# Object-Oriented Programming in C++

## Classes and Objects

A C++ *class* is a user-defined data type that encapsulates information and behavior about an object.
A class can have two types of *class members*:

- *Attributes*, also known as member data, consist of information about an instance of the class.
- *Methods*, also known as member functions, are functions that can be used with an instance of the class.

An *object* is an instance of a class and can be created by specifying the class name.

```
#include <iostream>

class Dog {
public:
  int age;

  void sound() {
    std::cout << "woof\n";
  }
};

int main() {
  Dog buddy;

  buddy.age = 5;

  buddy.sound();         // Outputs: woof
}
```

## Access Specifiers

*Access specifiers* are C++ keywords that determine the scope of class components:

- `public` : Class members are accessible from anywhere in the program.
- `private` : Class members are only accessible from inside the class.

*Encapsulation* is achieved by declaring class attributes as `private` :

- Accessor functions: return the value of `private` member variables.
- Mutator functions: change the value of `private` member variables.

```
#include <iostream>

class Computer {
private:
  int password;

public:
  int getPassword() {
    return password;
  }

  void setPassword(int new_password) {
    password = new_password;
  }
};

int main()
{
  Computer dell;

  dell.setPassword(12345);
  std::cout << dell.getPassword();

  return 0;
}
```

## Constructors

For a C++ class, a *constructor* is a special kind of method that enables control regarding how the objects of a class should be created. Different class constructors can be specified for the same class, but each constructor signature must be unique.
A constructor can have multiple parameters as well as default parameter values.
In order to initialize `const` or reference type attributes, use *member initializer lists* instead of normal constructors.

```cpp
#include <iostream>

using namespace std;

class House {
private:
  std::string location;
  int rooms;

public:
  // Constructor with default parameters
  House(std::string loc = "New York", int num = 5) {
    location = loc;
    rooms = num;
  }

  // Destructor
  ~House() {
    std::cout << "Moved away from " << location << "\n";
  }
};

int main()
{
  House default_house;   // Calls House("New York", 5)
  House texas_house("Texas");    // Calls House("Texas", 5)
  House big_florida_house("Florida", 10); // Calls House("Florida", 10)

  return 0;
}
```

## Inheritance

In C++, a class can inherit attributes and methods from another class. In an inheritance relationship, there are two categories of classes:

- *Base class*: The class being inherited from.
- *Derived class*: The class that inherits from the base class.

It's possible to have multi-level inheritance where classes are constructed in order from the "most base" class to the "most derived" class.

```
#include <iostream>

class Base {
public:
  int base_id;

  Base(int new_base) : base_id(new_base)
{}
};

class Derived: public Base {
public:
  int derived_id;

  Derived(int new_base, int new_derived)
    : Base(new_base),
derived_id(new_derived) {}

  void show() {
    std::cout << base_id << " " <<
derived_id;
  }
};

int main() {
  Derived temp(1, 2);

  temp.show(); // Outputs: 1 2

  return 0;
}
```

## Polymorphism

In C++, *polymorphism* occurs when a derived class overrides a method inherited from its base class with the same function signature.

Polymorphism gives a method many “forms”. Which form is executed depends on the type of the caller object.

```
#include <iostream>

class Employee {
public:
  void salary() {
    std::cout << "Normal salary.\n";
  }
};

class Manager: public Employee {
public:
  void salary() {
    std::cout << "Normal salary and bonus.\n";
  }
};

int main() {
  Employee newbie;
  Manager boss;

  newbie.salary(); // Outputs: Normal salary.
  boss.salary(); // Outputs: Normal salary and bonus.

  return 0;
}
```

## Class Members

A class is comprised of class members:

- *Attributes*, also known as member data, consist of information about an instance of the class.
- *Methods*, also known as member functions, are functions that can be used with an instance of the class.

```
class City {

  // Attribute
  int population;

public:
  // Method
  void add_resident() {
    population++;
  }

};
```

## Constructor

For a C++ class, a *constructor* is a special kind of method that enables control regarding how the objects of a class should be created. Different class constructors can be specified for the same class, but each constructor signature must be unique.

```
#include "city.hpp"

class City {

  std::string name;
  int population;

public:
  City(std::string new_name, int new_pop);

};
```

## Objects

In C++, an *object* is an instance of a class that encapsulates data and functionality pertaining to that data.

```
City nyc;
```

## Class

A C++ class is a user-defined data type that encapsulates information and behavior about an object. It serves as a blueprint for future inherited classes.

```
class Person {

};
```

### Access Control Operators

C++ classes have access control operators that designate the scope of class members:

- public
- private

public members are accessible everywhere; private members can only be accessed from within the same instance of the class or from friends classes.

```
class City {

  int population;

public:
  void add_resident() {
    population++;
  }

private:
  bool is_capital;

};
```

Print Share ▼

# C++'s Built-In Data Structures

## arrays

Arrays in C++ are used to store a collection of values of the same type. The size of an array is specified when it is declared and cannot change afterward.
Use `[]` and an integer index to access an array element.
Keep in mind: array indices start with `0`, not `1`!.
A multidimensional array is an “array of arrays” and is declared by adding extra sets of indices to the array name.

```
#include <iostream>

using namespace std;

int main()
{
  char vowels[5] = {'a', 'e', 'i', 'o',
'u'};

  std::cout << vowels[2];       //
Outputs: i

  char game[3][3] = {
    {'x', 'o', 'o'} ,
    {'o', 'x', 'x'} ,
    {'o', 'o', 'x'}
  };

        std::cout << game[0][2];        //
Outputs: o

  return 0;
}
```

## vectors

In C++, a vector is a data structure that stores a sequence of elements that can be accessed by index.
Unlike arrays, vectors can dynamically shrink and grow in size.
The standard `<vector>` library provide methods for vector operations:

- `.push_back()` : add element to the end of the vector.
- `.pop_back()` : remove element from the end of the vector.
- `.size()` : return the size of the vector.
- `.empty()` : return whether the vector is empty.

```
#include <iostream>
#include <vector>

int main () {
  std::vector <int> primes = {2, 3, 5, 7,
11};

  std::cout << primes[2];       //
Outputs: 5

  primes.push_back(13);
  primes.push_back(17);
  primes.pop_back();

  for (int i = 0; i < primes.size(); i++)
{
      std::cout << primes[i] << " ";
  }
  // Outputs: 2 3 5 7 11 13

  return 0;
}
```

## Stacks and Queues

In C++, *stacks* and *queues* are data structures for storing data in specific orders.
Stacks are designed to operate in a **Last-In-First-Out** context (LIFO), where elements are inserted and extracted only from one end of the container.

- .push() add an element at the top of the stack.
- .pop() remove the element at the top of the stack.

Queues are designed to operate in a **First-In-First-Out** context (FIFO), where elements are inserted into one end of the container and extracted from the other.

- .push() add an element at the end of the queue.
- .pop() remove the element at the front of the queue.

```
#include <iostream>
#include <stack>
#include <queue>

int main()
{
  std::stack<int> tower;

  tower.push(3);
  tower.push(2);
  tower.push(1);

  while(!tower.empty()) {
    std::cout << tower.top() << " ";
    tower.pop();
  }
  // Outputs: 1 2 3

  std::queue<int> order;

  order.push(10);
  order.push(9);
  order.push(8);

  while(!order.empty()) {
    std::cout << order.front() << " ";
    order.pop();
  }
  // Outputs: 10 9 8

  return 0;
}
```

## Sets

In C++, a *set* is a data structure that contains a collection of unique elements. Elements of a set are index by their own values, or *keys*.

A set cannot contain duplicate elements. Once an element has been added to a set, that element cannot be modified.

The following methods apply to both `unordered_set` and `set` :

- `.insert()` : add an element to the set.
- `.erase()` : removes an element from the set.
- `.count()` : check whether an element exists in the set.
- `.size()` : return the size of the set.

```cpp
#include <iostream>
#include <unordered_set>
#include <set>

int main()
{
  std::unordered_set<int> primes({2, 3, 5, 7});

  primes.insert(11);
  primes.insert(13);
  primes.insert(11);  // Duplicates are not inserted

  primes.erase(2);
  primes.erase(13);

  // Outputs: primes does not contain 2.
  if(primes.count(2))
    std::cout << "primes contains 2.\n";
  else
    std::cout << "primes does not contain 2.\n";

  // Outputs: Size of primes: 4
  std::cout << "Size of primes: " << primes.size() << "\n";

  return 0;
}
```

## Hash Maps

In C++, a *hash map* is a data structure that contains a collection of unique elements in the form of *key-value* pairs. Elements of a hash map are identified by key values, while the *mapped values* are the content associated with the keys.

Each element of a `map` or `unordered_map` is an object of type `pair`. A `pair` object has two member variables:

- `.first` is the value of the key
- `.second` is the mapped value

The following methods apply to both `unordered_map` and `map`:

- `.insert()` : add an element to the map.
- `.erase()` : removes an element from the map.
- `.count()` : check whether an element exists in the map.
- `.size()` : return the size of the map.
- `[]` operater:
  - If the specified key matches an element in the map, then access the mapped value associated with that key.
  - If the specified key doesn’t match any element in the map, add a new element to the map with that key.

```
#include <iostream>
#include <unordered_map>
#include <map>

int main() {
  std::unordered_map<std::string, int>
country_codes;

  country_codes.insert({"Thailand", 65});
  country_codes.insert({"Peru", 51});
  country_codes["Japan"] = 81;           //
Add a new element
  country_codes["Thailand"] = 66; //
Access an element

  country_codes.erase("Peru");

  // Outputs: There isn't a code for
Belgium
  if (country_codes.count("Belgium")) {
    std::cout << "There is a code for
Belgium\n";
  }
  else {
   std::cout << "There isn't a code for
Belgium\n";
  }

  // Outputs: 81
  std::cout << country_codes["Japan"] <<
"\n";

  // Outputs: 2
  std::cout << country_codes.size() <<
"\n";

  // Outputs: Japan 81
  //          Thailand 66
  for(auto it: country_codes){
    std::cout << it.first << " " <<
it.second << "\n";
  }

  return 0;
}
```

Print   Share ▼