**Pointer VS References**

Pointer: The pointer holds the address space of another variable. This is not fixed, the pointer can point to any other variable.

Reference : The reference variable is like another name for the variable.

Program:

#include<iostream>

Using namespace std;

Int main(){

Int n=10;

Int \*ptr=&n; //here ptr hold the address value of n

Cout<<\*ptr<< endl; // the output will be 10

Int &p=n; //here p holds the address space of n and is constant pointer to that address

Cout<<p<<endl // the output will be 10 here

return 0;

}

**Stack Vs Heap**

Stack:

* Stack allocation refers to the process of assigning memory for local variables and function calls in the call stack.
* It happens automatically when a function is called and is freed immediately when the function ends.
* Since memory is managed by the system, it is fast and efficient, but has limited space compared to heap allocation.
* If too many function calls exceed the stack's capacity, it results in a stack overflow error.

How Stack Allocation Works

* Memory is allocated in contiguous blocks within the call stack.
* The size of memory required is already known before execution.
* When a function is called, its local variables are allocated on the stack.
* Once the function finishes execution, the allocated memory is automatically freed.
* The programmer does not need to handle allocation or deallocation.
* Since stack memory is freed when a function completes, it is also called temporary memory allocation.

Program:

Void func(){

Int x =10;

}

Heap:

Heap Allocation

Heap memory is allocated dynamically during program execution. Unlike stack memory, heap memory is not freed automatically when a function ends. Instead, it requires manual deallocation (In C/C++) or a garbage collector (in Java or Python) to reclaim unused memory.

The name heap has no relation to the heap data structure; it simply refers to a large pool of memory available for dynamic allocation. Whenever an object is created, it is stored in heap memory, while references to these objects are stored in stack memory. Heap allocation is less safe than stack allocation because heap data is accessible by multiple threads, increasing the risk of data corruption and memory leaks if not handled properly.

Categories of Heap Memory

Heap memory is divided into three categories, helping prioritize object storage and garbage collection:

Young Generation – Stores new objects. When this space is full, unused objects are removed by the garbage collector, and surviving objects move to the Old Generation.

Old (Tenured) Generation – Holds older objects that are no longer frequently used. These objects stay in memory unless explicitly removed.

Permanent Generation (PermGen) – Stores JVM metadata, such as runtime classes and application methods.

Key Features of Heap Allocation

* If heap memory is full, JVM throws an error: java.lang.OutOfMemoryError.
* Unlike stack memory, automatic deallocation does not happen,a garbage collector is needed to free unused memory.
* Slower than stack memory due to manual allocation and garbage collection.
* Less thread-safe, as heap memory is shared among all threads.
* Typically larger in size compared to stack memory.
* Heap memory persists as long as the entire application is running.

**How Stack & Heap coexist**

When your program runs, memory is divided into different regions:

* **Stack segment** → Stores function call frames (local variables, parameters, return addresses).
* **Heap segment** → Stores dynamically allocated memory (anything you create with new, malloc, calloc in C/C++).
* **Data segment** → Stores global and static variables.
* **Code segment** → Stores your compiled program instructions.

**Key rule:**

* Variables created normally (e.g., int x = 10;) → stored on **stack**.
* Variables created with new → stored on **heap**, but **the pointer to them** lives on the stack (or in a global variable).

#include <iostream>

using namespace std;

void functionB(int b) {

int localB = b + 10; // Local variable on stack

int\* heapB = new int(500); // Dynamic allocation on heap

cout << "\n--- Inside functionB ---" << endl;

cout << "Address of localB (stack): " << &localB << " | Value: " << localB << endl;

cout << "heapB pointer (stack): " << &heapB << " | Points to heap: " << heapB

<< " | Value in heap: " << \*heapB << endl;

// We intentionally DON'T delete heapB here to show leak risk

}

void functionA(int a) {

int localA = a \* 2;

cout << "\n--- Inside functionA ---" << endl;

cout << "Address of localA (stack): " << &localA << " | Value: " << localA << endl;

functionB(localA);

}

int main() {

int mainVar = 42;

int\* heapMain = new int(999);

cout << "--- Inside main ---" << endl;

cout << "Address of mainVar (stack): " << &mainVar << " | Value: " << mainVar << endl;

cout << "heapMain pointer (stack): " << &heapMain << " | Points to heap: " << heapMain

<< " | Value in heap: " << \*heapMain << endl;

functionA(mainVar);

delete heapMain; // Clean up

return 0;

}

OOPS Concepts:

Encapsulation: Wrappig up of data. It mainly includes how data Is accesed and modified.

Program:

class SavingsAccount {

private:

double balance;

public:

SavingsAccount(double initialBalance) : balance(initialBalance) {}

void deposit(double amount) {

balance += amount;

cout << "Deposited: " << amount << " | New Balance: " << balance << endl;

}

void withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

cout << "Withdrawn: " << amount << " | New Balance: " << balance << endl;

} else {

cout << "Insufficient funds!" << endl;

}

}

double getBalance() {

return balance;

}

};

}

// Main Program

int main() {

using namespace BankSystem; // Make namespace content accessible

Account\* myAccount = new SavingsAccount(500.0); // Only see the interface, not internals

myAccount->deposit(200.0); // User knows WHAT to do, not HOW it works

myAccount->withdraw(100.0);

cout << "Final Balance: " << myAccount->getBalance() << endl;

delete myAccount; // Clean up

return 0;

}

Abstarction : Hiding of implementation . Basically hiding internal features and only showing outside world essential features.

Abstraction is achieved by abstract classes:

Abstract class is a class having a pure virtual function[ A pure virtual function is a function with no implementation In the base classes] and the child class must provide with the implementation.

Program:

#include <iostream>

using namespace std;

// Namespace to group our Bank related code

namespace BankSystem {

// Abstract class (Interface for account actions)

class Account {

public:

virtual void deposit(double amount) = 0; // Pure virtual function

virtual void withdraw(double amount) = 0;

virtual double getBalance() const = 0;

virtual ~Account() {} // Virtual destructor

};

// Implementation of Account

class SavingsAccount : public Account {

private:

double balance;

public:

SavingsAccount(double initialBalance) : balance(initialBalance) {}

void deposit(double amount) override {

balance += amount;

cout << "Deposited: " << amount << " | New Balance: " << balance << endl;

}

void withdraw(double amount) override {

if (amount <= balance) {

balance -= amount;

cout << "Withdrawn: " << amount << " | New Balance: " << balance << endl;

} else {

cout << "Insufficient funds!" << endl;

}

}

double getBalance() const override {

return balance;

}

};

}

// Main Program

int main() {

using namespace BankSystem; // Make namespace content accessible

Account\* myAccount = new SavingsAccount(500.0); // Only see the interface, not internals

myAccount->deposit(200.0); // User knows WHAT to do, not HOW it works

myAccount->withdraw(100.0);

cout << "Final Balance: " << myAccount->getBalance() << endl;

delete myAccount; // Clean up

return 0;

}

Polymorphism

Polymorphism = “many forms.”

Write code that uses a single interface while allowing different behaviors depending on the concrete type.

1) Compile-time (static) polymorphism

Resolved at compile time. No runtime cost.

* Function overloading: same function name, different parameter lists.
* Operator overloading: define +, <<, etc. for your types.
* Templates: functions/classes that work for many types (parametric polymorphism).
* CRTP (advanced): pattern for static “override”-like behavior with templates.

Run-time Polymorphism (Dynamic Polymorphism)

Definition

Run-time polymorphism means that the method that gets executed is determined at run time, not at compile time.

In C++, this is mainly achieved through:

* Inheritance (base class and derived class)
* Virtual functions
* Base class pointers or references

Program for compile time :

#include <iostream>

#include <vector>

namespace Poly::CompileTime {

// Overloading

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

// Template (parametric polymorphism)

template <typename T>

T sum(const std::vector<T>& v) {

T s{};

for (const auto& x : v) s += x;

return s;

}

// Operator overloading

struct Vec2 {

float x{}, y{};

Vec2(float x, float y): x(x), y(y) {}

};

Vec2 operator+(const Vec2& a, const Vec2& b) {

return Vec2(a.x + b.x, a.y + b.y);

}

}

Program Run Time :

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

virtual void makeSound() { // Virtual function

cout << "Some generic animal sound" << endl;

}

virtual ~Animal() {} // Virtual destructor for safe cleanup

};

// Derived class Dog

class Dog : public Animal {

public:

void makeSound() override { // Override base function

cout << "Woof! Woof!" << endl;

}

};

// Derived class Cat

class Cat : public Animal {

public:

void makeSound() override {

cout << "Meow! Meow!" << endl;

}

};

int main() {

Animal\* animalPtr; // Base class pointer

Dog d;

Cat c;

// Pointing base pointer to Dog object

animalPtr = &d;

animalPtr->makeSound(); // OUTPUT: Woof! Woof! (Dog's version)

// Pointing base pointer to Cat object

animalPtr = &c;

animalPtr->makeSound(); // OUTPUT: Meow! Meow! (Cat's version)

return 0;

}