

C++ DATA STRUCTURES



Patrick McClanahan
San Joaquin Delta College

San Joaquin Delta College

C++ Data Structures

Patrick McClanahan

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Licensing

ProgramPage

1: OOP Concepts

- 1.1: Object Oriented Programming
- 1.2: OOP Class
- 1.3: OOP Objects
- 1.4: OOP Encapsulation
- 1.5: OOP Abstraction
- 1.6: OOP Polymorphism
- 1.7: OOP Inheritance
- 1.8: Classes and Objects
- 1.9: Accessing Data Members
- 1.10: Public Access Specifiers
- 1.11: Private Access Specifier
- 1.12: Protected Access Specifier

2: C++ Pointers

- 2.1: Pointer Review
- 2.2: How to use a pointer
- 2.3: References and Pointers
- 2.4: Array Name as Pointers
- 2.5: Pointer Arithmetic
- 2.6: More Pointer
- 2.7: Pointers and Literals
- 2.8: Pointers to Pointers
- 2.9: Void Pointers
- 2.10: Invalid and NULL Pointers

3: Dynamic Memory Allocation and Arrays

- 3.1: Dynamic memory allocation
- 3.2: Arrays in C++
 - 3.2.1: Arrays in C++ (continued)
- 3.3: Multidimensional Arrays

4: Inheritance

- 4.1: The Concept of Inheritance
- 4.2: Why and when to use inheritance?
- 4.3: Implementing inheritance in C++
- 4.4: Modes of Inheritance
- 4.5: Types of Inheritance in C++
- 4.6: Types of Inheritance in C++
- 4.7: Types of Inheritance

- 4.8: Types of Inheritance
- 4.9: Types of Inheritance
- 4.10: Programming Assignment- Inheritance

5: Polymorphism

- 5.1: Polymorphism in C++
- 5.2: Virtual Functions in C++
- 5.3: More Virtual Functions
- 5.4: Difference between Inheritance and Polymorphism

6: Abstraction / Encapsulation

- A.1: Difference between Abstraction and Encapsulation in C++
- A.2: Abstraction
- A.3: Encapsulation

7: Linked Lists

- 7.1: Linked Lists
- 7.2: Structures in C++
- 7.3: Structures in C++ (cont'd)
- 7.4: Structures in C++ (cont'd)
- 7.5: Linked Lists
- 7.6: Linked Lists vs Arrays
- 7.7: Create Linked List
- 7.8: Linked List Traversal
- 7.9: Linked List Insertion
- 7.10: Linked List Node Delete

8: Recursion

- 8.1: Recursion

9: Stack Data Structure

- 9.1: Stacks
- 9.2: Stack Implementation
- 9.3: Stack code

10: Queues Data Structures

- 10.1: Queues
- 10.2: Looking at the queue code
- 10.3: Looking at the queue code (continued)
- 10.4: Looking at the code (continued)
- 10.5: Look at the code (continued)
- 10.6: Circular queue

11: Hashing

- 11.1: Hashing
- 11.2: Hashing continued
- 11.3: Hashing continued

- [11.4: Hashing- Separate Chaining](#)
- [11.5: Hashing- Open Addressing](#)
- [11.6: Open Addressing continued](#)

[12: Heap](#)

[13: Sorting](#)

[14: Binary Trees](#)

- [14.1: Binary Tree](#)
- [14.2: Binary Tree Video](#)
- [14.3: Binary Tree Properties](#)
- [14.4: Types of binary trees](#)

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

This page is a draft and is under active development.



An Open Educational Resource Supported by the Academic Senate for California Community Colleges Open Educational Resources Initiative

The Academic Senate for California Community Colleges Open Educational Resources Initiative (OERI) was funded by the California legislature in trailer bill language during the summer of 2018. The OERI's mission is to reduce the cost of educational resources for students by expanding the availability and adoption of high quality Open Educational Resources (OER). The OERI facilitates and coordinates the curation and development of OER texts, ancillaries, and support systems. In addition, the OERI supports local OER implementation efforts through the provision of professional development, technical support, and technical resources.

The information in this resource is intended solely for use by the user who accepts full responsibility for its use. Although the author(s) and ASCCC OERI have made every effort to ensure that the information in this resource is accurate, openly licensed, and accessible at press time, the author(s) and ASCCC OERI do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Please bring all such errors and changes to the attention of Academic Senate of California Community Colleges OER Initiative [via e-mail](mailto:oeri@asccc.org) (oeri@asccc.org).

Academic Senate for California Community Colleges
One Capitol Mall, Suite 230
Sacramento, CA 95814

CHAPTER OVERVIEW

1: OOP Concepts

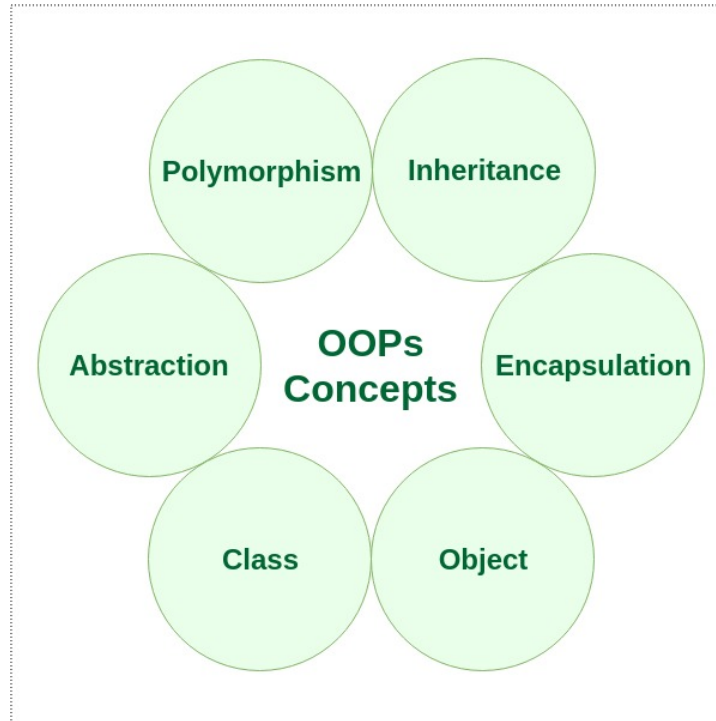
- [1.1: Object Oriented Programming](#)
- [1.2: OOP Class](#)
- [1.3: OOP Objects](#)
- [1.4: OOP Encapsulation](#)
- [1.5: OOP Abstraction](#)
- [1.6: OOP Polymorphism](#)
- [1.7: OOP Inheritance](#)
- [1.8: Classes and Objects](#)
- [1.9: Accessing Data Members](#)
- [1.10: Public Access Specifiers](#)
- [1.11: Private Access Specifier](#)
- [1.12: Protected Access Specifier](#)

This page titled [1: OOP Concepts](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.1: Object Oriented Programming

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming language



Adapted from:

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.1: Object Oriented Programming](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.2: OOP Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviors.

Adapted from:

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.2: OOP Class](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.3: OOP Objects

An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
    char name[20];
    int id;
    public:
        void getdetails(){}
};

int main()
{
    person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address just like any other data type in C++.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Adapted from

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.3: OOP Objects](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.4: OOP Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Adapted from:

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.4: OOP Encapsulation](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.5: OOP Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes:* We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files:* One more type of abstraction in C++ can be header files. For example, consider the `pow()` method present in `math.h` header file. Whenever we need to calculate the power of a number, we simply call the function `pow()` present in the `math.h` header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.



Contributed by Vankayala Karunakar
Geeks for Geeks

This page titled [1.5: OOP Abstraction](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.6: OOP Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.

Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.

Adapted from:

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

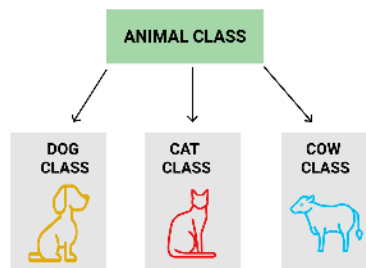
This page titled [1.6: OOP Polymorphism](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.7: OOP Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Adapted from:

"Object Oriented Programming in C++" by Vankayala Karunakar., [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.7: OOP Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.8: Classes and Objects

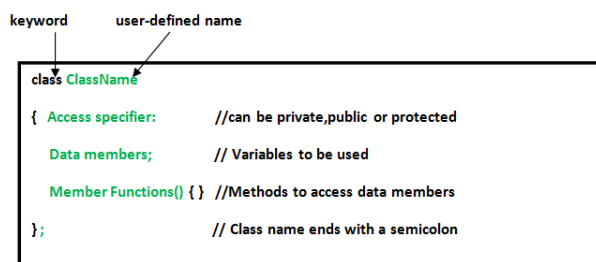
Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 *wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

A class is defined in C++ using keyword `class` followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()*.

Adapted from:

"C++ Classes and Objects" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.8: Classes and Objects](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.9: Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers : **public**, **private** and **protected**.

```
// C++ program to demonstrate
// accessing of data members
#include <bits/stdc++.h>
using namespace std;
class Geeks {

    // Access specifier
    public:

        // Data Members
        string geekname;

        // Member Functions()
        void printname()
        {
            cout << "Geekname is: " << geekname;
        }
};

int main()
{
    // Declare an object of class geeks
    Geeks obj1;
    // accessing data member
    obj1.geekname = "Steve Jobs";
    // accessing member function
    obj1.printname();

    return 0;
}
```

Output:

```
Geekname is: Abhi
```

Adapted from:

"C++ Classes and Objects" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.9: Accessing Data Members](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.10: Public Access Specifiers

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example:

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
using namespace std;

// class definition
class Circle
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5  
Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

Adapted from:

"C++ Classes and Objects" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.10: Public Access Specifiers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.11: Private Access Specifier

By **default** access to members of a C++ class is private.

The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double compute_area()
        {    // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }
};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
```

```
    return 0;
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

Output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
      double radius;
          ^
31:9: error: within this context
      obj.radius = 1.5;
          ^
```

However, we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        void compute_area(double r)
        {
            // member function can access private
            // data member radius
            radius = r;
            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
```

```
    }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class

    obj.compute_area(1.5);
    return 0;
}
```

Output:

```
Radius is: 1.5
Area is: 7.065
```

Adapted from:

"C++ Classes and Objects" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.11: Private Access Specifier](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

1.12: Protected Access Specifier

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example:

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;
// base class
class Parent
{
    // protected data members
    protected:
        int id_protected;
};

// sub class or derived class
class Child : public Parent
{
    public:
        void setId(int id)
        {
            // Child class is able to access the inherited
            // protected data members of base class
            id_protected = id;
        }

        void displayId()
        {
            cout << "id_protected is: " << id_protected << endl;
        }
};

// main function
int main()
{
```

```
Child obj1;

// member function of the derived class can
// access the protected data members of the base class
obj1.setId(81);
obj1.displayId();
return 0;
}
```

Output:

```
id_protected is: 81
```

Adapted from:

"C++ Classes and Objects" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [1.12: Protected Access Specifier](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

2: C++ Pointers

- [2.1: Pointer Review](#)
- [2.2: How to use a pointer](#)
- [2.3: References and Pointers](#)
- [2.4: Array Name as Pointers](#)
- [2.5: Pointer Arithmetic](#)
- [2.6: More Pointer](#)
- [2.7: Pointers and Literals](#)
- [2.8: Pointers to Pointers](#)
- [2.9: Void Pointers](#)
- [2.10: Invalid and NULL Pointers](#)

This page titled [2: C++ Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.1: Pointer Review

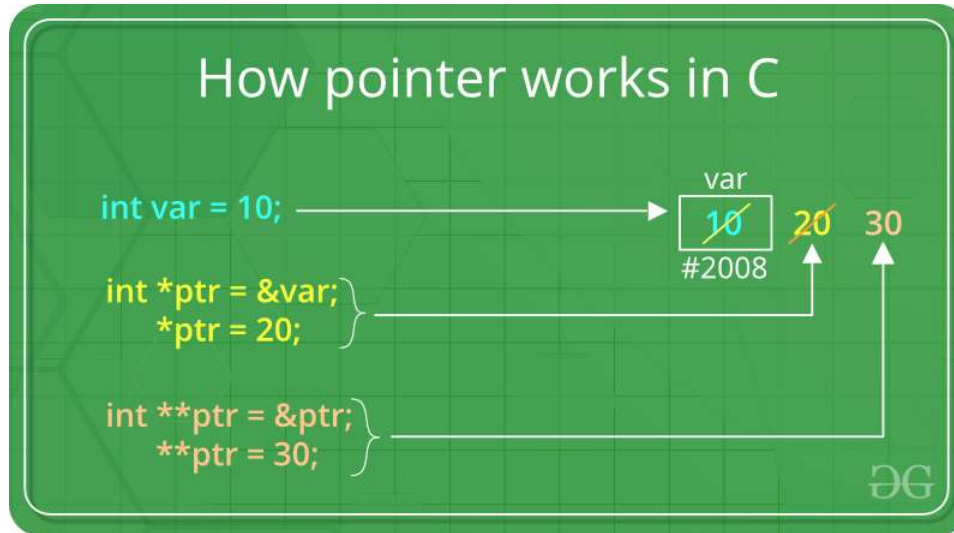
Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Its general declaration in C/C++ has the format:

Syntax:

```
datatype *var_name;
```

```
int *ptr; //ptr can point to an address which holds int data
```

Here is a visual example of how pointers work:



How to use a pointer?

- Define a pointer variable – you must use an asterisk (*) when defining a pointer variable. All three of the following are valid – they are the same:
 - `int *ptr;`
 - `int * ptr;`
 - `int* ptr;`
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

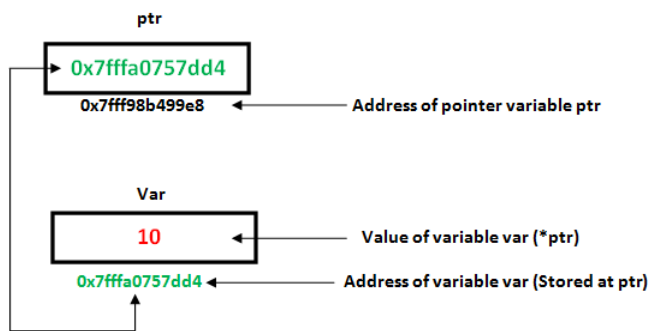
This page titled [2.1: Pointer Review](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.2: How to use a pointer

How to use a pointer?

- Define a pointer variable – you must use an asterisk (*) when defining a pointer variable. All three of the following are valid – they are the same:
 - `int *ptr;`
 - `int * ptr;`
 - `int* ptr;`
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.



```
// C++ program to illustrate Pointers in C++
#include <iostream>
using namespace std;

int main()
{
    int var = 20;

    // declare pointer variable
    int *ptr; // assignment happens at time of definition:
              // it would be: int *ptr = &var;

    // note that data type of ptr and var must be same
    // also note the absence of the * during this assignment
```

```
ptr = &var;

// assign the address of a variable to a pointer
cout << "Value at ptr = " << ptr << endl;
cout << "Value at var =" << var<< endl;
cout << "Value at *ptr =" << *ptr<< endl;

return 0;

}
```

Output:

Value at ptr = 0x7ffcb9e9ea4c

Value at var = 20

Value at *ptr = 20

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.2: How to use a pointer](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.3: References and Pointers

References and Pointers

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument

```
// C++ program to illustrate call-by-methods in C++

#include <iostream>
using namespace std;

//Pass-by-Value
int square1(int n)
{
    //Address of n in square1() is not the same as n1 in main()
    cout << "address of n1 in square1(): " << &n << "\n";

    // clone modified inside the function
    n *= n;
    return n;
}

//Pass-by-Reference with Pointer Arguments
void square2(int *n)
{
    //Address of n in square2() is the same as n2 in main()
    cout << "address of n2 in square2(): " << n << "\n";

    // Explicit de-referencing to get the value pointed-to
    *n *= *n;
}

//Pass-by-Reference with Reference Arguments
void square3(int &n)
{
    //Address of n in square3() is the same as n3 in main()
    cout << "address of n3 in square3(): " << &n << "\n";
}
```

```
// Implicit de-referencing (without '*')
n *= n;
}

int main()
{
    //Call-by-Value
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

    //Call-by-Reference with Pointer Arguments
    int n2=8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    //Call-by-Reference with Reference Arguments
    int n3=8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";

    return 0;
}
```

Output

```
address of n1 in main(): 0x7ffcdb2b4a44
address of n1 in square1(): 0x7ffcdb2b4a2c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7ffcdb2b4a48
address of n2 in square2(): 0x7ffcdb2b4a48
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffcdb2b4a4c
```

address of n3 in square3(): 0x7ffcdb2b4a4c

Square of n3: 64

Change reflected in n3: 64

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function.

If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.3: References and Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.4: Array Name as Pointers

Array Name as Pointers

An array name contains the address of first element of the array which acts like constant pointer. It means, the address stored in array name can't be changed.

For example, if we have an array named `val` then `val` and `&val[0]` can be used interchangeably.

```
// C program to illustrate call-by-methods
#include <iostream>
using namespace std;

int main()
{
    //Declare an array
    int val[3] = { 5, 10, 20 };

    //declare pointer variable
    int *ptr;

    //Assign the address of val[0] to ptr
    // We can use ptr=&val[0];(both are same)
    ptr = val ;
    cout << "Elements of the array are: ";
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2] << endl;

    return 0;
}
```

Output:

Elements of the array are: 5 10 20

val[0]	val[1]	val[2]
5	10	15
ptr[0]	ptr[1]	ptr[2]

If pointer `ptr` is sent to a function as an argument, the array `val` can be accessed in a similar fashion.

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.4: Array Name as Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.5: Pointer Arithmetic

Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers which are:

- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

```
// C++ program to illustrate Pointer Arithmetic in C++
#include <iostream>
using namespace std;

int main()
{
    //Declare an array
    int v[3] = {10, 100, 200};

    //declare pointer variable
    int *ptr;

    //Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++)
    {
        cout << "Value at ptr = " << ptr << "\n";
        cout << "Value at *ptr = " << *ptr << "\n";

        // Increment pointer ptr by 1
        ptr++;
    }

    return 0;
}
```

Output:

Value at ptr = 0x7fff9a9e7920

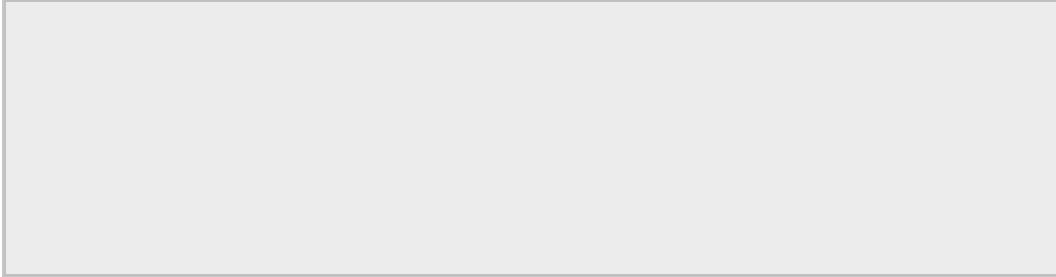
Value at *ptr = 10

Value at ptr = 0x7fff9a9e7924

Value at *ptr = 100

Value at ptr = 0x7fff9a9e7928

Value at *ptr = 200



Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.5: Pointer Arithmetic](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.6: More Pointer

Advanced Pointer Notation

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

```
int nums[2][3] = { { 16, 18, 20 }, { 25, 26, 27 } };
```

In general, `nums[ i ][ j ]` is equivalent to `*(*(nums+i)+j)`

Pointer Notation	Array Notation	Value
<code>*(*nums)</code>	<code>nums[0][0]</code>	16
<code>*(*nums+1)</code>	<code>nums[0][1]</code>	18
<code>*(*nums+2)</code>	<code>nums[0][2]</code>	20
<code>*(*(nums + 1))</code>	<code>nums[1][0]</code>	25
<code>*(*(nums + 1)+1)</code>	<code>nums[1][1]</code>	26
<code>*(*(nums + 1)+2)</code>	<code>nums[1][2]</code>	27

Yes - This can be a bit confusing...but if you slowly walk through each one it DOES make sense.

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.6: More Pointer](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.7: Pointers and Literals

Pointers and String literals

String literals are arrays containing null-terminated character sequences. String literals are arrays of type `char` plus terminating null-character, with each of the elements being of type `const char` (as characters of string can't be modified).

```
const char * ptr = "geek";
```

This declares an array with the literal representation for “geek”, and then a pointer to its first element is assigned to `ptr`. If we imagine that “geek” is stored at the memory locations that start at address 1800, we can represent the previous declaration as:

'g'	'e'	'e'	'k'	'\0'
1800	1801	1802	1803	1804

As pointers and arrays behave in the same way in expressions, `ptr` can be used to access the characters of string literal. For example:

```
char x = *(ptr+3);
```

```
char y = ptr[3];
```

Here, both `x` and `y` contain `k` stored at 1803 (1800+3).

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.7: Pointers and Literals](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.8: Pointers to Pointers

In C++, we can create a pointer to a pointer that in turn may point to data or other pointer. The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

```
char a;  
char *b;  
char ** c;  
a = 'g';  
b = &a;  
c = &b;
```

Here b points to a char that stores 'g' and c points to the pointer b.

Adapted from:

["Pointers in C/C++ with Examples"](#) by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.8: Pointers to Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.9: Void Pointers

There is some additional discussion of [Void Pointers](#)

This is a special type of pointer available in C++ which represents absence of type. void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

This means that void pointers have great flexibility as it can point to any data type. There is payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced. This transformation is called casting in C++. Notice in the following example - the first function, there is a void pointer for *data, BUT, in the if statement this variable is cast to a (char *). This forces the variable *data to be a char*. This is important for use of void pointers.

```
// C++ program to illustrate Void Pointer in C++
#include <iostream>
using namespace std;

void increase(void *data, int ptrsize)
{
    if(ptrsize == sizeof(char))
    {
        char *ptrchar;

        //Typecast data to a char pointer
        ptrchar = (char*)data;

        //Increase the char stored at *ptrchar by 1
        (*ptrchar)++;
        cout << "*data points to a char"<<"\n";
    }
    else if(ptrsize == sizeof(int))
    {
        int *ptrint;

        //Typecast data to a int pointer
        ptrint = (int*)data;

        //Increase the int stored at *ptrchar by 1
        (*ptrint)++;
        cout << "*data points to an int"<<"\n";
    }
}
```

```
int main()
{
    // Declare a character
    char c='x';

    // Declare an integer
    int i=10;

    //Call increase function using a char and int address respectively
    increase(&c,sizeof(c));
    cout << "The new value of c is: " << c <<"\n";

    increase(&i,sizeof(i));
    cout << "The new value of i is: " << i <<"\n";

    return 0;
}
```

Output:

*data points to a char

The new value of c is: y

*data points to an int

The new value of i is: 11

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.9: Void Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

2.10: Invalid and NULL Pointers

Invalid pointers

A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

```
int *ptr1;  
int arr[10];  
int *ptr2 = arr+20;
```

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer.

(Note: invalid pointers do not necessarily raise compile errors)

NULL pointers

Additional information on [NULL Pointers](#) is available.

Null pointer is a pointer which point nowhere and not just an invalid address.

Following are 2 methods to assign a pointer as NULL;

```
int *ptr1 = 0;  
int *ptr2 = NULL;
```

Please note that 0 in the above code is used in a pointer context, and it is different from 0 as an integer

Adapted from:

"Pointers in C/C++ with Examples" by Abhirav Kariya, [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [2.10: Invalid and NULL Pointers](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

3: Dynamic Memory Allocation and Arrays

[3.1: Dynamic memory allocation](#)

[3.2: Arrays in C++](#)

[3.2.1: Arrays in C++ \(continued\)](#)

[3.3: Multidimensional Arrays](#)

This page titled [3: Dynamic Memory Allocation and Arrays](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

3.1: Dynamic memory allocation

new and delete Operators in C++ for Dynamic Memory

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack** (Refer [Memory Layout C Programs](#) for details).

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, [Tree](#), etc.

For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes [memory leak](#) (memory is not deallocated until program terminates).

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax to use new operator: To allocate memory of any data type, the syntax is:

```
pointer-variable = new data-type;
```

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

- Initialize memory: We can also initialize the memory using new operator:

```
pointer-variable = new data-type(value);
Example:
int *p = new int(25);
float *q = new float(75.25);
```

- Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type *data-type*.

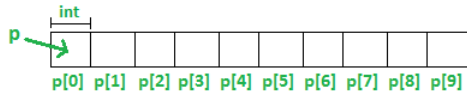
```
pointer-variable = new data-type[size];
```

where size(a variable) specifies the number of elements in an array.

Example:

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type `std::bad_alloc`, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer (scroll to section “Exception handling of new operator” in [this](#) article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
    cout << "Memory allocation failed\n";
}
```

delete Operator

Since it is programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by new.

Examples:

```
delete p;
delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

Example:

```
// It will free the entire array  
// pointed by p.  
delete[] p;
```

Adapted from:

"[new and delete operators in C++ for dynamic memory](#)" by Akash Gupta, [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

3.1: [Dynamic memory allocation](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

3.2: Arrays in C++

An array in C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They are used to store similar type of elements as in the data type must be the same for all elements. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C or C++ can store derived data types such as the structures, pointers etc. Given below is the picturesque representation of an array.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

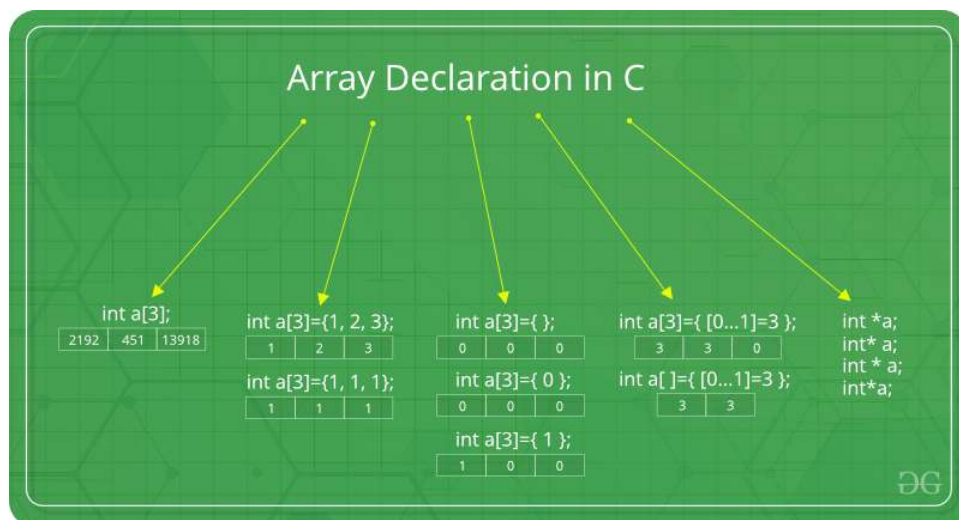
<- Array Indices

Array Length = 9
First Index = 0
Last Index = 8

Why do we need arrays?

We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

Array declaration in C++:



There are various ways in which we can declare an array. It can be done by specifying its type and size, by initializing it or both.

1. Array declaration by specifying size

```
// Array declaration by specifying size
int arr1[10];
// With recent C/C++ versions, we can also
// declare an array of user specified size
int n = 10;
int arr2[n];
```

2. Array declaration by initializing elements

```
// Array declaration by initializing elements
int arr[] = { 10, 20, 30, 40 }
```

```
// Compiler creates an array of size 4.  
// above is same as "int arr[4] = {10, 20, 30, 40}"
```

3. Array declaration by specifying size and initializing elements

```
// Array declaration by specifying size and initializing  
// elements  
int arr[6] = { 10, 20, 30, 40 }  
// Compiler creates an array of size 6, initializes first  
// 4 elements as specified by user and rest two elements as 0.  
// above is same as "int arr[] = {10, 20, 30, 40, 0, 0}"
```

Advantages of an Array in C/C++:

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Adapted From:

<https://www.geeksforgeeks.org/multidimensional-arrays-c-cpp>
<https://www.geeksforgeeks.org/arrays-in-c-cpp/>

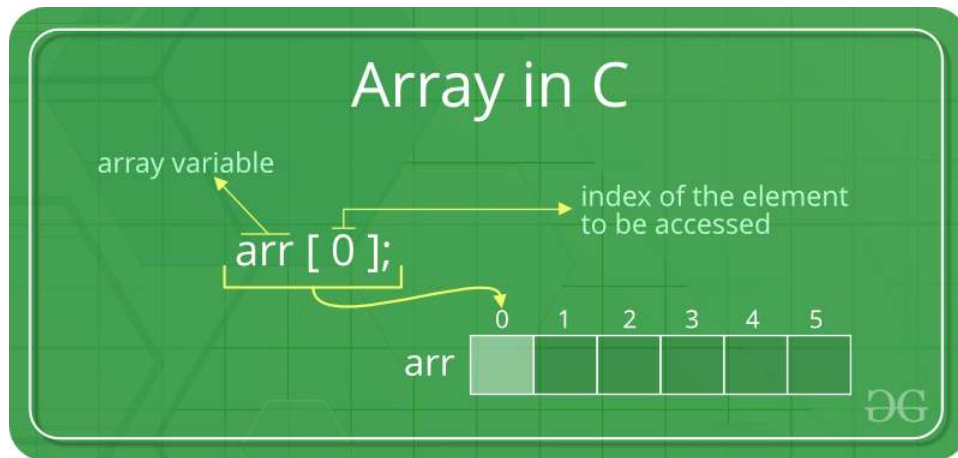
3.2: Arrays in C++ is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

3.2.1: Arrays in C++ (continued)

Facts about Array in C/C++:

- Accessing Array Elements:

Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



```
#include <iostream>
using namespace std;
int main()
{
    int arr[5];
    arr[0] = 5;
    arr[2] = -10;
    // this is same as arr[1] = 2
    arr[3 / 2] = 2;
    arr[3] = arr[0];
    cout << arr[0] << " " << arr[1] << " " << arr[2] << " " << arr[3];
    return 0;
}
```

- **Output:**

```
5 2 -10 5
```

- **No Index Out of bound Checking:**

There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
// This C++ program compiles fine
// as index out of bound
// is not checked in C.
#include <iostream>
using namespace std;
```

```
int main()
{
    int arr[2];
    cout << arr[3] << " ";
    cout << arr[-2] << " ";
    return 0;
}
```

Output: these are just meaningless values as its just whatever is sitting in memory locations relative to the array.

```
2008101287 4195777
```

In C, it is not compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just Warning.

```
#include <stdio.h>
int main()
{
    // Array declaration by initializing it with more
    // elements than specified size.
    int arr[2] = { 10, 20, 30, 40, 50 };
    return 0;
}
```

Warnings:

```
prog.c: In function 'main':
prog.c:7:25: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:25: note: (near initialization for 'arr')
prog.c:7:29: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:29: note: (near initialization for 'arr')
prog.c:7:33: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:33: note: (near initialization for 'arr')
```

Note: The program won't compile in C++. If we save the above program as a .cpp, the program generates compiler error "error: too many initializers for 'int [2]'".

- **The elements are stored at contiguous memory locations**

```
// C++ program to demonstrate that array elements
// are stored contiguous locations
#include <iostream>
```

```
using namespace std;
int main()
{
    // an array of 10 integers. If arr[0] is stored at
    // address x, then arr[1] is stored at x + sizeof(int)
    // arr[2] is stored at x + sizeof(int) + sizeof(int)
    // and so on.
    int arr[5], i;
    cout << "Size of integer in this compiler is " << sizeof(int) << "\n";
    for (i = 0; i < 5; i++)
        // The use of '&' before a variable name, yields
        // address of variable.
        cout << "Address arr[" << i << "] is " << &arr[i] << "\n";
    return 0;
}
```

Output:

```
Size of integer in this compiler is 4
Address arr[0] is 0x7ffd636b4260
Address arr[1] is 0x7ffd636b4264
Address arr[2] is 0x7ffd636b4268
Address arr[3] is 0x7ffd636b426c
Address arr[4] is 0x7ffd636b4270
```

Adapted From:

"Arrays in C/C++" by [sameer2209](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

3.2.1: Arrays in C++ (continued) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

3.3: Multidimensional Arrays

Multidimensional arrays in C++

In C/C++, initialization of a multidimensional arrays can have left most dimension as optional. Except the left most dimension, all other dimensions must be specified.

For example, following program fails in compilation. There are 3 dimensions in the declaration of the array `[][][2]` - two dimensions are not specified, and this will cause a compiler error.

```
#include <iostream>
using namespace std;
int main()
{
    int a[][][2] = { {{1, 2}, {3, 4}},
                    {{5, 6}, {7, 8}}
                  }; // error
    cout << "Sizeof the array " << sizeof(a) << endl;

    return 0;
}
```

The following 2 programs work without any error. The cout statement prints 4 times the sizeof an int - because the array has 4 members.

```
// Program 1
#include <iostream>
using namespace std;
int main()
{
    int a[][2] = {{1,2},{3,4}}; // Works
    cout << "Sizeof the array " << sizeof(a) << endl;
    return 0;
}
```

The following works, BECAUSE there are now 2 of the dimensions specified. The cout statement prints 8 times the sizeof an int - because the array has 8 members.

```
// Program 2
#include<stdio.h>
int main()
{
    int a[][2][2] = { {{1, 2}, {3, 4}},
                    {{5, 6}, {7, 8}}
                  }; // Works
    cout << "Sizeof the array " << sizeof(a) << endl;
}
```

```
    return 0;  
}
```

Hopefully this provides some clarification on how code can determine the size of an array.

Adapted from:

"Initialization of a multidimensional arrays in C/C++" by [HassanAli](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

3.3: Multidimensional Arrays is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

4: Inheritance

- [4.1: The Concept of Inheritance](#)
- [4.2: Why and when to use inheritance?](#)
- [4.3: Implementing inheritance in C++](#)
- [4.4: Modes of Inheritance](#)
- [4.5: Types of Inheritance in C++](#)
- [4.6: Types of Inheritance in C++](#)
- [4.7: Types of Inheritance](#)
- [4.8: Types of Inheritance](#)
- [4.9: Types of Inheritance](#)
- [4.10: Programming Assignment- Inheritance](#)

This page titled [4: Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.1: The Concept of Inheritance

Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

The article is divided into following subtopics:

1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

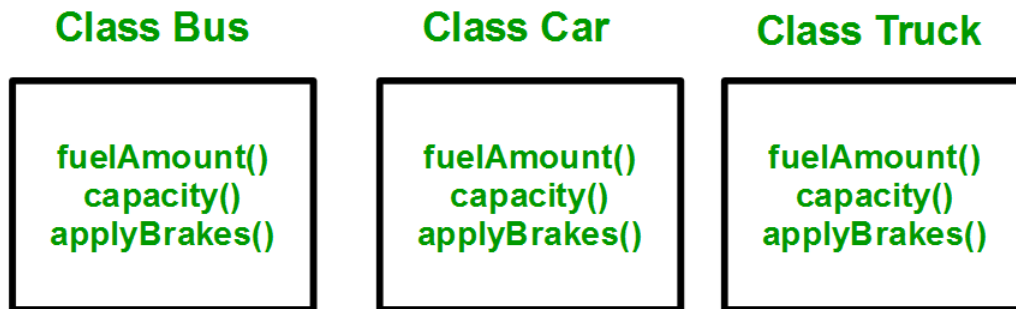
Adapted from:

"[Inheritance in C++](#)" by [Harsh Agarwal](#), [Geeks for Geeks](#) is licensed under [CC BY 4.0](#)

This page titled [4.1: The Concept of Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

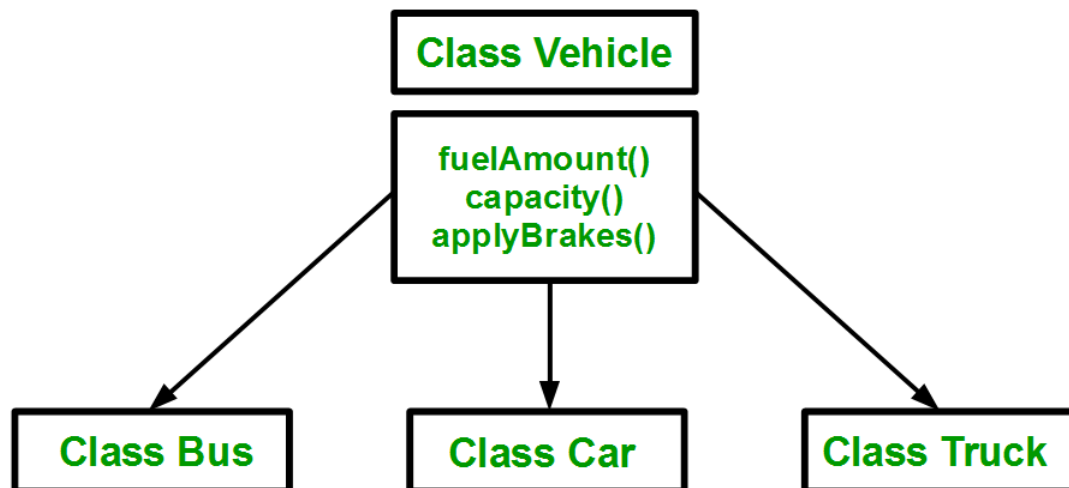
4.2: Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods `fuelAmount()`, `capacity()`, `applyBrakes()` will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Adapted from:

"Inheritance in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY 4.0

This page titled 4.2: Why and when to use inheritance? is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

4.3: Implementing inheritance in C++

For creating a sub-class which is inherited from the base class we have to follow the below syntax.

Syntax:

```
class subclass_name : access_mode base_class_name
{
    //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn't inherit **access** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// C++ program to demonstrate implementation
// of Inheritance

#include <bits/stdc++.h>
using namespace std;

//Base class
class Parent
{
    public:
    int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
    int id_c;
};

//main function
int main()
{
```

```
Child obj1;

// An object of class child has all data members
// and member functions of class parent
obj1.id_c = 7;
obj1.id_p = 91;
cout << "Child id is " << obj1.id_c << endl;
cout << "Parent id is " << obj1.id_p << endl;

return 0;
}
```

Output:

```
Child id is 7
Parent id is 91
```

In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.



Contributed by Harsh Agarwal

Geeks for Geeks

This page titled [4.3: Implementing inheritance in C++](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.4: Modes of Inheritance

We will discuss 3 modes of inheritance:

1. **Public mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode:** If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

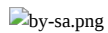
class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
```

```
class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)



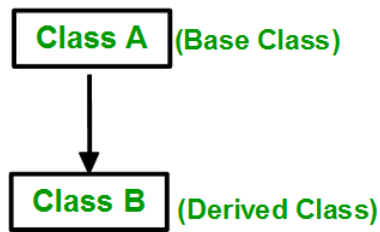
Contributed by Harsh Agarwal

Geeks for Geeks

This page titled [4.4: Modes of Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.5: Types of Inheritance in C++

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from the base classes
// although this class does NOTHING
class Car: public Vehicle
{
    public:
    Car()
```

```
{  
    cout << "This is a Car" << endl;  
}  
  
};  
  
// main function  
int main()  
{  
    // creating object of sub class will  
    // invoke the constructor of base classes  
    Car obj;  
    return 0;  
}
```

Output - notice that the parent class is initialized first, then the child class - which inherits from the parent class.

```
This is a Vehicle  
This is a Car
```

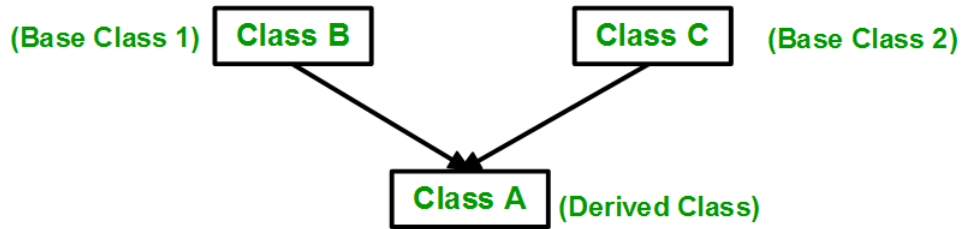


Contributed by Harsh Agarwal
Geeks for Geeks

This page titled [4.5: Types of Inheritance in C++](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.6: Types of Inheritance in C++

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2
{
    //body of subclass
};
```

Here, the number of base classes will be separated by a comma (',') and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler
{
public:
    FourWheeler()
```

```
{
    cout << "This is a 4 wheeler Vehicle" << endl;
}

};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler
{
    public:
    Car()
    {
        cout << "This vehicle is a 4 wheeldrive Car" << endl;
    }

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
This vehicle is a 4 wheeldrive Car
```

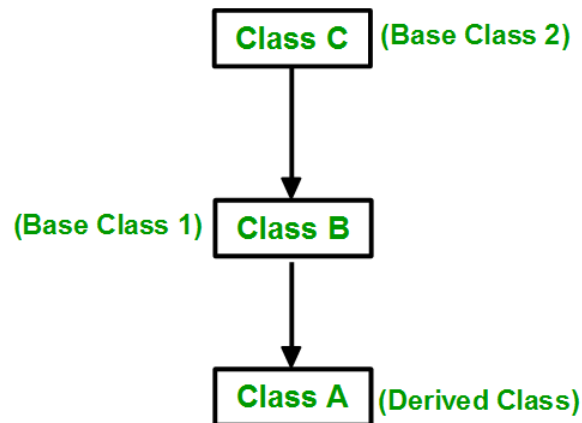


Contributed by Harsh Agarwal
Geeks for Geeks

This page titled [4.6: Types of Inheritance in C++](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.7: Types of Inheritance

Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
    public:
        Vehicle()
        {
            cout << "This is a Vehicle" << endl;
        }
};

class fourWheeler: public Vehicle
{
    public:
        fourWheeler()
        {
            cout<<"Objects with 4 wheels are vehicles"<<endl;
        }
};

// sub class derived from two base classes
```

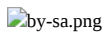
```
class Car: public fourWheeler
{
    public:
        Car()
        {
            cout<<"Car has 4 wheels"<<endl;
        }
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;

    return 0;
}
```

Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 wheels
```



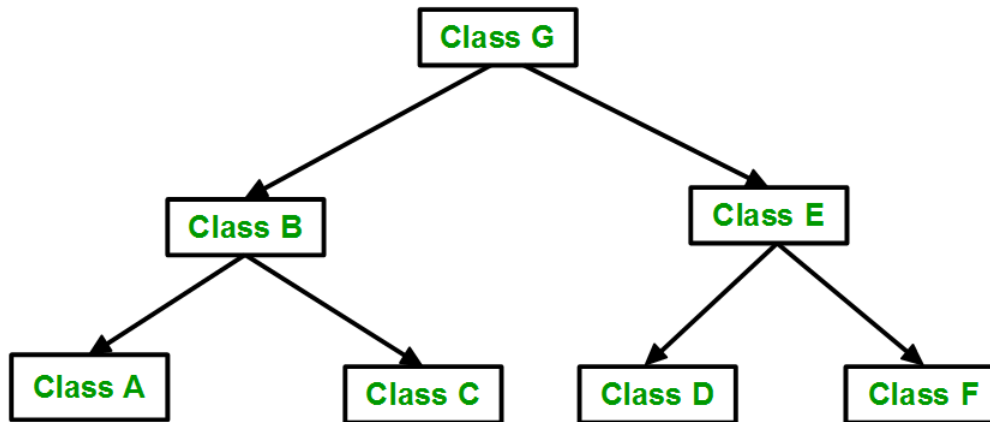
Contributed by Harsh Agarwal

Geeks for Geeks

This page titled [4.7: Types of Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.8: Types of Inheritance

Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// first sub class
class Car: public Vehicle
{
public:
    Car()
    {
        cout << "This is a Car" << endl;
    }
}
```

```
};

// second sub class
class Bus: public Vehicle
{
    public:
    Bus()
    {
        cout << "This is a Bus" << endl;
    }
};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;

    return 0;
}
```

Output - For each of the objects the parent class constructor gets called as well as the child class constructor.

```
This is a Vehicle
This is a Car
This is a Vehicle
This is a Bus
```



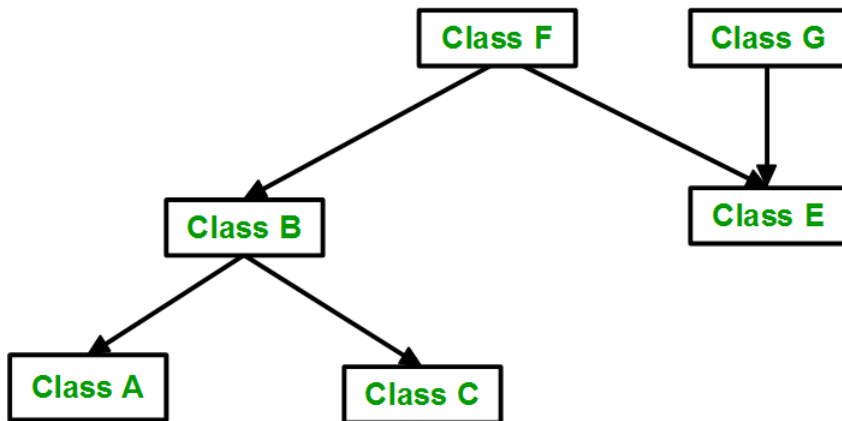
Contributed by Harsh Agarwal
Geeks for Geeks

This page titled [4.8: Types of Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.9: Types of Inheritance

Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:



```
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

//base class
class Fare
{
public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
```

```
// first child class - only inherits from Vehicle
class Car: public Vehicle
{
    public:
    Car()
    {
        cout << "Driving my car" << endl;
    }
};

// second child class - inherits from BOTH Vehicle and Fare
class Bus: public Vehicle, public Fare
{
    public:
    Bus()
    {
        cout << "Riding the bus" << endl;
    }
};

// main function
int main()
{
    // creating object of the child class will
    // invoke the constructor of base class - BOTH Vehicle and Fare
    Bus obj2;

    // creating a car object would NOT
    // invoke the constructor for Fare

    return 0;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```

Riding the bus

Notice that if we were to create an Car object that the Fare constructor is NOT called.



Contributed by Harsh Agarwal

Geeks for Geeks

This page titled [4.9: Types of Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

4.10: Programming Assignment- Inheritance

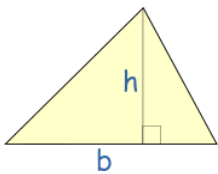
DO NOT FORGET - first three lines:

```
// Your Name  
// which class is this?  
// Assignment - Area of Shape
```

Write a program that

- defines a class named Shape with a constructor that accepts a width and height value.
- define two sub-classes: 1) a Triangle class and 2) a Rectangle class. These 2 classes will Inherit the 2 values from the Shape class
- In the main(), define two objects: 1) A Triangle object and 2) a Rectangle object
 - At time of creation provide the 2 values necessary
- call the getArea() function in this two class variables.
- DO NOT ask for user input....

Area of a triangle is calculated as: $(\text{height} * \text{base}) / 2$ (height - h and base - b in the following image)



Area of a rectangle is calculated as: $\text{height} * \text{width}$ (or $\text{length} * \text{width}$)

Submit your .cpp file

TEST DATA:

For an triangle with a base of 2 and a height of 5 the area would be 5

For a rectangle with a base (length) of 5 and a base of 3 the area would be 15

This page titled [4.10: Programming Assignment- Inheritance](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

5: Polymorphism

[5.1: Polymorphism in C++](#)

[5.2: Virtual Functions in C++](#)

[5.3: More Virtual Functions](#)

[5.4: Difference between Inheritance and Polymorphism](#)

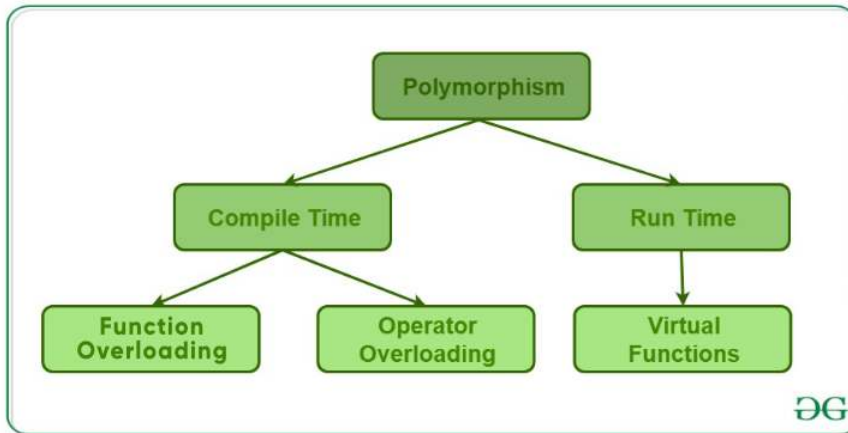
This page titled [5: Polymorphism](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

5.1: Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

- **Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Rules of Function Overloading

```
// C++ program for function overloading
#include <iostream>
using namespace std;

class Junk
{
    public:

    // function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name but 1 double parameter
    void func(double x)
    {
```

```
        cout << "value of x is " << x << endl;
    }

    // function with same name and 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};

int main()
{
    Junk newJunk;

    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    newJunk.func(7);

    // The second 'func' is called
    newJunk.func(9.132);

    // The third 'func' is called
    newJunk.func(85, 64);
    return 0;
}
```

Notice that the method `func()` is declared 3 times. The signature of each one is different - one accepts an `int` argument, the second accepts a `float` argument, and the third accepts 2 arguments. C++ figures out which one is to be called, based on the argument list, or upon the usage within the code.

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named `func` acts differently in three different situations which is the property of polymorphism.

- **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator (`+`) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator `+` when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;
```

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0) {real = r; imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

```
12 + i9
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit [this](#) link.

Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

- **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding
#include <iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    {
        cout<< "print in base class" <<endl;
    }
}
```

```
    }

    void show ()
    {
        cout<< "show in base class" <<endl;
    }
};

class derived:public base
{
    public:
        // print() is already virtual function in
        // derived class, we could have declared
        // it as virtual void print() explicitly
        void print ()
        {
            cout<< "print in derived class" <<endl;
        }

        void show ()
        {
            cout<< "show in derived class" <<endl;
        }
};

//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, bound at runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, bound at compile time
    bptr->show();

    return 0;
}
```

The idea is that **virtual functions** are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference. This means that in this example, bptr is pointing at a derived type, &d, so therefore at execution time we run the derived class print() method, because the base class print() method is virtual. Since the base class show() method is NOT virtual, then we run the base class show() method.

Output:

```
print in derived class  
show in base class
```

We will take a closer look at this example on the next page.

Adapted from:

"Polymorphism in C++" by [Harsh Agarwal](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

5.1: Polymorphism in C++ is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

5.2: Virtual Functions in C++

A virtual function is a member function which is declared within a base class and is re-defined(Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve [Runtime polymorphism](#)
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have [virtual destructor](#) but it cannot have a virtual constructor.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```
// C++ program for function overriding
#include <iostream>
using namespace std;

class base
{
    public:
        virtual void print ()
        {
            cout<< "print in base class" <<endl;
        }

        void show ()
        {
            cout<< "show in base class" <<endl;
        }
};

class derived:public base
{
    public:
        // print() is already virtual function in
        // derived class, we could have declared
        // it as virtual void print() explicitly
        void print ()
        {
```

```
        cout<< "print in derived class" <<endl;
    }

    void show ()
    {
        cout<< "show in derived class" <<endl;
    }
};

//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, bound at runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, bound at compile time
    bptr->show();

    return 0;
}
```

The idea is that **virtual functions** are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference. This means that in this example, bptr is pointing at a derived type, &d, so therefore at execution time we run the derived class print() method, because the base class print() method is virtual. Since the base class show() method is NOT virtual, then we run the base class show() method.

Output:

```
print in derived class
show in base class
```

Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type).

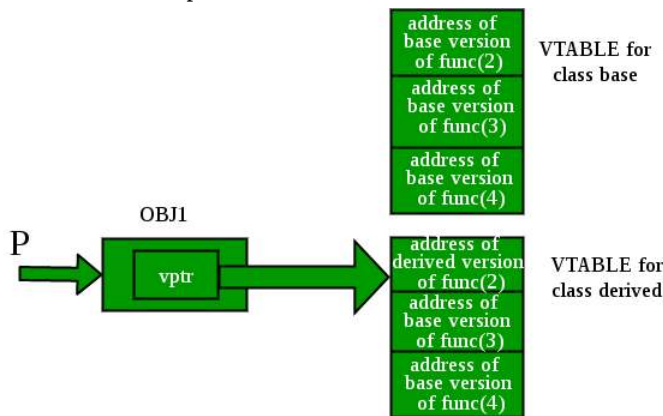
NOTE: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

Working of virtual functions(concept of VTABLE and VPTR)

As discussed [here](#), If a class contains a virtual function then compiler itself does two things:

1. If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of object is created or not, a **static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.

Consider the example below:



```
// CPP program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
```

```
p->fun_1();

// Late binding (RTP)
p->fun_2();

// Late binding (RTP)
p->fun_3();

// Late binding (RTP)
p->fun_4();

// The following is early binding but this
// function call is illegal(produces error)
// because pointer is of base type and
// function is of derived class
// p->fun_4(5);
}
```

Output:

```
base-1
derived-2
base-3
base-4
```

Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different.

Adapted from:

"Virtual Function in C++" by Yash Singla, [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

5.2: Virtual Functions in C++ is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

5.3: More Virtual Functions

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

Consider an employee management software for an organization.

Let the code has a simple base class *Employee*, the class contains virtual functions like `raiseSalary()`, `transfer()`, `promote()`, etc. Different types of employees like *Manager*, *Engineer*, etc. may have their own implementations of the virtual functions present in base class *Employee*.

In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don't need to worry about them because if `raiseSalary()` is present for a specific employee type, only that function would be called.

```
class Employee {
public:
    virtual void raiseSalary()
    {
        /* common raise salary code */
    }

    virtual void promote()
    {
        /* common promote code */
    }
};

class Manager : public Employee {
    virtual void raiseSalary()
    {
        /* Manager specific raise salary code, may contain
        increment of manager specific incentives*/
    }

    virtual void promote()
    {
        /* Manager specific promote */
    }
};

// Similarly, there may be other types of employees

// We need a very simple function
// to increment the salary of all employees
```

```
// Note that emp[] is an array of pointers
// and actual pointed objects can
// be any type of employees.
// This function should ideally
// be in a class like Organization,
// we have made it global to keep things simple
void globalRaiseSalary(Employee* emp[], int n)
{
    for (int i = 0; i < n; i++)

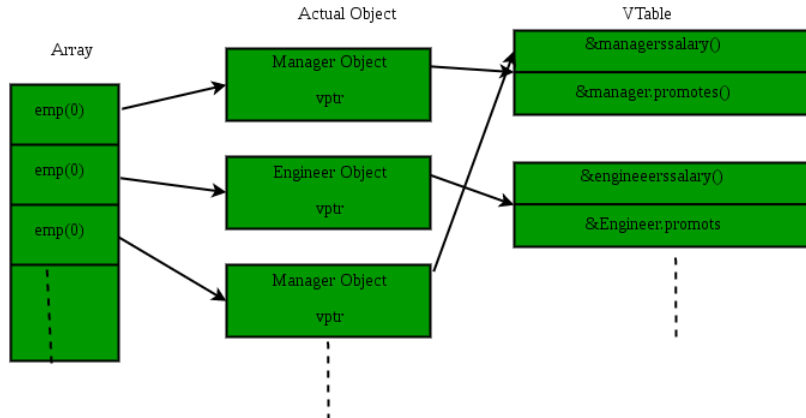
        // Polymorphic Call: Calls raiseSalary()
        // according to the actual object, not
        // according to the type of pointer
        emp[i]->raiseSalary();
}
```

Like *globalRaiseSalary()*, there can be many other operations that can be performed on a list of employees without even knowing the type of the object instance.

How does the compiler perform runtime resolution?

The compiler maintains two things to serve this purpose:

1. **vtable**: A table of function pointers, maintained per class.
2. **vpitr**: A pointer to vtable, maintained per object instance (see [this](#) for an example).



Compiler adds additional code at two places to maintain and use vpitr.

- 1) Code in every constructor. This code sets the vpitr of the object being created. This code sets vpitr to point to the vtable of the class.
- 2) Code with polymorphic function call (e.g. `bp->show()` in above code). Wherever a polymorphic call is made, the compiler inserts code to first look for vpitr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vpitr of derived class is accessed). Once vpitr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function `show()` is accessed and called.

Adapted from:

"Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)" by brain56, Geeks for Geeks is licensed under CC BY-SA 4.0

5.3: More Virtual Functions is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

5.4: Difference between Inheritance and Polymorphism

Inheritance:

Inheritance is one in which a new class is created that inherits the properties of the already exist class. It supports the concept of code reusability and reduces the length of the code in object-oriented programming.

Types of Inheritance are:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

Polymorphism:

Polymorphism is that in which we can perform a task in multiple forms or ways. It is applied to the functions or methods. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

Types of Polymorphism are:

1. Compile-time polymorphism (Method overloading)
2. Run-time polymorphism (Method Overriding)

Hopefully that makes it clear as to what the differences are. Here is a comparison table.

Difference between Inheritance and Polymorphism:

INHERITANCE		POLYMORPHISM
1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to classes.	Whereas it is basically applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
5.	It is used in pattern designing.	While it is also used in pattern designing.

Adapted from:

"Difference between Inheritance and Polymorphism" by MKS075, [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

5.4: Difference between Inheritance and Polymorphism is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

6: Abstraction / Encapsulation

[A.1: Difference between Abstraction and Encapsulation in C++](#)

[A.2: Abstraction](#)

[A.3: Encapsulation](#)

This page titled [6: Abstraction / Encapsulation](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

A.1: Difference between Abstraction and Encapsulation in C++

Abstraction

Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users. Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

Example of Abstraction:

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

class Summation {
private:
    // private variables
    int myNum1, myNum2, myNum3
public:
    void sum(int inNum1, int inNum2)
    {
        myNum1 = inNum1;
        myNum2 = inNum2;
        myNum3 = myNum1 + myNum2;
        cout << "Sum of the two number is : " << myNum3 << endl;
    }
};

int main()
{
    Summation mySum;
    mySum.sum(5, 4);
    return 0;
}
```

In this case the variables myNum1, myNum2 and myNum3 are private, thereby inaccessible to any code other than the class Summation. In this example the variables are set to values passed in as arguments to the sum method. This is not a very true example - often the values would NOT be set just before being used like this, but it shows the reality of the implementation.

Output:

```
Sum of the two number is: 9
```

Encapsulation

Encapsulation is a method of making a complex system easier to handle for end users. The user need not worry about internal details and complexities of the system. Encapsulation is a process of wrapping the data and the code, that operate on the data into a single entity. You can assume it as a protective wrapper that stops random access of code defined outside that wrapper.

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

class EncapsulationExample {
    private:
        // we declare a as private to hide it from outside
        int number1;

    public:
        // set() function to set the value of a
        void set(int input1)
        {
            number1 = input1;
        }

        // get() function to return the value of a
        int get()
        {
            return number1;
        }
};

// main function
int main()
{
    EncapsulationExample myInstance;
    myInstance.set(10);
    cout << myInstance.get() << endl;
    return 0;
}
```

In the this program, the variable number1 is made private so that this variable can be accessed and manipulated only by using the methods get() and set() that are present within the class. Therefore we can say that, the variable a and the methods set() as well as get() have bound together that is encapsulation. There is nothing special about the method names "get()" or "set()" - there may be other methods that manipulate the variable number1...all together this is called encapsulation.

Output:

```
10
```

Difference between Abstraction and Encapsulation:

ABSTRACTION		ENCAPSULATION
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
3.	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
4.	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
6.	The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

Adapted from:

"[Difference between Abstraction and Encapsulation in C++](#)" by [MKS075](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

[A.1: Difference between Abstraction and Encapsulation in C++](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

A.2: Abstraction

Abstraction in C++

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Abstraction using Classes: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the `pow()` method present in `math.h` header file. Whenever we need to calculate power of a number, we simply call the function `pow()` present in the `math.h` header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Abstraction using access specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class, can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

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

```
class implementAbstraction
{
    private:
        int num1, num2;
    public:

        // method to set values of
        // private members
        void set(int input1, int input2)
        {
            num1 = input1;
            num2 = input2;
        }
}
```

```
void display()
{
    cout<<"num1 = " << num1 << endl;
    cout<<"num2 = " << num2 << endl;
}

};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Output:

```
num1 = 10
num2 = 20
```

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

Adapted from:

"Abstraction in C++" by Harsh Agarwal, [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

A.2: Abstraction is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

A.3: Encapsulation

Encapsulation

Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

```
// c++ program to explain
// Encapsulation

#include<iostream>
using namespace std;

class Encapsulation
{
    private:
        // data hidden from outside world
        int someNum;

    public:
        // function to set value of
        // variable someNum
        void set(int inputVal)
        {
            someNum = inputVal;
        }

        // function to return value of
        // variable someNum
        int get()
        {
            return someNum;
        }
};

// main function
int main()
{
    Encapsulation myObj;
```

```
myObj.set(5);

cout << myObj.get() << endl;
return 0;
}
```

Output:

5

In the above program the variable **someNum** is made private. This variable can be accessed and manipulated only using the functions `get()` and `set()` which are present inside the class. Thus we can say that here, the variable `x` and the functions `get()` and `set()` are bound together which is nothing but encapsulation.

Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public** access specifier

Adapted from:

"Encapsulation in C++" by [Harsh Agarwal](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

A.3: Encapsulation is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

7: Linked Lists

- [7.1: Linked Lists](#)
- [7.2: Structures in C++](#)
- [7.3: Structures in C++ \(cont'd\)](#)
- [7.4: Structures in C++ \(cont'd\)](#)
- [7.5: Linked Lists](#)
- [7.6: Linked Lists vs Arrays](#)
- [7.7: Create Linked List](#)
- [7.8: Linked List Traversal](#)
- [7.9: Linked List Insertion](#)
- [7.10: Linked List Node Delete](#)

This page titled [7: Linked Lists](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.1: Linked Lists

Overview

We start into the realm of data structures with linked lists. As we grow in our understanding of software development we need to understand the more powerful concepts that we are starting to wander into.



Tasks to Complete in this Module

As you go through this module, you will be completing the following tasks:

1. Learn what a linked list is
2. Learn how to define a struct in C++
3. Learn to declare an instance of a struct in C++
4. How to allocate memory for a structure node
5. How to traverse a linked list
6. How to insert/delete a node

This page titled [7.1: Linked Lists](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.2: Structures in C++

In C++, a structure is the same as a class except for a few differences. The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details. Following are the points that expound on this difference:

1) Members of a class are private by default and members of a struct are public by default.

For example program 1 fails in compilation and program 2 works fine.

```
// Program 1
#include <iostream>
using namespace std

class Test {
    int x; // x is private
};

int main()
{
    Test t;

    t.x = 20; // compiler error because x is private
    return 0;
}
```

BUT...if we use a **struct** instead of **class** we get a different result

```
// Program 2
#include<iostream>
using namespace std;

struct Test {
    int x; // x is public
};

int main()
{
```

```
Test t;  
t.x = 20; // works fine because x is public  
return 0;  
}
```

Adapted from:

"Structure vs class in C++" by [roopkatha](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.2: Structures in C++](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.3: Structures in C++ (cont'd)

2) When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.

For example program 3 fails in compilation and program 4 works fine.

```
// Program 3
#include <iostream>
using namespace std;

class Base {
    public:
    int x;
};

class Derived : Base { }; // is equivalent to class Derived : private Base

int main()
{
    Derived d;
    d.x = 20; // compiler error because inheritance is private

    return 0;
}
```

So...if we use a **struct** instead of **class** we get a different result

```
// Program 4
#include <iostream>
using namespace std;

class Base {
    public:
    int x;
};

struct Derived : Base { }; // is equivalent to struct Derived : public Base
```

```
int main()
{
    Derived d;
    d.x = 20; // works fine because inheritance is public
    return 0;
}
```

Adapted from:

"Structure vs class in C++" by roopkatha, [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.3: Structures in C++ \(cont'd\)](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.4: Structures in C++ (cont'd)

1. **Member functions inside structure:** Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.
2. **Direct Initialization:** We cannot directly initialize structure data members in C but we can do it in C++.

```
// CPP program to initialize data member in c++
#include <iostream>
using namespace std;

struct Record {
    int x = 7;
};

// Driver Program
int main()
{
    Record s;
    cout << s.x << endl;
    return 0;
}
// Output
// 7
```

3. **Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use “struct Record” for Record variables. In C++, we need not use struct and using ‘Record’ only would work.
4. **Static Members:** C structures cannot have static members but is allowed in C++.

```
// C++ program with structure static member

struct Record {
    static int x;
};

// Driver program
int main()
{
```

```
    return 0;
}
```

5. **Constructor creation in structure:** Structures in C cannot have constructor inside structure but Structures in C++ can have Constructor creation.

```
// C program to illustrate empty structure
#include <stdio.h>

// empty structure
struct Record {

};
```

6. **sizeof operator:** This operator will generate **0** for an empty structure in C whereas **1** for an empty structure in C++.

```
// Driver program
int main()
{
    struct Record s;
    printf("%d\n", sizeof(s));
    return 0;
}
```

Output in C++:

```
1
```

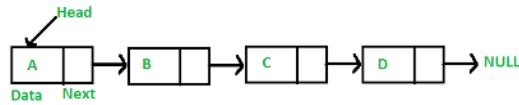
7. **Data Hiding:** C structures do not allow concept of Data hiding but is permitted in C++ as C++ is an object oriented language whereas C is not.
8. **Access Modifiers:** C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

Adapted from: "[Difference between C structures and C++ structures](#)" by [Gyanendra Singh Panwar](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.4: Structures in C++ \(cont'd\)](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.5: Linked Lists

Linked Lists are a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

This page titled [7.5: Linked Lists](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.6: Linked Lists vs Arrays

Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it [here](#).
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

- 1) Data
- 2) Pointer (Or Reference) to the next node

Adapted from:

"[Linked List | Set 1 \(Introduction\)](#)" by [ashwani khemani](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.6: Linked Lists vs Arrays](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.7: Create Linked List

First Simple Linked List in C Let us create a simple linked list with 3 nodes.

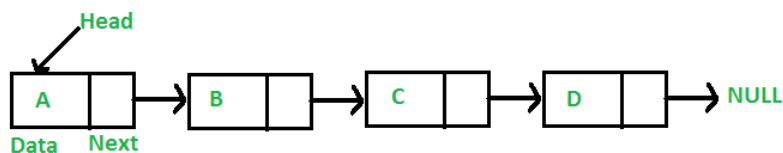
This code is covered in a video that follows this page...

```
// A simple CPP program to introduce
// a linked list
#include<bits/stdc++.h>
using namespace std;

class Node {
public:
    char data;
    Node* next;
};

// Program to create a simple linked
// list with 3 nodes
int main()
{
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node();
    second = new Node();
    third = new Node();
```



```
// the first node
head->data = 'A'; // assign data in first node
head->next = second; // Link first node with
```

```
// the second node
// assign data to second node
second->data = 'B';
// Link second node with the third node
second->next = third;

third->data = 'C'; // assign data to third node
third->next = NULL;

return 0;
}
```

"Linked List | Set 1 (Introduction)" by [ashwani khemani](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.7: Create Linked List](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.8: Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function `printList()` that prints any given list.

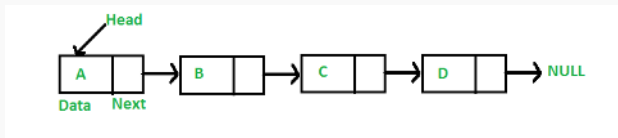
```
// A simple C++ program for traversal of a linked list
#include<bits/stdc++.h>
using namespace std;

struct Node {
    char data;
    Node* next;
};

// This function prints contents of linked list
// starting from the given node
void printList(Node* n)
{
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
    cout << endl;
}

// Driver code
int main()
{
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node();
    second = new Node();
    third = new Node();
```



```
head->data = 'A'; // assign data in first node
head->next = second; // Link first node with second

second->data = 'B'; // assign data to second node
second->next = third;

third->data = 'C'; // assign data to third node
third->next = NULL;

printList(head);

return 0;
}
```

Output:

A B C

"Linked List | Set 1 (Introduction)" by [ashwani khemani](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.8: Linked List Traversal](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

7.9: Linked List Insertion

Linked List Node Insert

Sometimes we have a linked list and we need to insert a node somewhere other than at the end of the list. We will take a look at a couple of different ways to insert a node into an existing list

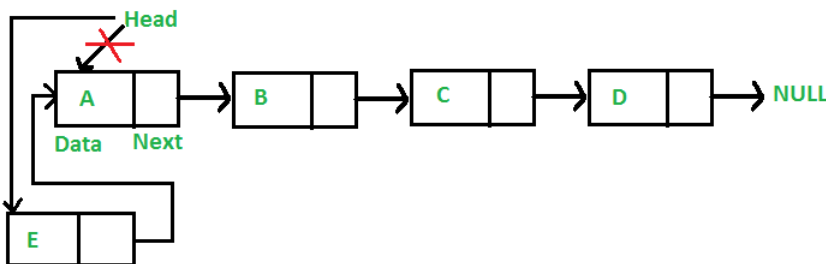
A new node can be added to a linked list in one of three ways

- 1) At the front of the linked list
- 2) After a given node.
- 3) At the end of the linked list.

Add a node at the front

The new node is added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Call the function that adds at the front of the list `push()`. This function must receive a pointer to the head pointer, because `push()` must change the head pointer to point to the new node.

In this picture we reset the HEAD to point to the new node, E, and then allow the new node's Next field point to the previous HEAD, A.



In this section all nodes have the format:

```
// A linked list node
class Node
{
    public:
        char data;
        Node *next;
};
```

There is a four step process to insert a new node at the head of the linked list:

1. Allocate memory for the new node
2. Put our data into the new node
3. Set Next of the new node to point to the previous Head
4. Reset Head to point to the new node

Here is the code for our `push()` function:

```
/* Given a reference (pointer to pointer)
to the head of a list and a char,
inserts a new node on the front of the list. */
```

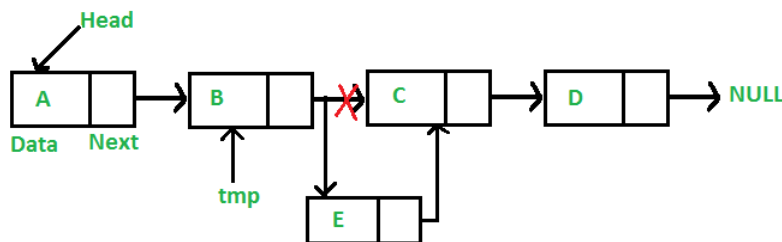
```
void push(Node** head_ref, char new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();
    /* 2. put in the data */
    new_node->data = new_data;
    /* 3. Make next of new node point to previous head */
    new_node->next = (*head_ref);
    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

This code performs our 4 steps:

1. allocate memory for a new node called new_node (you can call it whatever you want)
2. we insert the data, that was passed in as an argument, into the data field of the Node we just created
3. take the head_ref pointer, that points to the original head, and set the next field of the new node to point at the previous head
4. take the address of the new_node and reset the head_ref to point to this new node as the head of the linked list

Add a node after a given node:

As noted above, we can insert a node after a specific node. We are given pointer to a node, and the new node is inserted after the given node.



```
// Given a node prev_node, insert a
// new node after the given
// prev_node
void insertAfter(Node* prev_node, char new_data)
{
    // 1. Check if the given prev_node is NULL
    if (prev_node == NULL)
    {
        cout << "the given previous node cannot be NULL";
        return;
    }

    // 2. Allocate new node
    Node* new_node = new Node();

    // 3. Put in the data
```

```

new_node->data = new_data;

// 4. Make next of new node as next of prev_node
new_node->next = prev_node->next;

// 5. move the next of prev_node as new_node
prev_node->next = new_node;
}

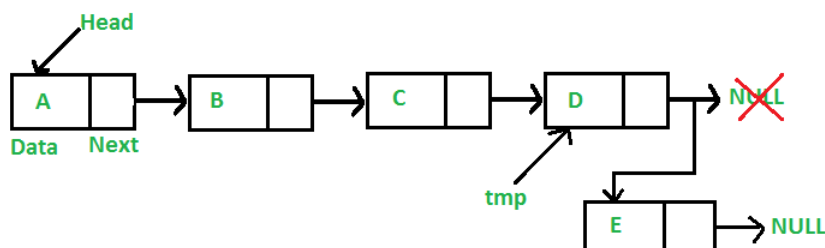
```

In this code the `push()` function is called with 2 arguments, the previous node, `prev_node`, and the data, `new_data`, for the new node. The code inserts the new node AFTER the previous node that is passed in. This function fails if the `prev_node` is NULL, if the `prev_node` is NULL, then something is wrong. We cannot insert a new node if we do not know where the previous node is. NOTICE: this is saying the NODE is NULL....NOT the `prev_node->next`. If we have a node where `prev_node` is NULL then we are at the end of the linked list, and we can insert a new node at the end of the list.

Add a node at the end:

The new node is added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

It is possible to insert a new node at the end of the linked list using the previous methodology. However, if we are given the head pointer, then we must traverse the linked list until we get to the end of the list. Once we find the end of the linked list - that is when the `prev_node->next` is NULL, then change the next of this last node to point to the new node, and set the next of the new node to NULL.



Here is the code for the above image. The Head is the address of the first node, the Next member is the memory address of the next node in the list.

Most of this code we have seen before, and should be at least familiar. This code is a good example of traversing the list, which is simply done by a short while loop. Step #5 shows how easy it is to walk across the list. The while loop condition looks at the Next member of the class, if it is set to NULL then we are at the end of the list. If the Next member is NOT NULL, we set the pointer `last` to the `last->next` then go back to the while condition and check for NULL again.

```

// Given a reference (pointer to pointer) to the head
// of a list and an char, appends a new node at the end
void append(Node** head_ref, char new_data)
{

    // 1. allocate node
    Node* new_node = new Node();

```

```
// Used in step 5
Node *last = *head_ref;

// 2. Put in the data
new_node->data = new_data;

// 3. This new node is going to be the last node, so make next of it as NULL
new_node->next = NULL;

// 4. If the Linked List is empty, then make the new node as head
if (*head_ref == NULL)
{
    *head_ref = new_node;
    return;
}

// 5. Else traverse till the last node
while (last->next != NULL)
    last = last->next;

// 6. Change the next of last node
last->next = new_node;
return;
}
```

Adapted from:

"[Linked List | Set 2 \(Inserting a node\)](#)" by [rathbhupendra](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.9: Linked List Insertion](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

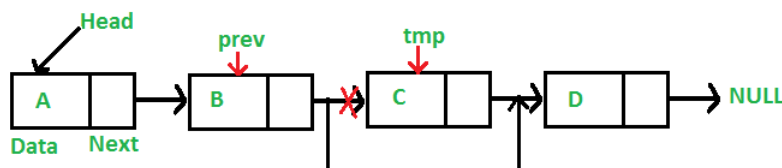
7.10: Linked List Node Delete

Delete Node

We have looked at creating a linked list, and we have worked through inserting a new node into a linked list. Now we will work through deleting a node from the list.

To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.



In this instance we remove the node where C is the Data. This simply means we reset the B node Next value to point the the D node. Remember: the Next value is a memory pointer that points the the place in memory where the various structures exist. Then we release the memory, using the `free()` function, location where the C node was.

The following code uses a struct instead of a Class, this allows you to see a slightly different way to implement linked lists. This is a fully implemented linked list using structures. Pay attention to the `printList()` function. Again, this is an example of walking across the list. Taking the head, which is the argument passed into this function, and seeing if it points to NULL. As long as it points to a valid address we print the Data member, and reset the node pointer to the current Next, then go back to the while statement.

```
// A complete working C program to demonstrate deletion in singly
// linked list
#include <iostream>
using namespace std;

// A linked list node
struct Node
{
    char data;
    struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
and a char, inserts a new node on the front of the list. */
void push(struct Node** head_ref, char new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
```

```
}

/* Given a reference (pointer to pointer) to the head of a list
and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct Node **head_ref, char key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;
    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next; // Changed head
        delete(temp);           // free old head memory
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL)
        return;
    // Unlink the node from linked list
    prev->next = temp->next;
    delete(temp); // Free memory
}

// This function prints contents of linked list starting from
// the given node
void printList(struct Node *node)
{
    while (node != NULL)
    {
        cout << node->data;
        node = node->next;
    }
    cout << endl;
}

/* Main to test above functions*/
int main()
```

```
{  
    /* Start with the empty list */  
    struct Node* head = NULL;  
  
    push(&head, 'A');  
    push(&head, 'B');  
    push(&head, 'C');  
    push(&head, 'D');  
  
    cout << "Created Linked List: " << endl;  
    printList(head);  
    deleteNode(&head, 'B');  
    cout << "Linked List after Deletion of 'C': " << endl;  
    printList(head);  
    return 0;  
}
```

Output:

```
Created Linked List:  
DCBA  
Linked List after Deletion of 'C':  
DCA
```

Adapted from:

["Linked List | Set 3 \(Deleting a node\)"](#) by [mlv](#), [Geeks for Geeks](#) is licensed under [CC BY-SA 4.0](#)

This page titled [7.10: Linked List Node Delete](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

8: Recursion

8.1: Recursion

This page titled [8: Recursion](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

8.1: Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi \(TOH\)](#), [Inorder/Preorder/Postorder Tree Traversals](#), [DFS of Graph](#), etc.

What is base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
    if (n <= 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, base case for $n \leq 1$ is defined and larger value of number can be solved by converting to smaller one till base case is reached.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of $(n-1)$. The base case for factorial would be $n = 0$. We return 1 when $n = 0$.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

If $\text{fact}(10)$ is called, it will call $\text{fact}(9)$, $\text{fact}(8)$, $\text{fact}(7)$ and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun . A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in Table 1.

```
// An example of direct recursion
```

```
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

What is difference between tailed and non-tailed recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function. Please refer [tail recursion article](#) for details.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

```
// A C++ program to demonstrate working of
// recursion
#include <bits/stdc++.h>
using namespace std;

void printFun(int test)
{
    if (test < 1)
        return;
    else {
        cout << test << " ";
```

```

        printFun(test - 1); // statement 2
        cout << test << " ";
        return;
    }
}

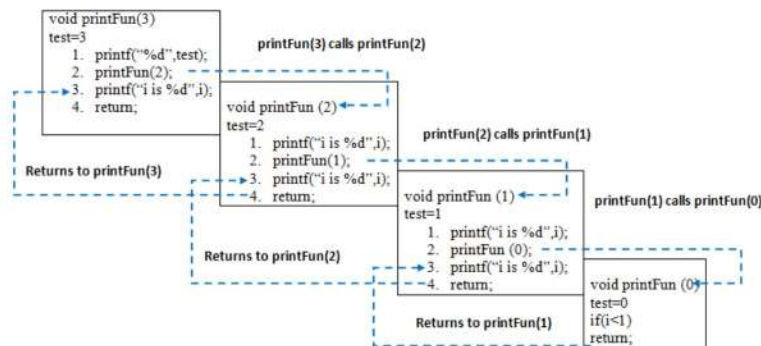
int main()
{
    int test = 3;
    printFun(test);
    return 0;
}

```

Output :

```
3 2 1 1 2 3
```

When **printFun(3)** is called from **main()**, memory is allocated to **printFun(3)** and a local variable **test** is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable **test** is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Below is a example of recursion. In this example the code performs a factorial. In mathematics, the factorial of a positive integer n , denoted by $n!$, is the product of all positive integers less than or equal to n :

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1.$$

For example,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

The value of $0!$ is 1, according to the convention for an empty product.

```
// Code to implement factorial
#include <iostream>
using namespace std;
// Factorial function
int factorial(int n)
{
    // Stop condition
    if (n == 0 || n == 1)
        return 1;
    // Recursive condition
    else
        // we recursively call the factorial function
        // UNTIL n is 0 OR 1...in which case we return 1
        return n * factorial(n - 1);
}

// Driver method
int main()
{
    int n = 5;
    cout << "Factorial of " << n << " is: " << factorial(n) << endl;
    return 0;
}
```

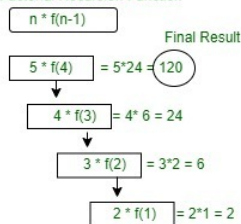
Output:

factorial of 5 is: 120

Working: (Read the text on the right from the BOTTOM UP

For user input : 5

Factorial Recursion Function



Finally - multiplying 5 by the returned value of 24....giving us the answer of 120 which is returned to main()



Then multiply 4 by the value returned giving us 24



Then it return 2, multiplied by 3, which returns 6



We call the factorial() function until we hit 0 or 1 in which case we return 1.

Adapted from:

"Recursion" by Sonal Tuteja, [Geeks for Geeks](https://www.geeksforgeeks.org/) is licensed under [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/)

8.1: Recursion is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

9: Stack Data Structure

[9.1: Stacks](#)

[9.2: Stack Implementation](#)

[9.3: Stack code](#)

This page titled [9: Stack Data Structure](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

9.1: Stacks

Stack Data Structure

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

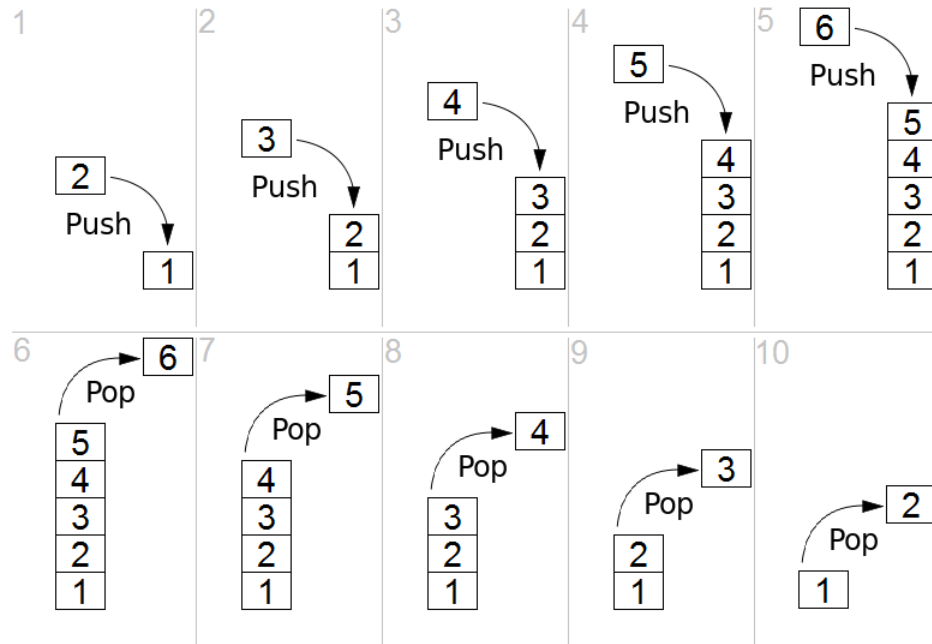


Figure 9.1.1:

Simple representation of a stack runtime with *push* and *pop* operations.

. (By Maxtremus - Own work, CC0, <https://commons.wikimedia.org/w/index.php?curid=44458752>)

The following four basic operations can be performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

There are many real-life examples of a stack. Consider an example of plates stacked in the cupboard. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottom most position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO (Last In First Out) / FILO (First In Last Out) order.

Adapted from: "[Stack Data Structure \(Introduction and Program\)](#)" by [Bhupendra Rathore](#) is licensed under [CC BY-SA 4.0](#)

This page titled [9.1: Stacks](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

9.2: Stack Implementation

There are two ways to implement a stack:

- Using array
 - **Pros:** Easy to implement. Memory is saved as pointers are not involved.
 - **Cons:** It is not dynamic. It doesn't grow and shrink depending on needs at runtime.
- Using linked list
 - **Pros:** The linked list implementation of stack can grow and shrink according to the needs at runtime.
 - **Cons:** Requires extra memory due to involvement of pointers.

We will look at using a Linked list to implement a Stack. Most of what we need to know we have already covered in our discussion of Linked Lists - stacks really only need a 'push' to build the stack. However, there are a couple of pieces that we need to add. In the "stack terminology" we have a 'pop' capability, which is just like the delete in our linked list. We also need to implement the 'peek' functionality - this simply returns the value that is sitting on the top of the stack, but does not alter the stack in any way. Lastly we will have to be able to determine if the stack is empty.

We will look at some of the code pieces next.

Adapted from: "[Stack Data Structure \(Introduction and Program\)](#)" by [Bhupendra Rathore](#) is licensed under [CC BY-SA 4.0](#)

This page titled [9.2: Stack Implementation](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

9.3: Stack code

Below is the code for implementing a stack using an array.

```
/* C++ program to implement basic stack operations using an array */
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

class Stack {
    int top;
public:
    int myArr[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int inVal);
    int pop();
    int peek();
    bool isEmpty();
};

// This is the push method for the Stack class
bool Stack::push(int inVal)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        myArr[++top] = inVal;
        cout << inVal << " pushed into stack\n";
        return true;
    }
}

// This is the pop method for the Stack class
int Stack::pop()
{
    if (top < 0) {
```

```
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int temp = myArr[top--];
        return temp;
    }
}

// This is the peek method for the Stack class
int Stack::peek()
{
    if (isEmpty) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int temp = myArr[top];
        return temp;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    class Stack myStack;
    myStack.push(10);
    myStack.push(20);
    myStack.push(30);
    cout << myStack.pop() << " Popped from stack\n";
    return 0;
}
```

Adapted from: "[Stack Data Structure \(Introduction and Program\)](#)" by [Bhupendra Rathore](#) is licensed under [CC BY-SA 4.0](#)

This page titled [9.3: Stack code](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

10: Queues Data Structures

10.1: Queues

10.2: Looking at the queue code

10.3: Looking at the queue code (continued)

10.4: Looking at the code (continued)

10.5: Look at the code (continued)

10.6: Circular queue

This page titled [10: Queues Data Structures](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.1: Queues

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out (FIFO)**. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the oldest item on the queue.

Operations on Queue:

Mainly the following two basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Some Queue implementations also provide two additional operations:

Front: Get the front item from queue.

Rear: Get the last item from queue.

 Picture showing Front of the Queue, where things are removed, and Rear of the Queue where things are added to the queue

Notice that they refer to the "rear" of the queue as where we add new items. The "front" is where we pull things off of the queue. This may seem a bit backwards, so make sure you have your terminology correct.

"Queue | Set 1 (Introduction and Array Implementation)" by [Soumik Mondal](#) is licensed under [CC BY-SA 4.0](#)

This page titled [10.1: Queues](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.2: Looking at the queue code

We have already learned how to use linked lists, and how to use that concept in creating and managing a Stack. So, we will take a look at the code for a Queue.

```
struct QNode {  
    int data;  
    QNode* next;  
};
```

This piece should look familiar. In the linked list we called it a Node, in the stack the example code called it a StackNode. It is simply a structure that contains our data and a pointer to the next item in the list/stack/queue.

Also - understand that the struct can contain multiple data types, it is not restricted to a single int. For example:

```
struct QNode {  
    int data;  
    string fName;  
    string lName;  
    float cost;  
    QNode* next;  
};
```

When we use these data types we manage the first, next and last points (and any other pointers we need to manage).

Adapted from: "[Queue – Linked List Implementation](#)" by [Gaurav Kumar](#) is licensed under [CC BY-SA 4.0](#)

This page titled [10.2: Looking at the queue code](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.3: Looking at the queue code (continued)

Let's take a quick look at the class we are going to define, and the constructor:

```
struct Queue {
    QNode *front, *rear;

    Queue()
    {
        front = rear = NULL;
    }
}
```

We want to keep track of the front of the queue - which is where we take things off of the queue, and we want to keep track of the rear of the queue, where we put things onto the queue. Remember, the default is that these pointers are private.

The constructor simply sets both the front and the rear pointers to NULL. So, when we declare an instance of this class in our code:

```
int main()
{
    Queue q;
```

the pointers are set to Null, as we have not yet allocated any memory for them.

Adapted from: "[Queue – Linked List Implementation](#)" by [Gaurav Kumar](#) is licensed under [CC BY-SA 4.0](#)

This page titled [10.3: Looking at the queue code \(continued\)](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.4: Looking at the code (continued)

To insert an item into the queue it is very similar to the linked list and the stack. This is the advantage of learning how to manage linked lists, they can be used in many other types of data structures.

```
void qInsert(int inVal)
{
    // Create a new LL node
    QNode* temp = new QNode();
    temp->data = inVal;
    temp->next = NULL;

    // If rear is NULL then we have a new
    // empty queue and this
    // new node is both front and rear
    if (rear == NULL) {
        front = rear = temp;
        return;
    }

    // Add the new node at
    // the end of queue and change rear
    rear->next = temp;
    rear = temp;
}
```

The first thing we need to do is allocate memory - which is done with the new QNode() code. Then the value that is passed as an argument to the qInsert() function is assigned to the temp node, and we set the next pointer to NULL,

If rear (and head) are NULL, we know that this is a new queue, and so we simply set both the front and the rear to the newly created temp value.

OTHERWISE

we simply have to adjust the current rear->next pointer to point at the temp value, and make rear point at this new item as the new rear.

Adapted from:

"Queue – Linked List Implementation" by [Gaurav Kumar](#) is licensed under [CC BY-SA 4.0](#)

This page titled [10.4: Looking at the code \(continued\)](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.5: Look at the code (continued)

The other functionality necessary is to remove an item from the front of the queue.

```
void qRemove()
{
    // If queue is empty, return NULL.
    if (front == NULL)
        return;

    // Store previous front and
    // move front one node ahead
    QNode* temp = front;
    front = front->next;

    // If front becomes NULL, then
    // change rear also as NULL
    if (front == NULL)
        rear = NULL;

    delete (temp);
}
}; //this is the end of the class Queue
```

If front is NULL - then we know that the queue is empty and we simply return to main().

OTHERWISE

Create a temp item, set it equal to the front item, and then reset front to the current front->next. After this we again check front to see if it is NULL, if so then we set rear to NULL as well. It is required to delete the temp space we allocated earlier.

Adapted from: "[Queue – Linked List Implementation](#)" by [Gaurav Kumar](#) is licensed under [CC BY-SA 4.0](#)

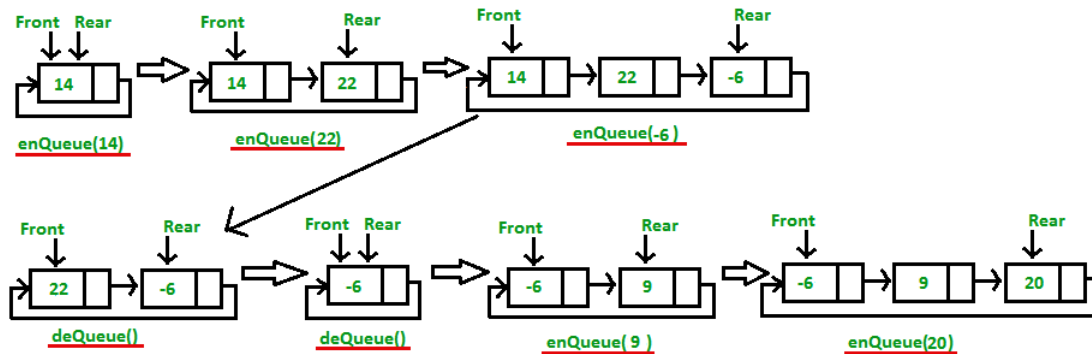
This page titled [10.5: Look at the code \(continued\)](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

10.6: Circular queue

Operations on Circular Queue:

- **qInsert(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position. **Steps:**
 1. Create a new node dynamically and insert value into it.
 2. Check if $\text{front} == \text{NULL}$, if it is true then $\text{front} = \text{rear} = (\text{newly created node})$
 3. If it is false then $\text{rear} = (\text{newly created node})$ and rear node always contains the address of the front node.
- **qRemove()** This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position. **Steps:**
 1. Check whether queue is empty or not means $\text{front} == \text{NULL}$.
 2. If it is empty then display Queue is empty. If queue is not empty then step 3
 3. Check if $(\text{front} == \text{rear})$ if it is true then set $\text{front} = \text{rear} = \text{NULL}$ else move the front forward in queue, update address of front in rear node and return the element.

Here is an example of inserting and removing a few values with a circular queue



This is the same as the non-circular EXCEPT that you must update the rear->next value to point to the head when you qRemove(), and set the new rear->next when you qInsert().

"Circular Queue | Set 2 (Circular Linked List Implementation)" by Arnab Kundu is licensed under CC BY-SA 4.0

This page titled 10.6: Circular queue is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

11: Hashing

[11.1: Hashing](#)

[11.2: Hashing continued](#)

[11.3: Hashing continued](#)

[11.4: Hashing- Separate Chaining](#)

[11.5: Hashing- Open Addressing](#)

[11.6: Open Addressing continued](#)

This page titled [11: Hashing](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

11.1: Hashing

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in $O(\log n)$ time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

Geeks for Geeks have their own YouTube channel - including a [video on Hashing](#).





"Hashing | Set 1 (Introduction)" by [Connor Fehrenbach](#) is licensed under [CC BY-SA 4.0](#)

This page titled [11.1: Hashing](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

11.2: Hashing continued

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in $O(\log n)$ time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in $O(1)$ time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get $O(1)$ search time on average (under reasonable assumptions) and $O(n)$ in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

"Hashing | Set 1 (Introduction)" by Connor Fehrenbach is licensed under CC BY-SA 4.0

This page titled 11.2: Hashing continued is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.3: Hashing continued

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

- 1) Efficiently computable.
- 2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

"Hashing | Set 1 (Introduction)" by [Connor Fehrenbach](#) is licensed under [CC BY-SA 4.0](#)

This page titled [11.3: Hashing continued](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

11.4: Hashing- Separate Chaining

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is [Birthday Paradox](#). With only 23 persons, the probability that two people have the same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

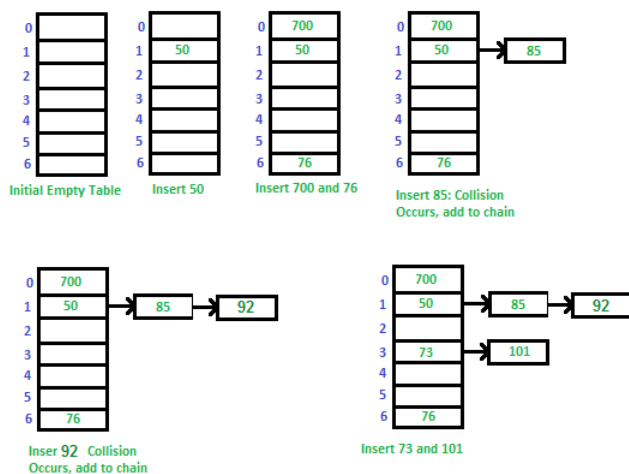
- 1) Separate Chaining
- 2) Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to the chain.
3. Less sensitive to the hash function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1. Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2. Wastage of Space (Some Parts of hash table are never used)
3. If the chain becomes long, then search time can become $O(n)$ in the worst case.
4. Uses extra space for links.

There is also a video that covers [Separate Chaining](#)



"Hashing | Set 2 (Separate Chaining)" by [Connor Fehrenbach](#) is licensed under [CC BY-SA 4.0](#)

This page titled [11.4: Hashing- Separate Chaining](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

11.5: Hashing- Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): **Delete operation is interesting**. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

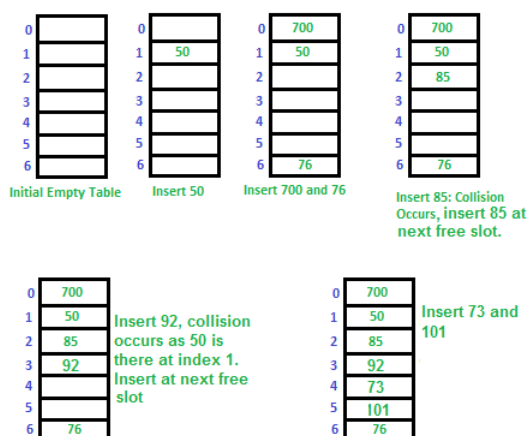
Open Addressing is done following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let $\text{hash}(x)$ be the slot index computed using hash function and S be the table size

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1) \% S$
 If $(\text{hash}(x) + 1) \% S$ is also full, then we try $(\text{hash}(x) + 2) \% S$
 If $(\text{hash}(x) + 2) \% S$ is also full, then we try $(\text{hash}(x) + 3) \% S$

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

"Hashing | Set 3 (Open Addressing)" by Pulkit Goel is licensed under CC BY-SA 4.0

This page titled 11.5: Hashing- Open Addressing is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.6: Open Addressing continued

b) **Quadratic Probing** We look for i^2 'th slot in i 'th iteration.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
.....
.....
```

c) **Double Hashing** We use another hash function hash2(x) and look for $i \cdot \text{hash2}(x)$ slot in i 'th rotation.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3
.....
.....
```

Geeks for Geeks provides [a video](#) covering these concepts.

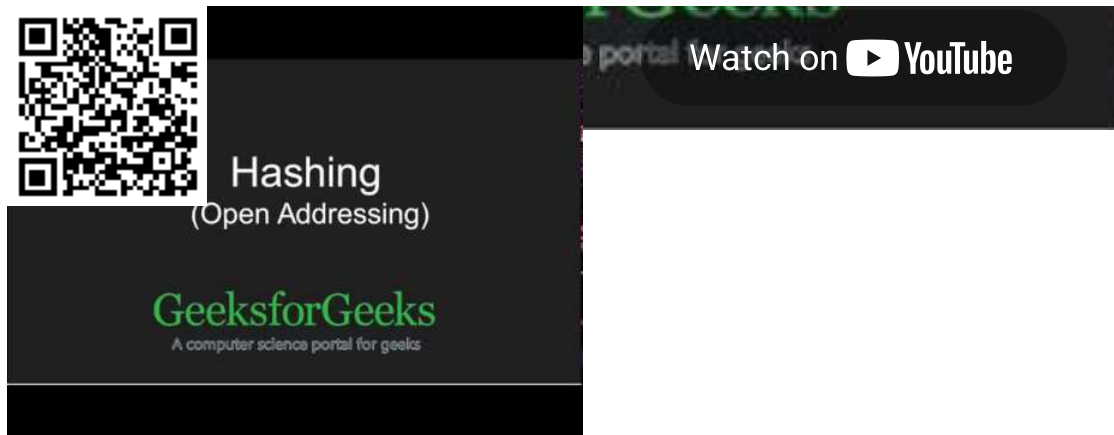
Comparison of above three:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.





"Hashing | Set 3 (Open Addressing)" by [Pulkit Goel](#) is licensed under [CC BY-SA 4.0](#)

This page titled [11.6: Open Addressing continued](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

12: Heap

This page titled [12: Heap](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

13: Sorting

This page titled [13: Sorting](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

CHAPTER OVERVIEW

14: Binary Trees

[14.1: Binary Tree](#)

[14.2: Binary Tree Video](#)

[14.3: Binary Tree Properties](#)

[14.4: Types of binary trees](#)

This page titled [14: Binary Trees](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

14.1: Binary Tree

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, 'a' is a child of 'f', and 'f' is the parent of 'a'. Finally, elements with no children are called leaves.

```
tree
----
  j    <-- root
 /  \
f    k
/  \  \
a   h  z    <-- leaves
```

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

```
file system
-----
  /    <-- root
 /      \
...      home
  /        \
ugrad      course
 /      /   |   \
...    cs101 cs112 cs113
```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

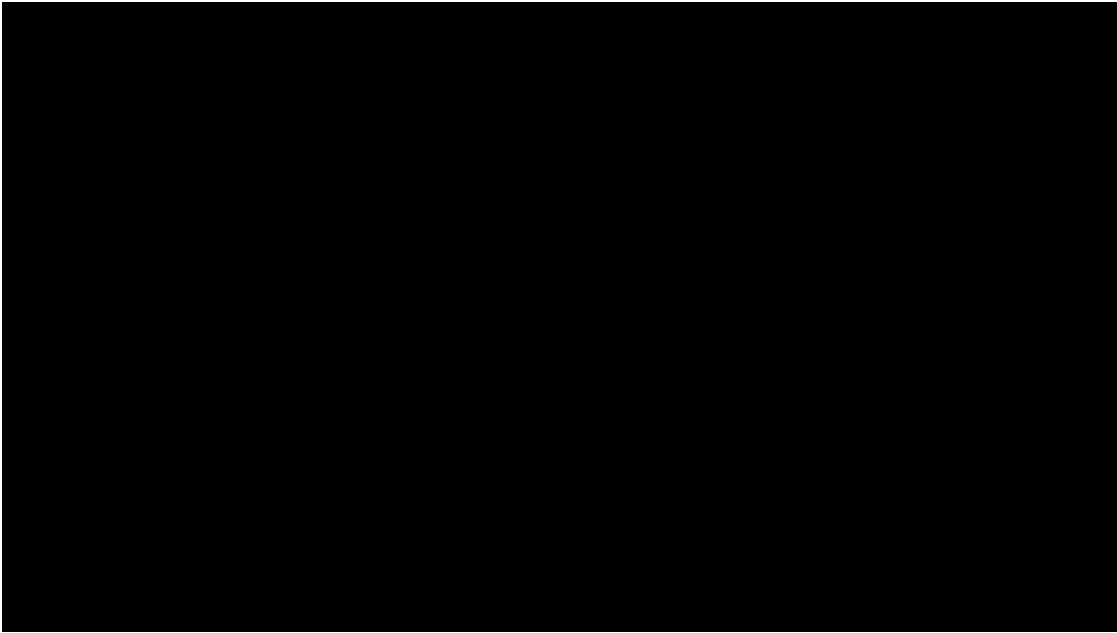
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

"Binary Tree | Set 1 (Introduction)" by [Shikha Singh](#) is licensed under [CC BY-SA 4.0](#)

This page titled 14.1: Binary Tree is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

14.2: Binary Tree Video

This is a [short video](#) talking about some of the binary tree stuff....



This page titled [14.2: Binary Tree Video](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

14.3: Binary Tree Properties

Binary Trees have certain properties, and some of them are calculated based on each tree. We will not go into the math that is depicted below, but I did want to present this, so that you are exposed to the concept of these properties.

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1} .

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

This can be proved by induction.

For root, $l = 1$, number of nodes = $2^{1-1} = 1$

Assume that maximum number of nodes on level l is 2^{l-1}

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.

Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + \dots + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.

In some books, height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \log_2(N+1) \rceil$?

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes $\lceil \log_2(N+1) \rceil - 1$

4) A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels

A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l , then below is true for number of leaves L .

$$L \leq 2^{l-1} \quad [\text{From Point 1}]$$

$$l = \lceil \log_2 L \rceil + 1$$

where l is the minimum number of levels.

5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

$$L = T + 1$$

Where L = Number of leaf nodes

T = Number of internal nodes with two children

See [Handshaking Lemma and Tree](#) for proof.

"Binary Tree | Set 2 (Properties)" by [Arjun Ashok](#) is licensed under [CC BY-SA 4.0](#)

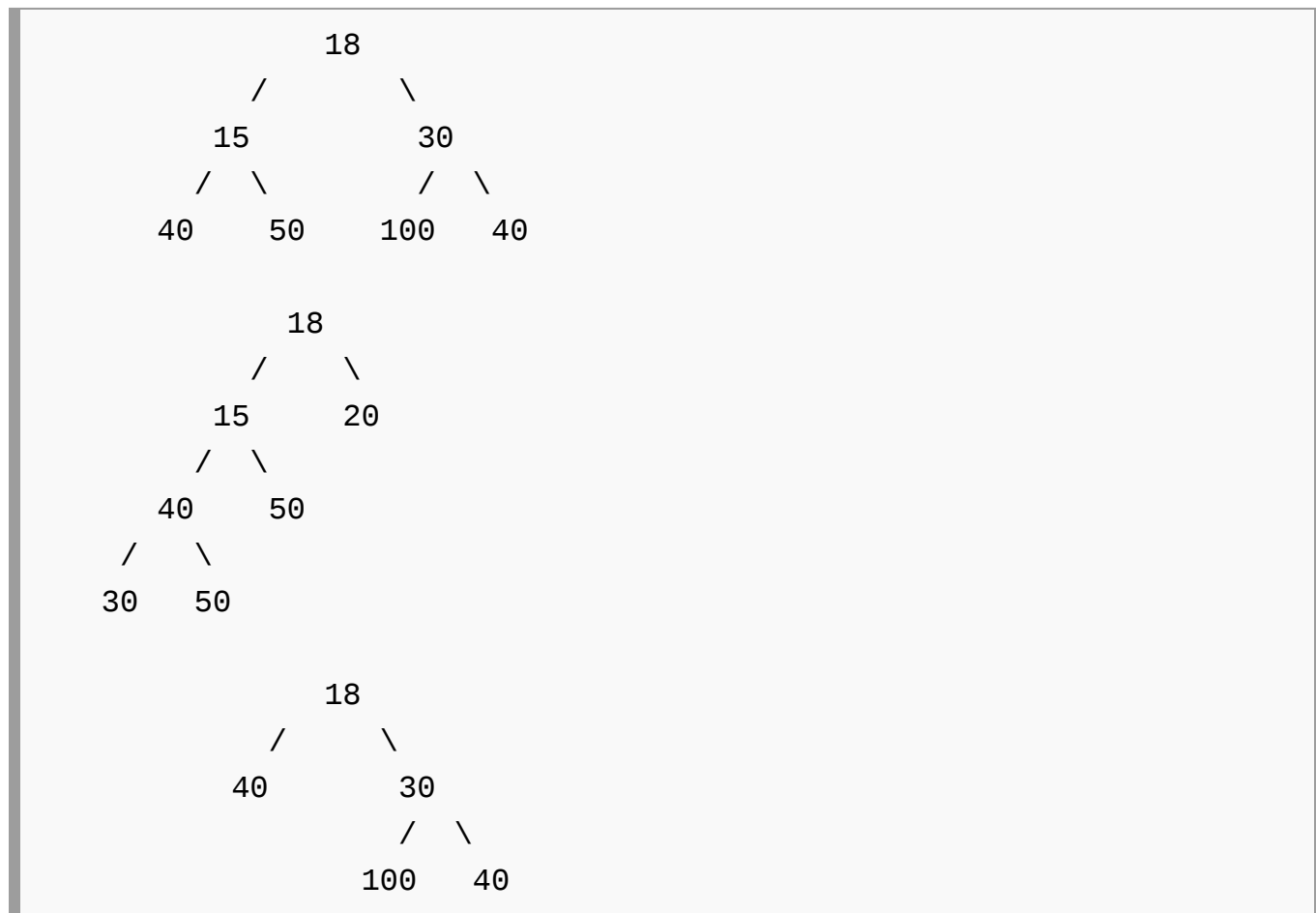
This page titled [14.3: Binary Tree Properties](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Patrick McClanahan](#).

14.4: Types of binary trees

We also need to discuss the various types of binary trees. Again - there is some math that is discussed below, and we will not delve into it. If you have questions about it feel free to ask.

Following are common types of Binary Trees.

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



In a Full Binary, number of leaf nodes is number of internal nodes plus 1

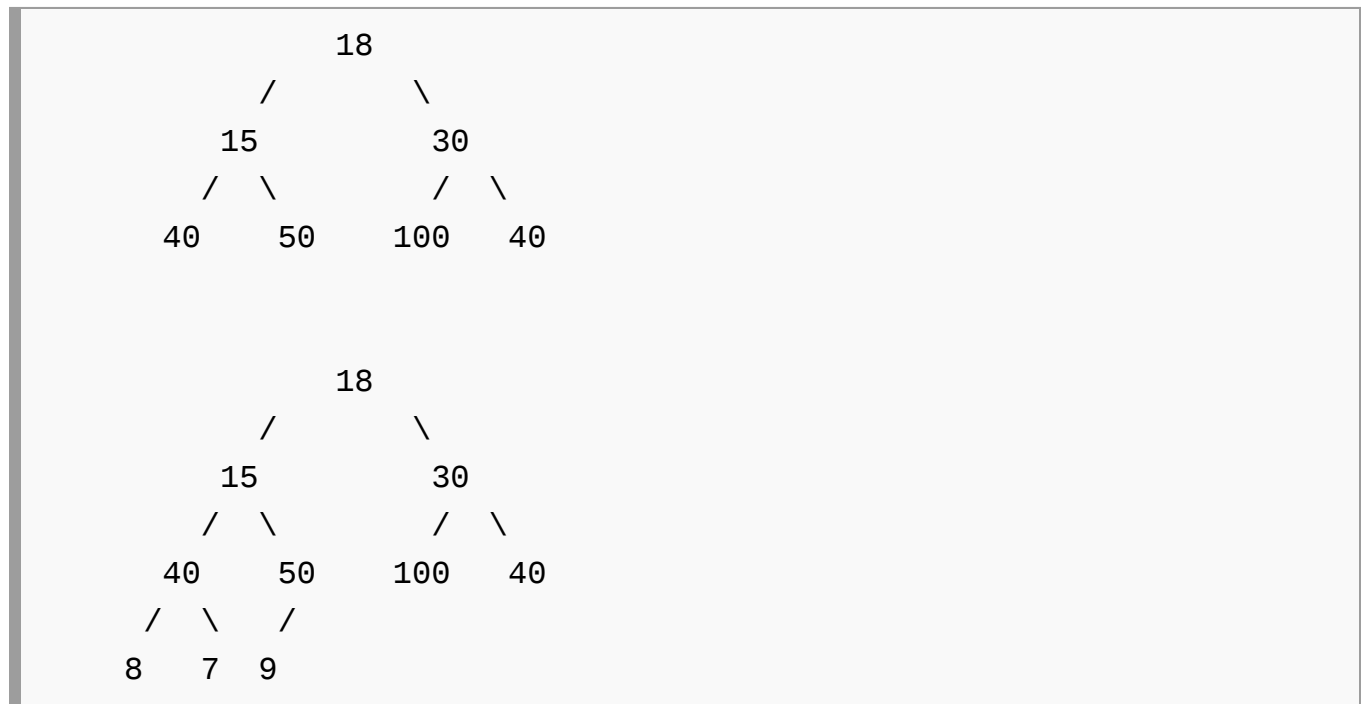
$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

See [Handshaking Lemma and Tree](#) for proof.

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

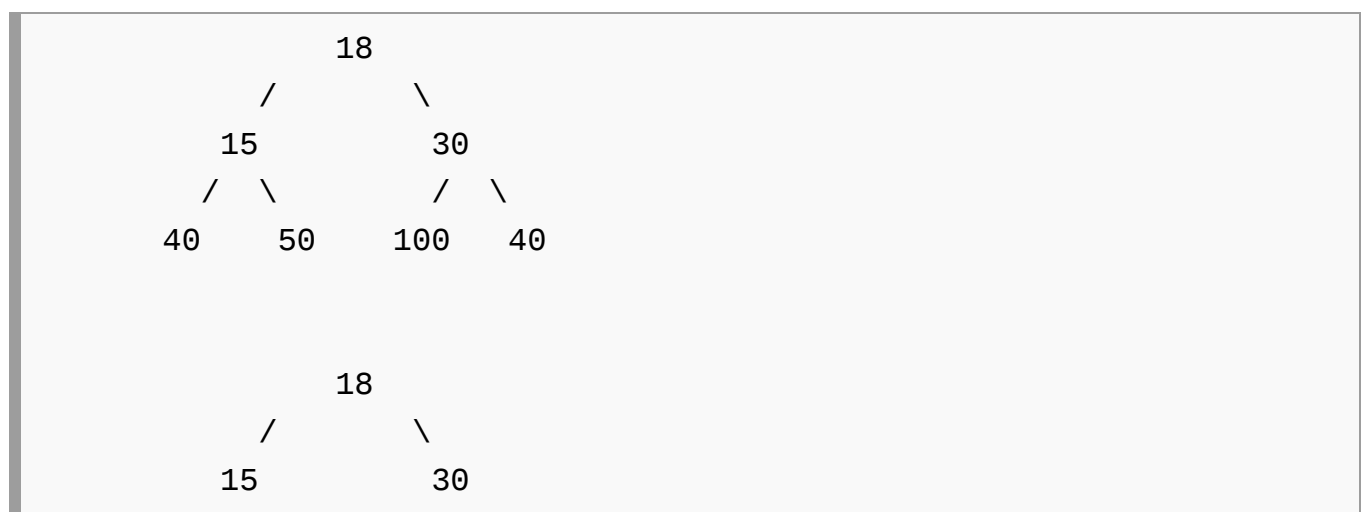
Following are examples of Complete Binary Trees



Practical example of Complete Binary Tree is [Binary Heap](#).

Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.

Following are examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.

Example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

A binary tree is balanced if the height of the tree is $O(\log n)$ where n is the number of nodes. For Example, AVL tree maintains $O(\log n)$ height by making sure that the difference between heights of left and right subtrees is at most 1. Red-Black trees maintain $O(\log n)$ height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide $O(\log n)$ time for search, insert and delete.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```
10
 /
20
 \
30
 \
40
```

Source:

https://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees

"Binary Tree | Set 3 (Types of Binary Tree)" by Shivam Kumar is licensed under CC BY-SA 4.0

This page titled 14.4: Types of binary trees is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

Index

D

dynamic memory allocation

[3.1: Dynamic memory allocation](#)

H

heap

[3.1: Dynamic memory allocation](#)

S

stack

[3.1: Dynamic memory allocation](#)

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: [C++ Data Structures](#)

Webpages: 98

All licenses found:

- [CC BY-SA 4.0](#): 77.6% (76 pages)
- [Undeclared](#): 22.4% (22 pages)

By Page

- [C++ Data Structures](#) - [CC BY-SA 4.0](#)
 - [Front Matter](#) - [Undeclared](#)
 - [TitlePage](#) - [Undeclared](#)
 - [InfoPage](#) - [Undeclared](#)
 - [Table of Contents](#) - [Undeclared](#)
 - [Licensing](#) - [Undeclared](#)
 - [ProgramPage](#) - [Undeclared](#)
 - [1: OOP Concepts](#) - [CC BY-SA 4.0](#)
 - [1.1: Object Oriented Programming](#) - [CC BY-SA 4.0](#)
 - [1.2: OOP Class](#) - [CC BY-SA 4.0](#)
 - [1.3: OOP Objects](#) - [CC BY-SA 4.0](#)
 - [1.4: OOP Encapsulation](#) - [CC BY-SA 4.0](#)
 - [1.5: OOP Abstraction](#) - [CC BY-SA 4.0](#)
 - [1.6: OOP Polymorphism](#) - [CC BY-SA 4.0](#)
 - [1.7: OOP Inheritance](#) - [CC BY-SA 4.0](#)
 - [1.8: Classes and Objects](#) - [CC BY-SA 4.0](#)
 - [1.9: Accessing Data Members](#) - [CC BY-SA 4.0](#)
 - [1.10: Public Access Specifiers](#) - [CC BY-SA 4.0](#)
 - [1.11: Private Access Specifier](#) - [CC BY-SA 4.0](#)
 - [1.12: Protected Access Specifier](#) - [CC BY-SA 4.0](#)
 - [2: C++ Pointers](#) - [CC BY-SA 4.0](#)
 - [2.1: Pointer Review](#) - [CC BY-SA 4.0](#)
 - [2.2: How to use a pointer](#) - [CC BY-SA 4.0](#)
 - [2.3: References and Pointers](#) - [CC BY-SA 4.0](#)
 - [2.4: Array Name as Pointers](#) - [CC BY-SA 4.0](#)
 - [2.5: Pointer Arithmetic](#) - [CC BY-SA 4.0](#)
 - [2.6: More Pointer](#) - [CC BY-SA 4.0](#)
 - [2.7: Pointers and Literals](#) - [CC BY-SA 4.0](#)
 - [2.8: Pointers to Pointers](#) - [CC BY-SA 4.0](#)
 - [2.9: Void Pointers](#) - [CC BY-SA 4.0](#)
 - [2.10: Invalid and NULL Pointers](#) - [CC BY-SA 4.0](#)
 - [3: Dynamic Memory Allocation and Arrays](#) - [CC BY-SA 4.0](#)
 - [3.1: Dynamic memory allocation](#) - [Undeclared](#)
 - [3.2: Arrays in C++](#) - [Undeclared](#)
 - [3.2.1: Arrays in C++ \(continued\)](#) - [Undeclared](#)
 - [3.3: Multidimensional Arrays](#) - [Undeclared](#)
 - [4: Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.1: The Concept of Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.2: Why and when to use inheritance?](#) - [CC BY-SA 4.0](#)
 - [4.3: Implementing inheritance in C++](#) - [CC BY-SA 4.0](#)
 - [4.4: Modes of Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.5: Types of Inheritance in C++](#) - [CC BY-SA 4.0](#)
 - [4.6: Types of Inheritance in C++](#) - [CC BY-SA 4.0](#)
 - [4.7: Types of Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.8: Types of Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.9: Types of Inheritance](#) - [CC BY-SA 4.0](#)
 - [4.10: Programming Assignment- Inheritance](#) - [CC BY-SA 4.0](#)
 - [5: Polymorphism](#) - [CC BY-SA 4.0](#)
 - [5.1: Polymorphism in C++](#) - [Undeclared](#)
 - [5.2: Virtual Functions in C++](#) - [Undeclared](#)
 - [5.3: More Virtual Functions](#) - [Undeclared](#)
 - [5.4: Difference between Inheritance and Polymorphism](#) - [Undeclared](#)
 - [6: Abstraction / Encapsulation](#) - [CC BY-SA 4.0](#)
 - [A.1: Difference between Abstraction and Encapsulation in C++](#) - [Undeclared](#)
 - [A.2: Abstraction](#) - [Undeclared](#)
 - [A.3: Encapsulation](#) - [Undeclared](#)
 - [7: Linked Lists](#) - [CC BY-SA 4.0](#)
 - [7.1: Linked Lists](#) - [CC BY-SA 4.0](#)
 - [7.2: Structures in C++](#) - [CC BY-SA 4.0](#)
 - [7.3: Structures in C++ \(cont'd\)](#) - [CC BY-SA 4.0](#)
 - [7.4: Structures in C++ \(cont'd\)](#) - [CC BY-SA 4.0](#)
 - [7.5: Linked Lists](#) - [CC BY-SA 4.0](#)
 - [7.6: Linked Lists vs Arrays](#) - [CC BY-SA 4.0](#)
 - [7.7: Create Linked List](#) - [CC BY-SA 4.0](#)
 - [7.8: Linked List Traversal](#) - [CC BY-SA 4.0](#)
 - [7.9: Linked List Insertion](#) - [CC BY-SA 4.0](#)
 - [7.10: Linked List Node Delete](#) - [CC BY-SA 4.0](#)

- 8: Recursion - *CC BY-SA 4.0*
 - 8.1: Recursion - *Undeclared*
- 9: Stack Data Structure - *CC BY-SA 4.0*
 - 9.1: Stacks - *CC BY-SA 4.0*
 - 9.2: Stack Implementation - *CC BY-SA 4.0*
 - 9.3: Stack code - *CC BY-SA 4.0*
- 10: Queues Data Structures - *CC BY-SA 4.0*
 - 10.1: Queues - *CC BY-SA 4.0*
 - 10.2: Looking at the queue code - *CC BY-SA 4.0*
 - 10.3: Looking at the queue code (continued) - *CC BY-SA 4.0*
 - 10.4: Looking at the code (continued) - *CC BY-SA 4.0*
 - 10.5: Look at the code (continued) - *CC BY-SA 4.0*
 - 10.6: Circular queue - *CC BY-SA 4.0*
- 11: Hashing - *CC BY-SA 4.0*
 - 11.1: Hashing - *CC BY-SA 4.0*
 - 11.2: Hashing continued - *CC BY-SA 4.0*
 - 11.3: Hashing continued - *CC BY-SA 4.0*
 - 11.4: Hashing- Separate Chaining - *CC BY-SA 4.0*
 - 11.5: Hashing- Open Addressing - *CC BY-SA 4.0*
 - 11.6: Open Addressing continued - *CC BY-SA 4.0*
- 12: Heap - *CC BY-SA 4.0*
- 13: Sorting - *CC BY-SA 4.0*
- 14: Binary Trees - *CC BY-SA 4.0*
 - 14.1: Binary Tree - *CC BY-SA 4.0*
 - 14.2: Binary Tree Video - *CC BY-SA 4.0*
 - 14.3: Binary Tree Properties - *CC BY-SA 4.0*
 - 14.4: Types of binary trees - *CC BY-SA 4.0*
- Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*