (tab-indented)
    - The commands to be executed to build the target from the prerequisites
    - These commands are typically shell commands
4. **Rules**
    - Each rule in a makefile consists of a target, prerequisites, and a recipe
    - **Explicit Rules :** Specify prerequisites for a specific target
    - **Implicit Rules :** Utilize known patterns (e.g., .c to .o).
    - structure of Rule
    `makefile    target: prerequisites         recipe`
5. **Variables**
    - Store values that can be used throughout the makefile to avoid repetition and make the file easier to maintain
    - Defining Variables

```
$ make VAR1=abc VAR2=xyz // command-line
override VAR1=dummy      // override directive to prevent redefinitions
```

Automatic Variables

- Set by make after a rule is matched
- `$@` : Target filename
- `$*` : Target filename without extension
- `$<` : First prerequisite filename
- `$^` : All prerequisite filenames, no duplicates
- `$+` : All prerequisite filenames, includes duplicates
- `$?` : Newer prerequisites than the target

Example - Makefile

1. example of a simple makefile

```
# Variables
CC = gcc
CFLAGS = -Wall -g

# Targets and rules
all: myprogram

myprogram: main.o utils.o
    $(CC) $(CFLAGS) -o myprogram main.o utils.o

main.o: main.c
    $(CC) $(CFLAGS) -c main.c

utils.o: utils.c
    $(CC) $(CFLAGS) -c utils.c

clean:
    rm -f myprogram main.o utils.o
```

- **Variables**
  - `CC` and `CFLAGS` are defined to specify the compiler and flags used during compilation
- **all**
  - A target that depends on `myprogram`
  - This target is usually the default target when `make` is run without arguments
- **myprogram**
  - Depends on `main.o` and `utils.o`
  - The recipe combines these object files into the final executable
- **main.o** and **utils.o**
  - These are object files generated from their respective source files
- **clean**
  - A special target to remove generated files, which helps to clean up the build environment

2. example2 Makefile

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
```

```
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

3. example3 with Automatic Variables

```
all: hello.exe

hello.exe: hello.o
    gcc -o $@ $<

hello.o: hello.c
    gcc -c $<

clean:
    rm hello.o hello.exe
```

CASE STUDY - Makefile

Write functions to find Area and Perimeter of following Triangles in each files

- An equilateral triangle
- An isosceles triangle
- A right-angled triangle
- A scalene triangle
  Write main function in main.c & Write a makefile for this project
- equilateral_triangle.c

```
#include <math.h>

double equilateral_area(double side) {
    return (sqrt(3) / 4) * side * side;
}
```

```c
double equilateral_perimeter(double side) {
    return 3 * side;
}
```

- equilateral_triangle.h

```c
#ifndef EQUILATERAL_TRIANGLE_H
#define EQUILATERAL_TRIANGLE_H

double equilateral_area(double side);
double equilateral_perimeter(double side);

#endif // EQUILATERAL_TRIANGLE_H
```

- isosceles_triangle.c

```c
#include <math.h>

double isosceles_area(double base, double side) {
        double height = sqrt(side * side - (base * base /4));
        return (base * height) / 2;
}

double isosceles_perimeter(double base, double side){
        return 2* side + base;
}
```

- isosceles_triangle.h

```c
#ifndef ISOSCELES_TRIANGLE_H
#define ISOSCELES_TRIANGLE_H

double isosceles_area(double base, double side);
double isosceles_perimeter(double base, double side);

#endif // ISOSCELES_TRIANGLE_H
```

- right_angled_triangle.c

```c
#include <math.h>

double right_angled_area(double base, double height){
        return ( base + height ) / 2;
}
```

```c
double right_angled_perimeter(double base, double height){
        double hypotenuse = sqrt(base * base + height * height);
        return base + height + hypotenuse;
}
```

- right_angled_triangle.h

```c
#ifndef RIGHT_ANGLED_TRIANGLE_H
#define RIGHT_ANGLED_TRIANGLE_H

double right_angled_area(double base, double height);
double right_angled_perimeter(double base, double height);

#endif // RIGHT_ANGLED_TRIANGLE_H
```

- scalene_triangle.c

```c
#include <math.h>

double scalene_area(double a, double b, double c){
        double s =  ( a + b + c ) / 2;
        return sqrt(s * ( s -a ) * ( s - b ) * ( s - c ));
}

double scalene_perimeter(double a, double b, double c){
        return a + b + c;
}
```

- scalene_triangle.h

```c
#ifndef SCALENE_TRIANGLE_H
#define SCALENE_TRIANGLE_H

double scalene_area(double a, double b, double c);
double scalene_perimeter(double a, double b, double c);

#endif // SCALENE_TRIANGLE_H
```

- main.c

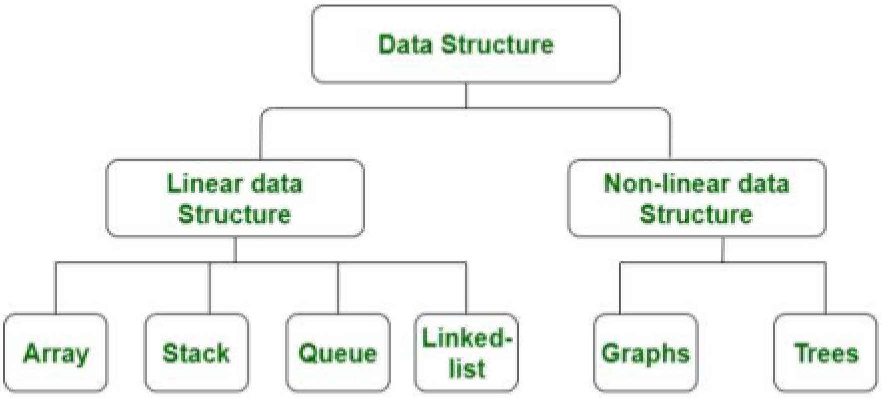
```c
#include <stdio.h>
#include "equilateral_triangle.h"
#include "isosceles_triangle.h"
#include "right_angled_triangle.h"
#include "scalene_triangle.h"
```

```c
int main(){
        double side, base, height, a, b, c;
        printf("Enter side base height a b c = ");
        scanf("%f%f%f%f%f%f",&side,&base,&height,&a,&b,&c);

        printf("Equilateral Triangle:\n");
        printf("Area: %f\n", equilateral_area(side));
        printf("Perimeter: %f\n\n", equilateral_perimeter(side));

        printf("Isosceles Triangle:\n");
        printf("Area: %f\n", isosceles_area(base, side));
        printf("Perimeter: %f\n\n", isosceles_perimeter(base, side));

        printf("Right-Angled Triangle:\n");
        printf("Area: %f\n", right_angled_area(base, height));
        printf("Perimeter: %f\n\n", right_angled_perimeter(base, height));

        printf("Scalene Triangle:\n");
        printf("Area: %f\n", scalene_area(a, b, c));
        printf("Perimeter: %f\n\n", scalene_perimeter(a, b, c));

        return 0;
}
```

- Makefile

```makefile
CC = gcc
CFLAGS = -Wall - Werror -std=c99

OBJECTS = main.o equilateral_triangle.o isosceles_triangle.o
right_angled_triangle.o scalene_triangle.o

all : triangle_calculator

triangle_calculator : $(OBJECTS)
        $(CC) $(CFLAGS) -o triangle_calculator $(OBJECTS)

main.o : main.c equilateral_triangle.h isosceles_triangle.h
right_angled_triangle.h scalene_triangle.h
        $(CC) $(CFLAGS) -c main.c

equilateral_triangle.o : equilateral_traingle.c equilateral_traingle.h
        $(CC) $(CFLAGS) -c equilateral_triangle.c

isosceles_triangle.o : isosceles_triangle.c isosceles_triangle.h
        $(CC) $(CFLAGS) -c isosceles_triangle.c

right_angled_triangle.o : right_angled_triangle.c right_angled_triangle.h
        $(CC) $(CFLAGS) -c right_angled_triangle.c
```

```
scalene_triangle.o : scalene_triangle.c scalene_triangle.h
	$(CC) $(CFLAGS) -c scalene_triangle.c

clean :
	rm -f *.o triangle_calculator
```



Applications of Stack (LIFO)

- To convert some infix expression into its postfix equivalent, or prefix equivalent
- Expression evaluation
- Function call implementation
- Parsing in compiler design
- Parenthesis Balance Checking
- Track nested function calls
- Undo Button

Applications of Queue (FIFO)

- Call Center phone systems to hold people calling them in an order, until a service representative is free
- Serving requests on a single shared resource, like a printer
- Data getting transferred between the IO Buffers (Input Output Buffers)
- CPU scheduling and Disk scheduling
- Managing shared resources between various processes
- Job scheduling algorithms

Linked List

- List Insertion and deletion is fast and less expensive
- Used for applications which do not need random access

- Random access is not allowed
- Extra memory space for a pointer is required with each element of the list
- Slow in accessing the elements in the worst case , if list is too large

Applications of Linked Lists

- Used to implement stacks, queues, graphs, etc
- Can insert elements at the beginning and end of the list
- In Linked Lists we don’t need to know the size in advance

C++

Structured Programming

- Emphasis on algorithm rather than data
- Programs divided into procedures
- Procedures are mostly independent
- eg : Pascal and C

History of OOP Languages

- Simula (1962) and Smalltalk (1969)
- "C with Classes" in 1979 by Bjarne Stroustrup, became C++ in 1983
- C++98 to C++17
- Java developed in 1995 by James Gosling

Benefits of OOP

- Class concept for data and methods
- Reusability and dynamic-ness in code
- Secure data hiding
- Popular languages : C++, Java, Python, etc

Applications of OOP

- Real-time systems, OORDBMS, AI, CAD/CAM, System Software, Office Automation, Neural Networks, Parallel Programming

Difference between struct and class

- `struct` members are public by default
- `class` members are private by default

Object Oriented Programming

- 5 Pillars of OOP
    1. Class & Object
        - Class defines the nature of an object, containing data and functions
        - Object is an instance of a class
    2. Abstraction
        - Simplifying complex real-world objects
        - Well-defined public interfaces for interaction
    3. Encapsulation
        - Hiding implementation details
        - Attributes and behaviors kept together
    4. Inheritance
        - Derived class inherits from base class
        - Adds unique attributes and behaviors
    5. Polymorphism
        - Ability to exist in multiple forms
        - Overridden behaviors across subclasses

Access Specifiers

- **private**
    - to encapsulate or hide the member data in the class
- **public**
    - to expose the member functions to the outside world,\that is, to outside functions as interfaces to the class

this pointer

- Created automatically by the compiler
- Contains the address of the object through which the function is invoked

```
class Simple {
private:
        int id;
public:
        void setID(int id) {
                this->id = id;
        }
        int getID() {
                return this->id;
        }
};
```

Scope Resolution Operator ::

- Allows the body of the member functions to be separated from the body of the class
- To define the function outside the class
- To access the global Variables
- To define the static variables
- To invoke the static functions

```
return_type class_name :: function_name () {}

data_type class_name :: variable_name = value ;
```

Static Class Members – Static Data Members

- Only one copy of that variable will exist
- All objects of that class will share that variable
- Not stored in objects

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

class static_demo{
private:
    static int a;
    int b;
public:
    void set ( int i, int j){
        a = i;
        b = j;
    }
    void showValues();
```

```
};

int static_demo :: a = 20;

void static_demo :: showValues(){
    cout << "static a : " << a << " non-static b : " << b << '\n';
}

int main(){
    static_demo x, y;
    x.set(1, 1);
    x.showValues(); // a = 1 & b = 1
    y.set(2, 2);
    y.showValues(); // a = 2 & b = 2
    x.showValues(); // a = 2 & b = 1
    return 0;
}
```

A mutable Object Member

- A const member function cannot modify the state of its object
- A mutable member is never const, even if its object is const ; therefore, it can be modified by a const member function

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

class CMF {
    mutable int value;
public:
    CMF(int v = 0) { // constructor
        value = v;
    }

    int getValue() const{
        value = 100; /* if mutable keyword was not used, we get compiler
error */
        return value;
    }
};

int main(){
    CMF t(50); // calling constructor + object creation
    cout << "value : " << t.getValue() << endl; // value : 100
    return 0;
}
```

Function Overloading

- Process of using the same name for two or more functions
- Functions either have different types of parameters or a different number of parameters or different return types

Constructors

- Invoked automatically when an object is created
- Declared in the public section
- Does not have return types

```
class_name()
```

Constructor - Initializing using initializer list

- Initialization list executes more faster than normal constructor
- Assignment within constructor cannot be used if try-throw-catch statements have to be included

```
class X{
        int a;
        float f;
public:
        X(int j, float x) : a(j) , f(x) , b(a) {} /* this->a = j, this->f
= x,      b = this->a */
```

Named constructors

- Declare all the constructors in the private section and you provide public static methods to return an object
- These static methods are called the Named Constructors

```
inline class_name :: constructor ( ) {}

inline class_name constructor :: function_name ( ) {}
```

- `inline` suggests that the compiler should replace the function call with the function code to reduce overhead
- object initialization is efficient, especially since it's a simple assignment operation

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

class Point {
    float x, y;
    Point(float x, float y) : x(x), y(y) { }
public:
    static Point rectangular(float x, float y) {
        return Point(x, y);
    }

    static Point polar(float radius, float angle) {
        return Point(radius * cos(angle), radius * sin(angle));
    }

    void show() const {
        cout << "x = " << x << ", y = " << y << endl;
    }
};

int main() {
    Point p1 = Point :: rectangular(3.0, 4.0);
    Point p2 = Point :: polar(5.0, 0.927295);
    p1.show();
    p2.show();
    return 0;
}
```

Copy constructors

- Constructor function with the same name as the class and used to make deep copy of objects
- if a class has a pointer variable, a copy constructor has to be defined
- When an object is created from another object of the same type
- When an object is passed by value as a parameter to a function
- When an object is returned from a function

```
constructor a;
constructor b(a);
```

Explicit constructors

- Constructor with only one required parameter is considered an implicit conversion function
- It converts the parameter type to the class type
- Not depends on the semantics of the constructor

```
test() { i=100; j=200; }
explicit test(int x) { i=x; j=x+10; }
test(int x , int y){ i = x; j = y; }
```

Array of objects

```
class Array{
        int i, j;
public:
        Array(int x, int y) : x(i), y(j) { }
};

int main() {
        Array obj[3] = { Array(1,2), Array(3,4), Array(5,6) };
        // Array obj[3] = { (1,2), (3,4), (5,6) };
        return 0;
}
```

Destructors

- Invoked implicitly when the object is destroyed
- Clean up and release resources
- Never call a destructor
- Does not take any parameter and does not return any value

```
~class_name(){
        delete variable;
}
```

Static Member Functions

- Static member functions can only access static data members
- They do not have a `this` pointer
- Cannot be virtual or const/volatile
- Useful for initializing static data before any object is created

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

class DemoStatic {
private:
    static int i;  // Static data member

public:
    static void init(int x) {
        i = x;  // Static member function
    }

    void show() const {
        cout << i << endl;  // Non-static member function
    }
};

int DemoStatic::i = 0; // Define the static member variable

int main() {
    DemoStatic::init(100); // Initialize static data before object
creation
    DemoStatic obj1;
    obj1.show();  // Displays : 100

    // Modify the static member through another object
    DemoStatic obj2;
    obj2.init(200);
    obj1.show();  // Displays : 200

    return 0;
}
```

Initialization of const static Data Members

- Integral types can be initialized inside the class
- Non-integral types must be defined outside the class

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

class Buff {
private:
    static const int MAX = 512;  // Initialization & definition inside the class
    static const char FLAG = 'a';// Initialization & definition inside the class
    static const string MSG;// Declaration inside the class - non-intergral type

public:
    void show() const {
        cout << "MAX: " << MAX << endl;
        cout << "FLAG: " << FLAG << endl;
        cout << "MSG: " << MSG << endl;
    }
};

// Definition outside the class for non-integral types
const string Buff :: MSG = "Hello, World!";

int main() {
    Buff b;
    b.show();  // Displays MAX, FLAG, and MSG values
    return 0;
}
```

Call by Reference

- Copies the reference of an argument into the formal parameter
- Changes made to the parameter affect the actual argument

```
void swap(int &x, int &y) {
        int temp = x;
        x = y;
        y = temp;
} // swap(a,b);
```

Inline Functions

- Declared with the `inline` keyword or defined inside a class
- Can be used instead of macros for small functions to improve performance

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

class Math {
public:
    inline int add(int a, int b) {
        return a + b;
    }

    inline int subtract(int a, int b);
};

inline int Math :: subtract(int a, int b) {
    return a - b;
}

int main() {
    Math math;
    int x = 10;
    int y = 5;
    cout << x << "+" << y << "=" << math.add(x, y) << endl;
    cout << x << "-" << y << "=" << math.subtract(x, y) << endl;
    return 0;
}
```

Default Function Arguments

- Default values must be specified once in the function declaration
- Parameters with default values must appear to the right of non-default parameters

Constant Arguments

- Parameters passed as constant cannot be changed inside the function

Objects as Arguments

- Objects can be passed as arguments to initialize data members of another object

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

class Circle {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double getRadius() const {
        return radius;
    }
};

// Function that calculates the area of a Circle object
double calculateArea(const Circle &c) {
    const double PI = 3.14159265358979323846;
    return (PI * c.getRadius() * c.getRadius());
}

int main() {
    Circle circle(5.0);
    // Calculate the area by passing the Circle object to the function
    double area = calculateArea(circle);

    cout << "The area of the circle " <<  " = " << area << endl;
    return 0;
}
```

- Functions can return objects

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

class Rectangle {
private:
    double length, width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}

    void display() const {
        cout << "Length: " << length << " Width: " << width << endl;
    }
```

```
    // Static method to create a square
    static Rectangle createSquare(double sideLength) {
        return Rectangle(sideLength, sideLength);
    }
};

int main() {
    // Using the static method to create a square
    Rectangle square = Rectangle :: createSquare(7.0);
    cout << "Square dimensions : " << endl;
    square.display();
    return 0;
}
```

Function Overloading and Ambiguity

- Ambiguity occurs when the compiler cannot decide between overloaded functions
- Main cause of ambiguity involves C++'s automatic type conversions
- Ambiguity can be caused by using default arguments in overloaded functions
- Two functions cannot be overloaded when the only difference is that call-by-reference & call-by-value parameter

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

void f(int x) {
    cout << "In f(int)" << endl;
}

void f(int &x) {
    cout << "In f(int &)" << endl;
}

int main() {
    int a = 10;
    f(a); // This will call f(int &)
    f(10); // This will call f(int)
    return 0;
}
```

Dynamic memory Concepts using new and delete

- `new` : Allocates memory dynamically and returns a pointer to the allocated memory
- `delete` : Deallocates memory that was previously allocated by `new`

```
int number = 88;
int *p1 = &number;  // Static allocation

int *p2 = nullptr;  // Dynamic allocation
p2 = new int;       // Allocates memory dynamically
*p2 = 99;

delete p2;          // Deallocates memory
p2 = nullptr;
```

`new[]` and `delete[]` Operators

- Dynamic Arrays : Allocated at runtime using `new[]`
- Deallocating Arrays : Use `delete[]` instead of `delete`

```
int *Array = new int[5];
delete[] Array;
```

Dynamic Memory Allocation and Initialization

- Initializing allocated memory using initializers or constructors

```
int *p1 = new int(88);
double *p2 = new double(1.23);

Date *date1 = new Date(1999, 1, 1);
Time *time1 = new Time(12, 34, 56);
```

Pointers to Classes & Objects

- Objects can be dynamically allocated and pointed to by pointers
- Used for dynamic creation and deallocation of objects

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

class Rectangle{
private:
    int width,height;
public:
    Rectangle(int w, int h){
        width = w;
```

```
        height = h;
    }

    int area(){
        return width*height;
    }

    void display(){
        cout << " area : " << area() << endl;
    }
};

int main(){
    Rectangle obj(3,4);
    obj.display();                    // area : 12

    Rectangle *rect1,*rect2,*rect3;
    rect1 = &obj;
    rect1->display();                 // area : 12

    rect2 = new Rectangle(5,6);
    rect2->display();                 // area : 30

    rect3 = new Rectangle[2]{{7,8}, {9,10}};
    rect3[0].display();               // area : 56
    rect3[1].display();               // area : 90

    delete rect2;                     // Deallocate memory
    delete[] rect3;
    return 0;
}
```

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

class Student{
private:
    int num;
    string name;
public:
    Student() : num(0), name("") {}
    Student(int n, string s) : num(n), name(s) {}

    void setData(int n, string s){
        num = n;
        name = s;
        cout << "roll number : " << num << " name : " << name << endl;
    }
};
```

```
int main(){
    Student *ptr;
    int i,j,num;
    string name;
    cout << "No of many students : ";
    cin >> j;

    ptr = new Student[j];  // dynamic allocation of array of j student
objects
    for(i=0; i<j; i++){
        cout << "enter student details : ";
        cin >> num >> name;
        ptr[i].setData(num,name);
    }

    delete[] ptr;         // deallocate memory
    return 0;
}
```

Memory Leaks

- Memory leaks occur when dynamically allocated memory is not deallocated

Dangling Pointers and Wild Pointers

- Pointers that do not point to a valid object of the appropriate type
- **Dangling Pointers** : Pointers that point to memory that has been deallocated
- **Wild Pointers** : Pointers that have not been initialized and point to arbitrary memory locations

```
char *dp = nullptr;
{
  char c;
  dp = &c;
}
// dp is now a dangling pointer

char *wp;  // wp is a wild pointer
static char *scp; /* scp is not a wild pointer:
```

Creation and Using References

- A reference acts as an alias to another object or value
- Can be used as function parameters, return values, or stand-alone references

```
int a = 5;
int &ref = a;  // ref is a reference to a
ref = 10;      // a is now 10
```

Call-by-Value and Call-by-Reference

- **Call-by-Value** : Passing a copy of the argument to the function
- **Call-by-Reference** : Passing the address of the argument to the function

```
void fnnegate(int ival) { ival = -ival; } // call-by-value
void fnnegate(int *ival) { *ival = -*ival; } // call-by-reference using pointer
void fnnegate(int &ival) { ival = -ival; } // call-by-reference using reference parameter
```

Pointers

- Can be initialized anytime
- Can be reinitialized any number of time
- Can be null
- Require `*` to dereference

References

- Must be initialized when created
- Cannot reinitialize a reference
- Cannot be null
- Automatically dereferenced
- Passing object by reference - no copy of obj is made

Independent References

- References that are variables, must be initialized, cannot change the object they refer to

```
int a;
int &ref = a;
a = 10;     // a is now 10
ref = 100;  // a is now 100
```

Return by Reference

- `dataType& functionName(parameters)`
- Functions can return references, making code easier to read and maintain

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

int x;

int& retByRef() {
    return x;
}

int main() {
    retByRef() = 10;
    cout << x;  // Output will be 10
    return 0;
}
```

Inheritance Basics

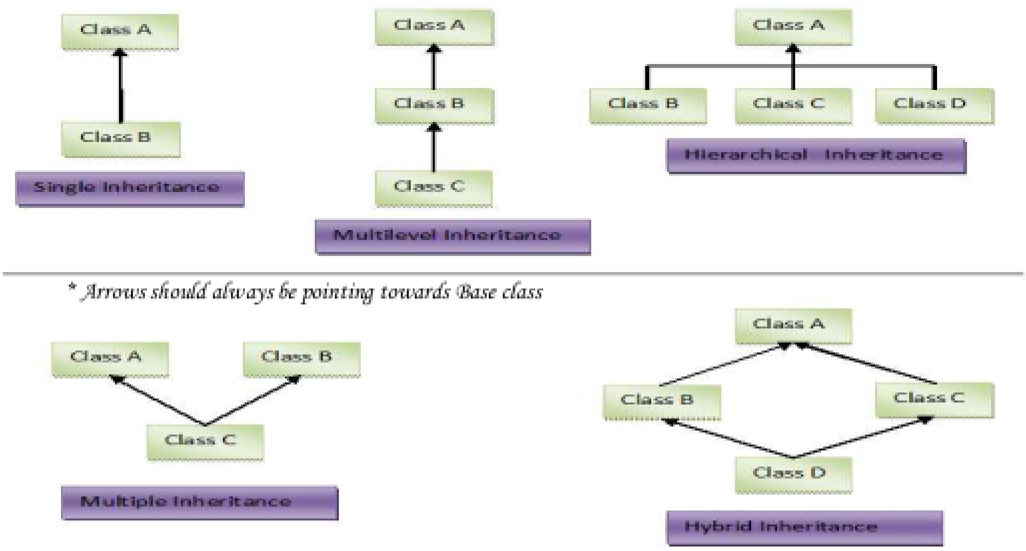
- Inheritance is a mechanism where a new class (derived class) is created from an existing class (base class), inheriting its properties and behaviors
- Reuse existing code and create a hierarchical relationship between classes
- `class derived-class-name : access base-class-name`

Access Specifiers

- **Public** : Members of the base class are accessible by the derived class and other parts of the program
- **Private** : Members of the base class are not accessible by the derived class (default)
- **Protected** : Members of the base class are accessible by derived classes but not by other parts of the program

Types of Inheritance

- **Single Inheritance** : A class inherits from one base class
- **Multiple Inheritance** : A class inherits from more than one base class
- **Multilevel Inheritance** : A class inherits from a derived class, forming a chain
- **Hierarchical Inheritance** : Multiple classes inherit from a single base class
- **Hybrid Inheritance** : A combination of two or more types of inheritance



Constructors and Destructors in Inheritance

- Constructors of the base class are executed before the derived class
- Destructors of the derived class are executed before the base class
- In multiple inheritance, constructors are executed in the order of derivation and destructors in the reverse order

Single Inheritance : Constructor & Destructor

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

class base{
public:
    base(){
        cout << "constructing base\n";
    }
    ~base(){
        cout << "destructing base\n";
    }
};

class derived : public base{
public:
    derived(){
        cout << "constructing derived\n";
```

```
    }
    ~derived(){
        cout << "destructing derived\n";
    }
};

int main(){
    derived ob;
    return 0;
}
```

Multi-Level Inheritance : Constructor & Destructor

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

class base{
public:
    base(){
        cout << "constructing base\n";
    }
    ~base(){
        cout << "destructing base\n";
    }
};

class derived1 : public base{
public:
    derived1(){
        cout << "constructing derived1\n";
    }
    ~derived1(){
        cout << "destructing derived1\n";
    }
};

class derived2 : public base{
public:
    derived2(){
        cout << "constructing derived2\n";
    }
    ~derived2(){
        cout << "destructing derived2\n";
    }
};

int main(){
    derived1 obj1;
    derived2 obj2;
```

```
    return 0;
}
```

Multiple Inheritance : Constructor & Destructor

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

class base1{
public:
    base1(){
        cout << "constructing base1\n";
    }
    ~base1(){
        cout << "destructing base1\n";
    }
};

class base2{
public:
    base2(){
        cout << "constructing base2\n";
    }
    ~base2(){
        cout << "destructing base2\n";
    }
};


class derived : public base1, public base2{
public:
    derived(){
        cout << "constructing derived\n";
    }
    ~derived(){
        cout << "destructing derived\n";
    }
};

int main(){
    derived ob;
    return 0;
}
```

Replicated base classes

- Derived classes will have duplicate sets of members inherited from the base

- More than one copy of the base is visible
- This creates ambiguity
- If the base class has a public member i, they can be accessed by specifying derived1::i, derived2::i

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

class base{
    public: int i;
};

class derived1 : public base{
    public: int j;
};

class derived2 : public base{
    public: int k;
};

class derived3 : public derived1, public derived2{
public:
    int sum;
};

int main(){
    derived3 ob;
    ob.derived1::i = 50;
    ob.j=90;
    ob.k=80;
    int sum = ob.derived1::i + ob.j + ob.k;
    cout << sum << endl;
}
```

Virtual base classes

- Duplication of inherited members due to these multiple paths can be avoided
- When a class is made a virtual base class, only one copy of the class is inherited

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

class base{
    public: int i;
};
```

```
class derived1 : virtual public base{
    public:int j;
};

class derived2 : virtual public base{
    public:int k;
};

class derived3 : public derived1, public derived2{
    public: int sum;
};

int main(){
    derived3 ob;
    ob.i = 50; ob.j = 20; ob.k = 30;
    int sum = ob.i + ob.j + ob.k;
    cout << sum << endl;
    return 0;
}
```

Virtual Functions (Polymorphism)

- `virtual return_type function`
- Declared in a base class and can be overridden by derived classes.
- Virtual Functions are Hierarchical : If not overridden in a derived class, the base class's function is used

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

class Base {
public:
    virtual void display() {
        cout << "Display Base class" << endl;
    }
};

class Derived : public Base {
public:
    void display() override { // Overriding the base class method
        cout << "Display Derived class" << endl;
    }
};
```

```
int main() {
    Base *ptr;          // Base class pointer
    Base base;
    Derived obj;

    ptr = &base;        // Pointing to Base class object
    ptr->display();     // Display Base class

    ptr = &obj;         // Pointing to Derived class object
    ptr->display();     // Display Derived class

    return 0;
}
```

Pure Virtual Functions(Abstract Class)

- `virtual type func_name (parameter_list) = 0`
- A pure virtual function is a virtual function that has no definition in the base class
- Derived class must provide its own definition of the virtual function
- We cannot create an object or cannot be instantiated for an abstract class
- We can create a pointer variable for abstract class which can refer to any of its derived class

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

class Shape{
public:
    virtual void draw() = 0;
    virtual ~Shape(){}
};

class Circle : public Shape{
public:
    void draw() override{
        cout << "circle" << endl;
    }
};

class Rectangle : public Shape{
public:
    void draw() override{
        cout << "rectangle" << endl;
    }
```

```
};

int main(){
    Shape *c = new Circle();
    Shape *r = new Rectangle();

    c->draw();  // circle
    r->draw();  // rectangle

    delete c;
    delete r;
    return 0;
}
```

Virtual Function Mechanics – The Virtual Table

- Late binding : The table contains pointers to virtual functions of a class
- The compiler places the addresses of the virtual functions for that particular class in the VTABLE

Virtual Destructor

- call the inherited class destructor as well, thus properly disposing the class instances
- Ensure proper cleanup of derived class objects when deleting a pointer to a base class

Friend functions

- Can access private and protected members of a class.
- The function can be invoked without the use of an object
- The friend function can have its argument as objects

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

class Box{
private:
    double width;
public:
    Box() : width(0) {}

    friend void setW(Box &obj, double w); // friend function declaration
```

```
    void display(){
        cout << "width of box : " << width << endl;
    }
};

void setW(Box &obj, double w){
    obj.width = w;
}

int main(){
    Box obj;
    setW(obj,3.4);
    obj.display();
    return 0;
}
```

Friend Classes

- One class can be a friend of another, gaining access to its private members

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

class B; // forward declaration

class A{
private:
    int data;
public:
    A() : data(0) {}
    friend class B;
};

class B{
public:
    void setData(A &obj, int value){
        obj.data = value;
        cout << "Data in class A : " << obj.data << endl;
    }
};

int main(){
    A aobj;
    B bobj;
    bobj.setData(aobj,100);  // Data in class A : 100
```

```
    return 0;
}
```

Namespaces

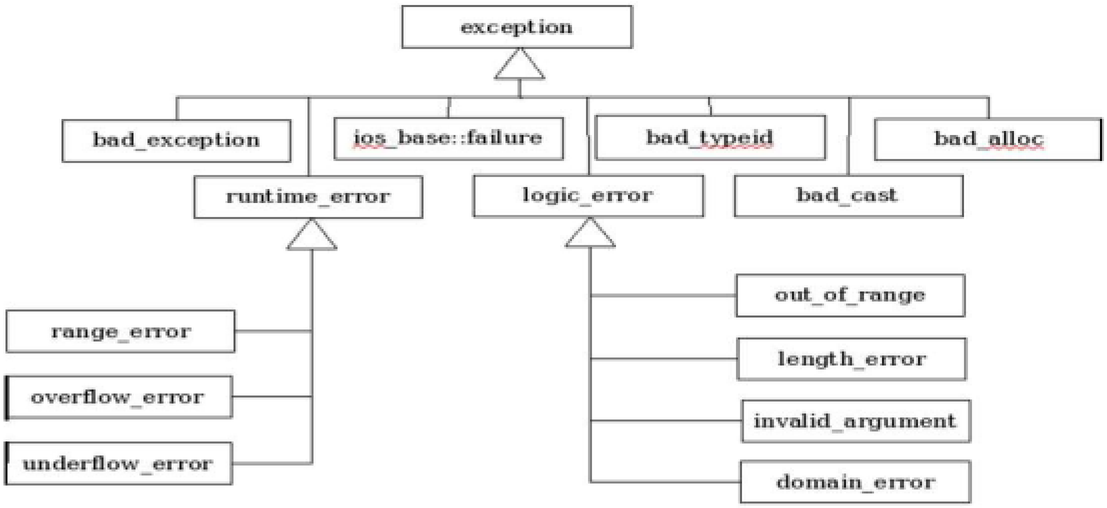
- Group classes, objects, and functions under a single name to avoid name conflicts
- `namespace identifier { entities }`

Operator Overloading

- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it
- These below operators can be overloaded
  - new delete
  - +-*/%^&|~
  - ! = < > += -= *= /= %=
  - ^= &= |= << >> >>= <<= == !=
  - <= >= && || ++ -- , ->* ->
  - () []
- Operator that are not overloaded are
  - scope operator - ::
  - sizeof
  - Period - .
  - ternary operator - ?:
  - typeid() operator

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

class FLOAT {
    float no;
public:
    FLOAT() : no(0.0) {}  // Default constructor initializing no to 0.0
    FLOAT(float n) : no(n) {}  // Parameterized constructor

    void getdata() {
        cout << "Enter a float number : ";
        cin >> no;
    }
```

```
    void putdata() const {
        cout << "Result : " << no << endl;
    }

    FLOAT operator+(FLOAT f) {
        return FLOAT(no + f.no);
    }

    FLOAT operator*(FLOAT f) {
        return FLOAT(no * f.no);
    }

    FLOAT operator-(FLOAT f) {
        return FLOAT(no - f.no);
    }

    FLOAT operator/(FLOAT f) {
        if (f.no != 0) {
            return FLOAT(no / f.no);
        } else {
            cout << "\nDivision by zero error!";
            return FLOAT(0);  // Return zero if division by zero
        }
    }
};

int main() {
    FLOAT f1, f2, result;

    f1.getdata();
    f2.getdata();

    result = f1 + f2;
    cout << "\nAddition";
    result.putdata();

    result = f1 - f2;
    cout << "\nSubtraction";
    result.putdata();

    result = f1 * f2;
    cout << "\nMultiplication";
    result.putdata();

    result = f1 / f2;
    cout << "\nDivision";
    result.putdata();

    return 0;
}
```



try – throw - catch block

- **try block** : This block contains the code that may generate an exception
- **throw statement** : This is used to signal the occurrence of an exception
- **catch block** : This block handles the exception. It's declared with an exception parameter

```
try {
    // Code that may throw an exception
    if (condition) {
        throw exception; // Throw an exception
    }
}
catch (type exception) {
    // Code to handle the exception
}
```

```
try {
    // Code that may throw an exception
}
catch (int e) {
    // Handle integer exceptions
}
catch (const char* e) {
    // Handle string exceptions
}
catch (...) {
    // Handle any type of exception - ellipsis
}
```

Exception Handling - eg

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
using namespace std;

// Custom exception class
class MyException : public exception {
public:
    const char* what(){
        return "My custom exception";
    }
};

int main() {
        int num1,num2;
    cout << "enter two numbers : ";
    cin >> num1 >> num2;

        string str;
        cout << "enter a string : ";
        cin >> str;

        int arr[5] = {1, 2, 3, 4, 5};
    int index = 10;

    bool customError = true;

    try {
        // Integer exception
        if (num2 == 0) {
            throw runtime_error("Division by zero");
        }
        cout << "Division result: " << num1 / num2 << endl;

        // String exception
        if (str.length() > 5) {
            throw string("String too long");
        }
        cout << "String: " << str << endl;

        // Array out-of-bounds exception
        if (index >= 5) {
            throw out_of_range("Array index out of bounds");
        }
        cout << "Array element: " << arr[index] << endl;
```

```
        // Custom exception
        if (customError) {
            throw MyException();
        }
    }
    catch (const runtime_error& e) {
        cout << "Runtime error : " << e.what() << endl;
    }
    catch (const string& e) {
        cout << "String exception : " << e << endl;
    }
    catch (const out_of_range& e) {
        cout << "Out_of_range exception : " << e.what() << endl;
    }
    catch (const MyException& e) {
        cout << "Custom exception: " << e.what() << endl;
    }
    catch (const exception& e) {
        cout << "Caught an exception"<< endl;
    }

    return 0;
}
```

Templates in C++

- Templates allow writing generic and reusable code
- They enable functions and classes to operate with generic types, reducing code duplication
- Fall under the category of "meta-programming" and auto code generation

Function Templates

- Function templates define a pattern for functions that can operate on different data types
- The compiler generates the specific function code based on the types used

```
template <typename T>
T maximum(T a, T b, T c) {
    T max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
```

```
}

int main() {
    int i1 = 1, i2 = 2, i3 = 3;
    double d1 = 1.1, d2 = 2.2, d3 = 3.3;

    cout << "Max int: " << maximum(i1, i2, i3) << endl;
    cout << "Max double: " << maximum(d1, d2, d3) << endl;

    return 0;
}
```

Class Templates

- Class templates define a blueprint for a class that can handle different data types, allowing for the creation of type-specific instances

```
template <typename T1, typename T2>
class Pair {
public:
    T1 first;
    T2 second;
    Pair(T1 f, T2 s) : first(f), second(s) {}
};

int main() {
    Pair<int, double> p1(1, 2.2);
    Pair<string, string> p2("Hello", "World");

    cout << "Pair 1: " << p1.first << ", " << p1.second << endl;
    cout << "Pair 2: " << p2.first << ", " << p2.second << endl;

    return 0;
}
```

Template with Multiple Generic Types

- This allows a function or a class to accept parameters of different types while still maintaining type safety and flexibility

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

template<class T, class U, class V>
void tempfun(T a, U b, V c) {
```

```
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}

int main() {
    int i = 10;
    float j = 3.14f;
    char k = 'T';
    tempfun(i, j, k);  // Calls tempfun with int, float, and char
    return 0;
}
```

Template Function Overloading

- Define several template functions with the same name but with different type parameters or different numbers of parameters

```
template <typename T>
T add(T a, T b) {      // Template function to add two integers
        return a + b;
}

template <typename T>
T add(T a, T b, T c) { // Template function to add three integers
        return a + b + c;
}

template <>
string add<string>(string a, string b) { // Template function to add two
strings
        return a + " " + b;
}
```

Explicitly Overloading a Generic Function

- If you overload a generic function, that overloaded function overrides (or hides) the generic function relative to that specific version

---

Standard Template Library

- STL is a library in C++ that provides a set of common classes and interfaces for DSA

1 Containers

- Data structures like arrays, lists, and queues
- Functions shared by all containers
  - Default constructor, copy constructor, destructor
  - = < <= > >= == !=
  - size queries (`size()`, `empty()`, etc.)

1. **Sequence Containers**
   - Store data in a linear sequence
   - eg : `vector`, `deque`, `list`
2. **Associative Containers**
   - Store data in key-value pairs for quick access
   - eg : `set`, `map`, `multimap`
3. **Container Adapters**
   - Provide different interface for existing sequence containers
   - eg : `stack`, `queue`, `priority_queue`

2 Algorithms

- Functions for data manipulation like searching and sorting

3 Iterators

- Objects that allow traversing elements in a container, similar to pointers
- `*` : Dereferences the iterator to access the element
- `++` : Moves the iterator to the next element
- `begin()` : Returns an iterator to the first element
- `end()` : Returns an iterator to the past-the-end element

Sequence Containers : Vector Container

- Have to include the following header file `<vector>`
- Data structure with contiguous memory locations
- `push_back(value)` : Adds an element to the end
- `size()` : Returns the number of elements
- `capacity()` : Returns the size of the allocated storage

- `insert(iterator, value)` : Inserts an element before the specified position
- `erase(iterator)` : Removes the element at the specified position
- `clear()` : Removes all elements
- `begin()`, `end()` : Returns iterators to the beginning and end of the vector

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

int main(){
    vector<int>v;

    v.push_back(100);
    v.push_back(1000);
    v.push_back(10000);
    v.insert(v.begin(), 10);
    v.insert(v.end(),100000);
    v.erase(v.begin() + 3);

    for(int i : v){
        cout << i << endl;  // 10 100 1000 100000
    }
    return 0;
}
```

Associative Container : Map

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

int main() {
    map<int, string> myMap;
    myMap[1] = "One";
    myMap[2] = "Two";
    myMap[3] = "Three";
    for(const auto &pair : myMap) {
        cout << pair.first << ": " << pair.second << endl;
    }

        /*for (const auto &[key, value] : myMap) {
                cout << key << ": " << value << endl;
        }*/

    // 1: One
    // 2: Two
```

```
    // 3: Three
    return 0;
}
```

Container Adapter : Stack

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

int main(){
    stack<int>s;
    for(int i=0;i<5;i++){
        s.push(i);
    }

    while(!s.empty()){
        cout << " " << s.top(); // 4 3 2 1 0
        s.pop();
    }
    return 0;
}
```

Algorithm

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

int main(){
    vector<int>v(5);
    bool found;

    for(int i=0;i<5;i++){
        v[i]=i; // v.push_back(i)
    }

    found = binary_search(v.begin(), v.end(), 3);
    cout << found << endl;

    found = binary_search(v.begin(), v.end(), 9);
    cout << found << endl;

    sort(v.begin(), v.end());
```

```
    for(int i : v){
        cout << i << "->";
    }
    return 0;
}
```

Iterator

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

int main(){
    vector<int>v;

    v.push_back(100);
    v.push_back(1000);
    v.push_back(10000);
    v.insert(v.begin(), 10);
    v.insert(v.end(),100000);
    v.erase(v.begin() + 3);

    for(auto i = v.begin(); i != v.end(); i++) {
        cout << *i << " ";  // 10 100 1000 100000
    }
    return 0;
}
```