C++ Cheatsheet - A Beginners Guide

This is a simple C++ cheat sheet for beginners. It is a quick reference guide to the C++ language. It is not a tutorial, but it is a handy reference to the C++ syntax.

Table of Contents

Introduction to C++

C++ is a general-purpose programming language that was developed by Bjarne Stroustrup in 1979. It is an extension of the C programming language and is still one of the fastest and most efficient programming languages available today.

It is used by companies like Adobe, Microsoft, Google, NASA, and many more. Every 3 years, a new version of C++ is released, which adds new features to the language.

Applications of C++

C++ is one of the most popular programming languages in the world. It is used in a variety of applications, including:

- High-Performance Applications
- Video Games
- Device Drivers
- Web Browsers
- Servers
- Operating Systems

Mastering C++

To master C++, we need to learn 2 things:

1. The syntax and semantics of the language itself
2. C++ Standard Library (STL) - A collection of classes and functions that are part of the C++ Standard Library for example, algorithms, containers, iterators, etc.

Writing C++ Programs

Let's cover how and where to write C++ programs.

Popular IDEs for C++

There are many IDEs available for C++, some of them are free and some are paid. Some of the popular ones are:

- Visual Studio
- Xcode
- CLion

First C++ Program

- A C++ program is a collection of functions, and every C++ program must have a `main` function
- The `main` function is the entry point of the program, and it is called when the program is executed
- The `main` function returns an integer value, which is used to indicate the status of the program

> Note: C++ is case-sensitive, so `main` and `Main` are different functions.

```
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- The `#include <iostream>` directive tells the compiler to include the iostream library, which provides input and output functionality
- `cout` stands for "character output"

Compiling and Running C++ Programs

C++ is a compiled language, which means to run a C++ program, we need to compile it first. The compilation process translates the source code into machine code that can be executed by the computer.

Installation and Setup

1. Open Visual Studio Code from the Developer Command Prompt for Visual Studio.
2. Install C/C++ extension pack for Visual Studio Code.
3. Install MinGW via MSYS2 page or use this direct installer link.
4. Run the installer and install MinGW.
5. Once installed, open MSYS2 shell and run the following command -

   ```
   pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
   ```

6. Select the default installation options and proceed with the installation.
7. Add the MinGW bin directory to the system path environment variable.

   ```
   C:\Windows\msys64\ucrt64\bin
   ```

8. Check MinGW installation by running the following command in a terminal -

   ```
   gcc --version
   g++ --version
   gdb --version
   ```

9. Create a new C++ file and add some code to test the installation.

   ```
   #include <iostream>

   int main() {
     std::cout << "Hello, World!" << std::endl;
     return 0;
   }
   ```

10. Run the file using by clicking the "Run" button in Visual Studio Code or by running the following command in the terminal -

    ###### Compile the file

    ```
    g++ -g -o main main.cpp
    ```

    ###### Run the file

    ```
    # run the file
    ./main
    ```

Comments

We write comments in C++ using `//` for single-line comments and `/* ... */` for multi-line comments.

```
// This is a single-line comment

/*
  This is a multi-line comment
*/
```

Variables

Variables are used to store data in a program. We can declare a variable using the following syntax -

```
int age = 25;
```

In the above code -

- `int` is the data type of the variable.
- `age` is the name of the variable.
- `25` is the value assigned to the variable.

We can also declare a variable without assigning a value to it.

```
int age;

// assign a value later
age = 25;
```

We can also modify the value of a variable later in the program.

```
int age = 25;

// modify the value
age = 30;
```

Constants

A constant is a value that cannot be altered by the program during normal execution. It is a type of variable that is used to store a value that does not change.

```
const pi = 3.14;
```

If we try to modify the value of a constant variable, the compiler will throw an error.

```
const pi = 3.14;

// error: assignment of read-only variable 'pi'
pi = 3.14159;
```

Naming Conventions

- Always use meaningful names for variables
- Variables names can follow any case convention (camelCase, snake_case, etc.)
- It depends on the organization's coding standards

The Standard Library

The C++ Standard Library is a collection of classes and functions that are part of the C++ Standard Library. It is a powerful tool that can be used to perform a variety of tasks.

Writing to the Console

The `iostream` library provides the `cout` object, which is used to print text to the console.

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

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

- The `<<` operator is used to insert data into the `cout` object
- The `endl` object is used to insert a newline character into the `cout` object
- The `using namespace std` directive tells the compiler to use the `std` namespace, which contains the `cout` object

Reading from the Console

The `iostream` library also provides the `cin` object, which is used to read input from the console.

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

int main() {
  int age;
  cout << "Enter your age: ";
  cin >> age;
  cout << "Your age is: " << age << endl;
  return 0;
}
```

- The `>>` operator is used to extract data from the `cin` object
- The `int age` statement declares a variable called `age` of type `int`
- The `cin >> age` statement reads an integer value from the console and stores it in the `age` variable
- The `cout << "Your age is: " << age << endl` statement prints the value of the `age` variable to the console

Mathematical Functions

The `cmath` library provides a variety of mathematical functions, such as `sqrt`, `pow`, `abs`, `sin`, `cos`, `tan`, etc.

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

int main() {
  // Square Root
  cout << "Square Root: " << sqrt(16) << endl; // 4

  // Power
  cout << "Power: " << pow(2, 3) << endl; // 2^3 = 8

  // Floor
  cout << "Floor: " << floor(2.6) << endl; // 2

  // Ceiling
  cout << "Ceiling: " << ceil(2.3) << endl; // 3

  // Absolute
  cout << "Absolute: " << abs(-10) << endl; // 10

  return 0;
}
```

Data Types

C++ is a statically typed language, which means that the type of a variable cannot be changed once it has been declared. Other languages, such as Python and JavaScript, are dynamically typed, which means that the type of a variable can change during the execution of the program.

Fundamental Data Types

C++ provides a variety of data types that can be used to store different types of data. The following table lists the fundamental data types in C++.

Type	Bytes	Range
int	4	-2,147,483,648 to 2,147,483,647
char	1	-128 to 127 or 0 to 255
bool	1	true or false
float	4	3.4e-38 to 3.4e+38 (6 decimal places)
double	8	1.7e-308 to 1.7e+308 (15 decimal places)
long	4	-2,147,483,648 to 2,147,483,647
short	2	-32,768 to 32,767

Int

- `int` is used to store integer values
- Has a size of 4 bytes and a range of -2,147,483,648 to 2,147,483,647
- Can use the `unsigned` keyword to store only positive values

```
int age = 25;
```

Int vs Long

The `int` and `long` data types are used to store integer values. The `long` data type can store larger values than the `int` data type.

```
int a = 2147483647;
long b = 2147483648L;
```

- The `L` suffix is used to indicate that the value is a `long` literal
- If the `L` suffix is not used, the value is treated as an `int` literal

Float

- `float` is used to store decimal numbers
- Has a size of 4 bytes and a range of 3.4e-38 to 3.4e+38
- `f` suffix can be used to declare a float value

```
float pi = 3.14f;
```

Double

- `double` is used to store double-precision floating-point numbers
- Cam store double the range of float and hence more accurate
- Has a size of 8 bytes and a range of 1.7e-308 to 1.7e+308
- `d` suffix can be used to declare a double value

```
double pi = 3.14159265359;
```

Double vs Float

The `double` data type is a double-precision floating-point number, which means that it can store more decimal places than the `float` data type.

```
float a = 3.14159F;
double b = 3.14159;
```

- The `F` suffix is used to indicate that the value is a `float` literal
- If the `F` suffix is not used, the value is treated as a `double` literal which can lead to a loss of precision
- The `double` data type is the default floating-point type in C++

Char

- `char` is used to store single characters
- Has a size of 1 byte and a range of -128 to 127

```
char grade = 'A';
```

String

- `string` is used to store a sequence of characters

```
string name = "John Doe";
```

Bool

- `bool` is used to store boolean values
- Has a size of 1 byte and can store either `true` or `false`

```
bool isStudent = true;
```

Array

- `array` is used to store a collection of elements of the same data type
- Can be declared using square brackets `[]`
- Type of array needs to be specified while declaring
- Cannot store elements of different data types
- Size of array can also be specified while declaring

```
int numbers[] = {1, 2, 3, 4, 5};

// specify size
int numbers[5] = {1, 2, 3, 4, 5};

// access elements
cout << numbers[0] << endl; // 1

// modify elements
numbers[0] = 10;
cout << numbers[0] << endl; // 10
```

These were some of the basic data types in C++. There are more data types such as `long`, `long long`, `short`, `unsigned int`, `long double`, etc. which are used in specific scenarios. Please refer to the official documentation for more details.

`Auto` Keyword

The `auto` keyword is used to automatically deduce the data type of a variable at compile time.

```
auto a = 10; // int
auto b = 3.14; // double
auto c = 3.14F; // float
```

- The `auto` keyword is useful when the data type of a variable is complex or difficult to determine
- The `auto` keyword is not used to declare function parameters or return types
- The `auto` keyword is not used to declare class members or global variables

Narrowing Conversion

Narrowing conversion is the process of converting a value of a larger data type to a value of a smaller data type. This can lead to a loss of precision.

```
int a = 1'000'000;
short b = a; // Narrowing Conversion

cout << "a: " << a << endl; // 1000000
cout << "b: " << b << endl; // 16960
```

- The value of `a` is too large to be stored in a `short` variable, so the value is truncated
- The value of `b` is 16960, which is the remainder of 1000000 divided by 65536 (the maximum value of a `short` variable)
- The `short` data type is 2 bytes, which can store values from -32,768 to 32,767

Mathematical Operators

C++ provides a variety of operators that can be used to perform mathematical operations.

Common Arithmetic Operators

```
 double a = 10;
int b = 20;

// Addition
double sum = a + b;
cout << "Sum: " << sum << endl; // 30

// Subtraction
double difference = a - b;
cout << "Difference: " << difference << endl; // -10

// Multiplication
double product = a * b;
cout << "Product: " << product << endl; // 200

// Division
double quotient = a / b;
cout << "Quotient: " << quotient << endl; // 0.5

// Modulus
int remainder = b % a;
cout << "Remainder: " << remainder << endl; // 0

// Increment
a++;
cout << "Increment: " << a << endl; // 11

// Decrement
b = b - 1;
cout << "Decrement: " << b << endl; // 19
```

Order of Operations

The order of operations in C++ is the same as in mathematics. The following table lists the precedence and associativity of C++ operators.

Precedence	Operator	Description
1	()	Parentheses
2	* / %	Multiplication, Division, Modulus
3	+ -	Addition, Subtraction

This can be referred to as BODMAS (Brackets, Orders, Division and Multiplication, Addition and Subtraction).

Code Example

```
 int result = 10 + 20 * 30;
cout << "Result: " << result << endl; // 610
```

Generating Random Numbers

The `cstdlib` library provides the `rand` function, which is used to generate random numbers.

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

int main() {
  // Seed
  srand(time(0));

  // Random Number
  int random = rand() % 100;
  cout << "Random Number: " << random << endl;

  return 0;
}
```

- The `srand` function is used to seed the random number generator with a value
- The `time(0)` function is used to generate a seed based on the current time
- The `rand` function is used to generate a random number but it is not truly random because it is based on a mathematical formula
- The `%` operator is used to generate a random number between 0 and 99

Conditionals

Conditionals are used to make decisions in a program. We can use `if`, `else if`, and `else` statements to execute different blocks of code based on different conditions.

If-Else Statement

```
int age = 25;

if (age >= 18) {
  cout << "You are an adult" << endl;
} else {
  cout << "You are a minor" << endl;
}
```

Ternary Operator

We can also use the ternary operator to write a more concise version of the above code.

```
int age = 25;

string message = (age >= 18) ? "You are an adult" : "You are a minor";
cout << message << endl;
```

Nested If-Else Statement

We can also use nested if-else statements to check for multiple conditions.

```
int age = 25;
bool hasLicense = true;

if (age >= 18) {
  if (hasLicense) {
    cout << "You can drive" << endl;
  } else {
    cout << "You cannot drive" << endl;
  }
} else {
  cout << "You are a minor" << endl;
}
```

Switch Statement

We can use the switch statement to execute different blocks of code based on different values of a variable.

```
 enum Weather {
   Sunny,
   Rainy,
   Cold
};

Weather today = Sunny;

switch (today) {
  case Sunny:
    cout << "It's sunny today" << endl;
    break;
  case Rainy:
    cout << "It's rainy today" << endl;
    break;
  case Cold:
    cout << "It's cold today" << endl;
    break;
  default:
    cout << "Invalid weather" << endl;
}
```

Enums

Enums are used to define a collection of constants. They are used to define a set of named integer constants.

```
 enum Weather {
   Sunny,
   Rainy,
   Cold
};
```

- `enum` is the keyword used to define an enumeration.
- `Weather` is the name of the enumeration.
- `Sunny`, `Rainy`, and `Cold` are the named constants.
- By default, the value of the first constant is 0, and the value of each subsequent constant is incremented by 1.
- We can also assign custom values to the constants.

```
 enum Weather {
   Sunny = 1,
   Rainy = 2,
   Cold = 3
};
```

Loops

Loops are used to execute a block of code multiple times. We can use `for`, `while`, and `do-while` loops in C++.

For Loop

```
 for (int i = 0; i < 5; i++) {
  cout << i << endl;
}
```

While Loop

```
 int i = 0;

while (i < 5) {
  cout << i << endl;
  i++;
}
```

Do-While Loop

```
int i = 0;

do {
  cout << i << endl;
  i++;
} while (i < 5);
```

All the above loops will print the following output -

```
0
1
2
3
4
```

The difference between the `while` and `do-while` loop is that the `do-while` loop will execute the block of code at least once, even if the condition is false.

Infinite Loop

Infinite loops are loops that run indefinitely. We can create an infinite loop using the following syntax -

```
for (;;) {
  cout << "Hello, World!" << endl;
}
```

```
while (true) {
  cout << "Hello, World!" << endl;
}
```

```
do {
  cout << "Hello, World!" << endl;
} while (true);
```

This needs to be used with caution as it can lead to system crashes if not handled properly. It is generally used in scenarios where we want the program to run indefinitely, such as server applications.

Functions

Functions are used to group a block of code that can be reused multiple times. We can define a function using the following syntax -

```
void greet() {
  cout << "Hello, World!" << endl;
}
```

Function Parameters

We can also pass parameters to a function.

```
float add(float a, float b) {
  return a + b;
}
```

In the above code -

- `float` is the return type of the function.
- `add` is the name of the function.
- `float a` and `float b` are the parameters of the function.
- `return a + b` is the return statement of the function.
- `void` is used when the function does not return any value.

Function Declaration and Definition

We can declare a function at the top of the file and define it later in the file.

```
// function declaration
float add(float a, float b);

int main() {
  cout << add(5, 3) << endl; // 8
  cout << add(10, 5) << endl; // 15

  return 0;
}

// function definition
float add(float a, float b) {
  return a + b;
}
```

Pointers

Pointers are used to store the memory address of another variable. This allows us to access the memory location of a variable and perform operations on it from a different location in the program.

Pointers with Functions

We can use pointers to modify the value of a variable from a function.

Without Pointers

```
void myBirthday(int age) {
  age++;
  cout << "Happy Birthday! You are " << age << " years old" << endl;
}

int main() {
  int age = 25;
  cout << age << endl; // 25
  myBirthday(age); // Happy Birthday! You are 26 years old
  cout << age << endl; // 25
  return 0;
}
```

So, the value of `age` is not modified in the `main` function even though it is modified in the `myBirthday` function. This is because the `age` variable is passed by value to the `myBirthday` function and a copy of the variable is created in the function and only the copy is modified.

With Pointers

```
void myBirthday(int *age) {
  (*age)++;
  cout << "Happy Birthday! You are " << *age << " years old" << endl;
}

int main() {
  int age = 25;
  cout << age << endl; // 25
  myBirthday(&age); // Happy Birthday! You are 26 years old
  cout << age << endl; // 26
  return 0;
}
```

In the above code -

- `int *age` is a pointer to an integer variable.
- `&age` is the memory address of the `age` variable.
- `(*age)++` is used to increment the value of the `age` variable.
- `*age` is used to dereference the pointer and access the value of the `age` variable.

Pointers with Arrays

We can use pointers to access the elements of an array. The name of the array is a pointer to the first element of the array.

```
 int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // points to address of first element

cout << *ptr << endl; // 1
cout << ptr << endl; // 0x61feec
cout << &ptr << endl; // 0x61fee8



ptr++; // points to address of second element

cout << *ptr << endl; // 2
cout << ptr << endl; // 0x61fef0
```

In the above code -

- `int *ptr` is a pointer to an integer variable.
- `arr` is the name of the array and is a pointer to the first element of the array.
- `*ptr` is used to dereference the pointer and access the value of the `arr` variable.
- `ptr++` is used to increment the pointer to point to the next element of the array.
- `&ptr` is the memory address of the `ptr` variable.

Classes and Objects

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

- A class is a user-defined data type that has data members and member functions
- A class is a template for objects, and an object is an instance of a class

Creating a class

```
 class Employee {
  string Name;
  int Age;
  string Company;
};
```

- `Name`, `Age`, and `Company` are member variables or attributes
- Every attribute of a class is private by default
- Since the attributes are private, they cannot be accessed directly from outside the class

Creating an object

```
 int main() {
  Employee employee1;
  employee1.Name = "John"; // Error: 'Name' is private within this context
}
```

To access the attributes of a class, we need to create a public member function or method

```
 class Employee {
  public: // Access modifier
    string Name;
    int Age;
    string Company;

    void introduce() {
      cout << "Name - " << Name << endl;
      cout << "Age - " << Age << endl;
      cout << "Company - " << Company << endl;

    }
};
```

Accessing the attributes of a class

```
int main() {
  Employee employee1;
  employee1.Name = "John";
  employee1.Age = 30;
  employee1.Company = "Google";

  employee1.introduce(); // Name - John, Age - 30, Company - Google
}
```

Access Modifiers

- `public` : Members are accessible from outside the class
- `private` : Members cannot be accessed (or viewed) from outside the class
- `protected` : Members cannot be accessed from outside the class, however, they can be accessed in inherited classes

```
class Employee {
  public:
    string Name;
    int Age;
    string Company;
  private:
    int Salary;
  protected:
    string Department;
};
```

Constructors

A constructor is a function that is automatically called when an object is created. It is used to initialize the object's attributes.

> As of now if we want to create 100 employees, we need to set the attributes of each employee manually. This is not efficient. We can use a constructor to initialize the attributes of an object when it is created.

Currently, we are using a default constructor, which is provided by the compiler if we don't define one.

Creating our constructor

```
class Employee {
  public:
    string Name;
    int Age;
    string Company;

    Employee(string name, int age, string company) {
      Name = name;
      Age = age;
      Company = company;
    }
};
```

Creating an object using a constructor

```
int main() {
  Employee employee1("John", 30, "Google");
  employee1.introduce(); // Name - John, Age - 30, Company - Google

  Employee employee2("Jane", 25, "Facebook");
  employee2.introduce(); // Name - Jane, Age - 25, Company - Facebook

  Employee employee3("Tom", 35, "Amazon");
  employee3.introduce(); // Name - Tom, Age - 35, Company - Amazon
}
```

Rule of Constructors

- A constructor has the same name as the class
- A constructor does not have a return type
- Constructors need to be public (there can be special cases where we need a private constructor, but that is not common)

Encapsulation

Encapsulation is the process of wrapping up the data (attributes) and the code (methods) into a single unit called a class. It is used to restrict access to certain attributes and methods of a class to prevent the data from being modified by accident.

Getters and Setters

To access or modify the attributes of a class, we create our own public member functions or methods called getters and setters.

- Getters are used to access the attributes of a class
- Setters are used to modify the attributes of a class

```
class Employee {
  private:
    int Salary;

  public:
    void setSalary(int salary) {
      if (salary < 0) {
        cout << "Salary cannot be negative" << endl;
      } else {
        Salary = salary;
      }
    }

    int getSalary() {
      return Salary;
    }
};
```

Using Getters and Setters

```
int main() {
  Employee employee1;
  employee1.setSalary(50000);
  cout << employee1.getSalary(); // 50000
}
```

Abstraction

Abstraction is the process of creating a simple model of a complex system. It hides the complex details of a system and only shows the necessary details of the object.

In C++, we can create an abstract class using the `virtual` keyword.

```
class AbstractEmployee {
  virtual void askForPromotion() = 0;
};
```

In the above example, `askForPromotion` is an abstract method. It is a method without a body. We use the `= 0` to make it an abstract method. We use the `virtual` keyword to make it an abstract class.

Rules of Abstract Classes

- An abstract class is a class that cannot be instantiated
- An abstract class can have abstract methods (methods without a body)
- Any class that inherits an abstract class must implement the abstract methods

```
class Employee: public AbstractEmployee {
  public:
    void askForPromotion() {
      if (Salary > 1000) {
        cout << "You got a promotion" << endl;
      } else {
        cout << "Sorry, no promotion for you" << endl;
      }
    }
};
```

If we don't implement the abstract method in the derived class, we will get an error.

Using an Abstract Class

```
int main() {
  Employee employee1;
  employee1.setSalary(50000);
  employee1.askForPromotion(); // You got a promotion
}
```

Inheritance

Inheritance is the process of creating a new class from an existing class. The new class is called a derived class, and the existing class is called a base class.

- The derived class inherits the attributes and methods of the base class
- The derived class can also have its attributes and methods

```
class Chef {
  public:
    void makeChicken() {
      cout << "The chef makes chicken" << endl;
    }
    void makeSalad() {
      cout << "The chef makes salad" << endl;
    }
};

class ItalianChef: public Chef {
  public:
    void makePasta() {
      cout << "The chef makes pasta" << endl;
    }
};
```

Using Inheritance

```
int main() {
  ItalianChef chef;
  chef.makeChicken(); // The chef makes chicken
  chef.makePasta(); // The chef makes pasta
}
```

Polymorphism

Polymorphism is the ability of an object to take on many forms. It is used to perform a single action in different ways.

- Polymorphism is one of the primary features of OOP
- Polymorphism allows us to perform a single action in different ways
- Polymorphism is achieved by method overloading and method overriding

Method Overloading

Method overloading is the process of using the same method name for different methods. It is used to create multiple methods with the same name but different parameters.

```
class Employee {
  public:
    void work() {
      cout << "Employee is working" << endl;
    }
    void work(int hours) {
      cout << "Employee is working for " << hours << " hours" << endl;
    }
};
```

Using Method Overloading

```
int main() {
  Employee employee;
  employee.work(); // Employee is working
  employee.work(8); // Employee is working for 8 hours
}
```

Method Overriding

Method overriding is the process of redefining a base class method in a derived class. It is used to create a new implementation of a method in the derived class.

```
class Employee {
  public:
    virtual void work() {
      cout << "Employee is working" << endl;
    }
};

class Manager: public Employee {
  public:
    void work() {
      cout << "Manager is working" << endl;
    }
};

class Chef: public Employee {
};
```

In the above example, we use the `virtual` keyword to make the `work` a virtual method. This first checks if the derived class has a method with the same name. If it does, it uses the derived class method. If it doesn't, it uses the base class method.

Using Method Overriding

```
int main() {
  Manager manager;
  manager.work(); // Manager is working

  Chef chef;
  chef.work(); // Employee is working
}
```

Dynamic Polymorphism

Dynamic polymorphism is the process of using the base class pointer to call the derived class method. It can be used to perform a single action for all the different types of derived classes since the base class pointer can point to any derived class object.

```
int main() {
  Employee *employees[3]; = {&employee, &manager, &chef};

  for (int i = 0; i < 3; i++) {
    employees[i]->work();
  }
}
```

If we didn't use base class pointers, we would need to make a separate call to the `work` method for each derived class object since each Employee class instance is from a different type of derived class. Imagine the efficiency of this when we have 1000 derived class objects.

```
int main() {
  Employee employee;
  Manager manager;
  Chef chef;

  employee.work();
  manager.work();
  chef.work();
}
```

Abstraction vs Polymorphism

Abstract classes are used to provide a common interface for all the derived classes which means it tells the derived class what to do but not how to do it. It is the responsibility of the derived class to implement the abstract methods.

Abstract Class

```
class AbstractEmployee {
  virtual void startJob() = 0;
  virtual void morningMeeting() = 0;
  virtual void lunchBreak() = 0;
  virtual void endJob() = 0;
};
```

Derived Classes

```
 class Employee: public AbstractEmployee {
  public:
    void startJob() {
      cout << "Employee starts the job" << endl;
    }
    void morningMeeting() {
      cout << "Employee attends the morning meeting" << endl;
    }
    void lunchBreak() {
      cout << "Employee takes a lunch break" << endl;
    }
    void endJob() {
      cout << "Employee ends the job" << endl;
    }
};

class Manager: public AbstractEmployee {
  public:
    void startJob() {
      cout << "Manager starts the job" << endl;
    }
    void morningMeeting() {
      cout << "Manager attends the morning meeting" << endl;
    }
    void lunchBreak() {
      cout << "Manager takes a lunch break" << endl;
    }
    void endJob() {
      cout << "Manager ends the job" << endl;
    }
};
```

In Polymorphism, the derived class may or may not implement the method of the base class. The base class method has a default implementation, and the derived class can override it if it wants to.

Base Class

```
 class Employee {
  public:
    virtual void work() {
      cout << "Employee is working" << endl;
    }
};
```

Derived Classes

```

class Manager: public Employee {
  public:
    void work() {
      cout << "Manager is working" << endl;
    }
};

class Chef: public Employee {
};
```

OOPs Summary

- OOP is a programming paradigm that is based on the concept of objects
- A class is a blueprint for creating objects
- An object is an instance of a class
- A class has data members (attributes) and member functions (methods)
- Encapsulation is the process of wrapping up the data and the code into a single unit called a class
- Abstraction is the process of creating a simple model of a complex system

- Inheritance is the process of creating a new class from an existing class
- Polymorphism is the ability of an object to take on many forms
- Polymorphism is achieved by method overloading and method overriding

Conclusion

If you found the cheatsheet helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.