There are mainly two types of polymorphism:

* Compile Time Polymorphism in C++: Compile time polymorphism in C++ is invoked during the compile time of the program. There are two ways to achieve compile-time polymorphism in C++:
  + Function Overloading
  + Operator Overloading
* Run Time Polymorphism in C++: Run time polymorphism in C++ is invoked during the run time of the program. There is one way to achieve the run time polymorphism in C++:
  + Function Overriding

## **What is Compile Time Polymorphism in C++?**

The compile time polymorphism in C++ is a type of polymorphism, which refers to the ability of a programming language to determine the appropriate method or function to call at compile time-based on the types of arguments being passed.

There are a couple of ways to achieve compile time polymorphism in C++.

### **1. Function Overloading:**

The C++ language’s function overloading feature enables us to define multiple functions that have the same name but different parameters. When we want to perform the same operation on various data types, or when we want to offer different levels of functionality depending on the quantity or kind of arguments passed, this can be helpful.

In C++, we simply define two or more functions with the same name but different parameters to overload a function. The compiler chooses which version of the function to call based on the number, type, and ordering of the arguments passed.

Let’s take an example to understand the compile time polymorphism in C++ using the function overloading.

* [C++](https://www.prepbytes.com/blog/cpp-programming/compile-time-polymorphism-in-cpp/#tabs_desc_14600_1)

// compile time polymorphism in C++

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

class PrepBytes {

public:

// Function with 1 int parameter

void add(int x,int y)

{

std::cout << "The Sum of "<< x <<" and "<<y <<" is: " << x+y<<"\n";

}

// Function with same name but

// 1 double parameter

void add(int x, int y, int z)

{

std::cout << "The Sum of "<< x <<", "<< y <<" and "<<z <<" is: " << x+y+z << "\n";

}

// Function with same name and

// 2 int parameters

void add(double x, double y)

{

std::cout << "The Sum of "<< x <<" and "<<y <<" is: " << x+y<< "\n";

}

void add(double x, double y, double z)

{

std::cout << "The Sum of "<< x << ", " << y <<" and "<< z <<" is: " << x+y+z<< "\n";

}

};

int main()

{

PrepBytes obj;

obj.add(7,8);

obj.add(10,14,16);

obj.add(4.5, 6.2);

obj.add(1.3, 4.6, 7.2);

return 0;

}

**Output:**

****The Sum of 7 and 8 is: 15

The Sum of 10, 14 and 16 is: 40

The Sum of 4.5 and 6.2 is: 10.7

The Sum of 1.3, 4.6 and 7.2 is: 13.1

In the above C++ program, the class Prepbytes contains several methods with the same name add. During compile time, it will be decided which method to call based on the given arguments in the function parameter. For example, if we give 3 int data type arguments in the add function then it will call the second add function.

### **2. Operator Overloading:**

In C++, a feature called operator overloading enables operators like +, -, \*, /, and others to be redefined for user-defined data types. As a result, we can specify what the operator does when used with objects belonging to our own class, which enables us to write more logical and expressive code.

In C++, we define a function that starts with the operator keyword and the symbol we want to overload. For instance, we might define a function with the following signature to overload the addition operator + for the class "MyClass":

MyClass operator+(const MyClass& obj) const;

The ampersand (&) indicates that the function takes its argument by reference rather than by value and the keyword "const" indicates that it does not modify the object it is called on.

Let’s take an example to understand the compile time polymorphism in C++ using operator overloading.

// compile time ploymorphism in C++

#include <iostream>

using namespace std;

class Complex {

public:

Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

Complex operator+(const Complex& obj) const {

return Complex(real + obj.real, imag + obj.imag);

}

void display() const {

cout << real << " + " << imag << "i" << endl;

}

private:

double real, imag;

};

int main() {

Complex c1(1.0, 2.0), c2(2.0, 3.0);

Complex c3 = c1 + c2; // calls the overloaded + operator

c3.display();

return 0;

}

**Output:**

****3 + 5i

In this example, we define a class called "Complex" that represents complex numbers with a real and imaginary components. We overload the addition operator (+) by defining a method called "operator+" that takes a Complex object as a reference and returns a new Complex object representing the sum of the two Complex objects.