OOPS in C++

Table of Contents

Overview of C++

C++ is a versatile and powerful programming language that supports several programming paradigms, including procedural, object-oriented, and generic programming[1]. Developed by Bjarne Stroustrup in 1979, it is considered a middle-level language because it combines features from both high-level and low-level languages[2]. A significant aspect of C++ is that it's a superset of C, meaning that almost any valid C program is also a legal C++ program[3].

Object-Oriented Programming (OOP)

C++ fully supports the principles of object-oriented programming[4]. The document highlights the four "pillars" of object-oriented development[5]:

- Encapsulation
- Data Hiding
- Inheritance
- Polymorphism

Standard Libraries & Learning C++ :

Standard C++ consists of three important parts[1]:

- The core language, which provides fundamental building blocks like variables, data types, and literals[2].
- The C++ Standard Library, which offers a rich set of functions for tasks like manipulating files[3].
- The Standard Template Library (STL), which contains a rich set of methods for manipulating data structures[4].

The ANSI Standard :

The ANSI standard was created to ensure that C++ is a portable language[5]. This means that code written with one compiler, such as Microsoft's, should compile without errors using a different compiler on a different system, like a Mac or Unix[6].

Learning C++ :

The most crucial aspect of learning C++ is to follow the concepts. The goal of learning a programming language is to become a more effective programmer by improving skills in designing and implementing new systems. C++ is widely used for creating software and device drivers that rely on direct manipulation of hardware.

Usage of C++ :

C++ is a powerful language used to develop a variety of secure and robust applications1. Some of its uses include:

- Window Applications
- Device drivers
- Embedded firmware

C++ Program Example :

The page provides a basic "Hello C++ Programming" example:

```
#include <iostream>

using namespace std;

int main () {

   Cout <<" Hello C++ Programming";

   return 0;

}
```

Key Differences between C vs. C++ :

- **Programming Paradigm:** C follows a procedural programming style, while C++ supports both procedural and object-oriented programming.
- **Data Security:** Data is considered less secure in C. In contrast, C++ allows the use of modifiers for class members to control accessibility, making data more secure.
- **Approach:** C uses a top-down approach, whereas C++ uses a bottom-up approach.
- **Variables:** C does not support reference variables, but C++ does.
- **Input/Output:** C primarily uses scanf() and printf() for input and output operations, while C++ uses cin and cout.
- **Features:** C does not support inheritance or namespaces. C++, however, supports both inheritance and namespaces.

Turbo C++ Download & Installation :

The document states that while many C++ compilers are available, it will use Turbo C++ for its examples, as it works for both C and C++1.

To install the Turbo C++ software, you need to follow these steps2:

1. Download Turbo C++.
2. Create a turboc directory inside your C drive and extract the tc3.zip file into it.
3. Double-click on the install.exe file to begin the installation.
4. To write and run a program, click on the TC application file located inside the C:\TC\BIN directory.

C++ Basic Input/Output and Header Files :

C++ I/O operations are handled using the concept of **streams**. A stream is a sequence of bytes, representing a flow of data, that makes performance fast.

- An **output operation** occurs when bytes flow from the main memory to a device like a printer or display screen.
- An **input operation** occurs when bytes flow from a device (e.g., a keyboard, printer, or network connection) into the main memory.

I/O Library Header Files

The document lists several header files that provide functionalities for I/O operations:

- **<iostream>**: This header file is used to define the cout, cin, and cerr objects, which correspond to the standard output, standard input, and standard error streams, respectively.
- **<iomanip>**: This header file declares services that are useful for formatted I/O, such as setprecision.
- **<fstream>**: This header file is used to declare services for user-controlled file processing.

Standard Input Stream (cin) :

- The cin is a predefined object of the istream class.
- It is connected with the standard input device, which is usually a console.
- The cin object is used in conjunction with the stream extraction operator (>>) to read input from a console.

```
#include <iostream>

using namespace std;

int main () {

   int age;

   cout << "Enter your age:";

   cin >> age;

   cout << "Your age is " << age << endl;

}
```

Standard Output Stream (cout) :

- The cout is a predefined object of the ostream class.
- It is connected to the standard output device, which is usually a display screen.
- The cout object is used with the stream insertion operator (<<) to display output on the console.

```
#include <iostream>

int main() {

   char ary[] = "Welcome to c++ tutorial";

   cout << "Value of ary is: " << ary << endl;

}
```

Standard End Line (endl) :

- The endl is a predefined object of the ostream class.
- It is used to insert a new line character and flush the stream.

```
#include <iostream>

using namespace std;

int main(){

   Cout << "C++ Tutorial";

   Cout << "Javatpoint" << endl;

   Cout << "End of line " << endl;

}
```

C++ Variables:

A variable is a name given to a memory location, and its value can be changed. It acts as a symbolic representation of a memory location for easy identification.

Syntax and Rules:

To declare a variable, you specify its data type followed by the variable name.

- int x;
- float y;
- char z;

In these examples, x, y, and z are variables, while int, float, and char are their data types.

A variable name can include alphabets, digits, and underscores, but it cannot contain white spaces.

C++ Data Types :

A data type defines the kind of data a variable can store, such as an integer, string, or character.

Data Type Categories in C++:

- **Basic Data Types:** These include integer-based and floating-point-based types like int, char, float, and double. C++ supports both signed and unsigned literals. The memory size of these types can vary depending on the operating system (32-bit or 64-bit).
- **Derived Data Types:** These are built from basic types and include array and pointer.
- **Enumeration:** Defined using the enum keyword.
- **User-defined Data Types:** This category includes structure.

Basic Data Types and Memory Sizes :

This page presents a table detailing the memory size and value range for various basic data types.

Data Type	Memory Size	Range
char	1 byte	-128 to 127
Signed char	1 byte	-128 to 127
Unsigned char	1 byte	0 to 127
Short	2 bytes	-32,768 to 32767
signed short	2 bytes	-32,768 to 32767
unsigned short	2 bytes	0 to 32,767
int	2 bytes	-32,767 to 32767
signed int	2 bytes	-32,767 to 32767
unsigned int	2 bytes	0 to 32,767
short int	2 bytes	-32,767 to 32767

Data Type	Memory Size	Range
signed short int	2 bytes	-32,767 to 32767
unsigned short int	2 bytes	0 to 32,767
Float	4 bytes	
Double	8 bytes	

C++ Keywords :

A keyword is a reserved word that cannot be used as a variable or constant name. The page lists 32 keywords found in both C and C++:

- Auto
- Break
- Case
- Char
- Const
- double
- else
- enum
- extern
- float
- int
- long
- register
- return
- short
- struct
- switch
- union
- unsigned.

C++ Operators :

An operator is a symbol used to perform operations. The document categorizes different types of operators:

- Arithmetic operator
- Relational operator

- Logical operator
- Bitwise operator
- Assignment operator
- Unary operator
- Conditional operator
- Misc operator

C++ Operators :

This page lists the symbols for various operator types.

- **Arithmetic:** +, -, *, /, %.
- **Relational:** ==, !=, >, <, >=, <=.
- **Logical:** &&, ||, !.
- **Bitwise:** &, |, ^, ~, <<, >>.
- **Assignment:** =, +=, -=, *=, /=, %=.
- **Unary:** ++, --.
- **Conditional:** ?:.

C++ Identifiers :

Identifiers are names used in a program to refer to variables, functions, arrays, and other user-defined elements. They are a fundamental requirement of any programming language. Identifiers represent essential program elements such as constants, variables, functions, labels, and user-defined data types.

Identifiers Example :

This page provides a simple code example to demonstrate the concept of identifiers. In the code, a and A are used as distinct identifiers.

```
#include <iostream>
using namespace std;
int main()
{
   int a;
   int A;
   cout << "Enter the values of 'a' & 'A': ";
   cin >> a;
   cin >> A;
```

```
    cout << "The values that you have entered
are: " << a << ", " << A;

    return 0;

}
```

Page 22: Identifiers vs. Keywords :

This page outlines the differences between identifiers and keywords in a table.

Identifiers	Keywords
Names defined by the programmer[11].	Reserved words with a compiler-known meaning[12].
Used to identify variables and other entities[13].	Used to specify the type of an entity[14].
Can contain letters, digits, and underscores[15].	Contain only letters[16].
Can use both uppercase and lowercase letters[17].	Use only lowercase letters[18].
Can be classified as internal or external[19].	Cannot be further classified[20].
Examples: test, result, sum.	Examples: for, if, else, break

C++ Expressions :

A C++ expression is a combination of operators, constants, and variables arranged according to the language's rules. An expression can also include function calls that return values.

Types of Expressions:

- Constant expression
- Integral expression
- Float expression
- Pointer expression
- Relational expression
- Logical expression
- Bitwise expression
- Special assignment expression

C++ Control Statements (If-Else) :

C++ control statements are used to test a condition and execute code based on the result.

Types of if statements:

- If statement
- If-else statement
- Nested-if statement

Syntax of if statement:

```
if (condition)
{
   // code to be executed
}
```

A code example is provided to check if a number is even.

```
#include <iostream>
using namespace std;
int main()
{
   int num = 10;
   if (num % 2 == 0)
   {
      cout << "It is even number";
   }
   return 0;
}
```

The output of this code is "It is even number"[37].

if Statement Flowchart and Example:

This page illustrates the flow of an if statement. The program checks if a number is even and prints a message if the condition is true.

Code:

```
#include <iostream>
using namespace std;
int main()
{
   int num = 10;
   if (num % 2 == 0)
   {
      cout << "It is even number";
   }
   return 0;
}
```

Output:

```
It is even number
```

C++ if-else Statement:

This page introduces the if-else statement, which provides an alternative code path for when a condition is false.

Syntax of if-else statement:

```
C++
if (condition)
{
   // code if condition is true.
}
else
{
   // code if condition is false.
}
```

if-else Example:

This page provides an example of an if-else statement to determine if a number is even or odd.

Code:

```
#include <iostream>

using namespace std;

int main()

{

   int num = 11;

   if (num % 2 == 0)

   {

      cout << "It is even number";

   }

   else

   {

      cout << "It is odd number";

   }

   return 0;

}
```

Output:

```
It is odd number.
```

C++ if-else-if Ladder :

This page describes the if-else-if ladder, which is used to check multiple conditions sequentially.

Syntax:

```
if (condition1)

{

   // code

}

else if (condition2)

{

   // code

}

else if (condition3)

{

   // code

}

else
```

```
{
   // code
}
```

Example :

This page provides a complete code example for a grade-checking program using an if-else-if ladder. The program prompts the user to enter a number to check their grade and then evaluates it against a series of conditions.

Code Snippet:

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number to check Grade: ";
   cin >> num;
   if (num < 0 || num > 100) {
      cout << "wrong number";
   } else if (num >= 0 && num < 50) {
      cout << "fail";
   } else if (num >= 50 && num < 60) {
      cout << "D Grade";
   }
   // ... (continued on next page)
}
// ... (continued from previous page)
else if (num >= 60 && num < 70) {
   cout << "C Grade";
} else if (num >= 70 && num < 80) {
   cout << "B Grade";
} else if (num >= 80 && num < 90) {
   cout << "A Grade";
}
// ... (end of main function)
```

Output:

The page shows two example outputs:

- For an input of 66, the output is C Grade.
- For an input of -2, the output is wrong number.

C++ switch Statement :

The switch statement is an alternative to long if-else-if ladders, allowing a program to execute different code blocks based on the value of a single expression.

Syntax:

```
switch (expression) {
   case Value1:
      // Statements
      break;
   case Value2:
      // Statements
      break;
   default:
      // Default statements
      break;
}
```

Example :

Code Snippet:

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number to check Grade: ";
   cin >> num;
   switch (num) {
      case 10:
         cout << "It is 10";
         break;
```

```
    case 20:
      cout << "It is 20";
      break;
    case 30:
      cout << "It is 30";
      break;
    default:
      cout << "Not 10, 20, 30";
      break;
  }
  return 0;
}
```

Output:

For an input of 10, the output is It is 10.

C++ for Loop :

The for loop is used to iterate a part of a program a specific number of times.

Syntax:

```
for (initialization; condition;
increment/decrement) {
  // code to be executed
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
  for (int i = 1; i <= 10; i++) {
    cout << i << "\n";
  }
  return 0;
}
```

Output:

The program prints the numbers from 1 to 10, each on a new line.

C++ while Loop:

The while loop is used to repeatedly execute a block of code as long as a specified condition remains true. It is recommended for situations where the number of iterations is not fixed.

Syntax:

```
while (condition) {
    // code to be executed
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    while (i <= 10) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

C++ do-while Loop :

The do-while loop is similar to a while loop but guarantees that the code block is executed at least once before the condition is checked.

Syntax:

```
do {
    // code to be executed
} while (condition);
```

Example :

This page provides an example of a do-while loop. The loop prints numbers from 1 to 10, ensuring that the code inside the loop runs at least once.

```
#include <iostream>

using namespace std;

int main() {

   int i = 1;

   do {

      cout << i << "\n";

      i++;

   } while (i <= 10);

   return 0;

}
```

Output:

The program prints the numbers from 1 to 10, each on a new line.

C++ break Statement :

The break statement is a jump statement used to exit a loop or switch statement, breaking the current flow of the program.

Syntax:

break;

Example:

The loop is designed to print numbers from 1 to 10, but the if (i == 5) condition causes the break statement to execute, which terminates the loop prematurely.

```
#include <iostream>

using namespace std;

int main()

{

   for(int i = 1; i <= 10; i++)

   {

      if (i == 5)

         break;

      cout << i << "\n";

   }

   return 0;

}
```

```
Output:

1

2

3

4
```

C++ continue Statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.

Syntax:

continue;

Example:

This example demonstrates how continue is used to skip printing the number 5. The loop runs from 1 to 10, but when i is equal to 5, the continue statement is triggered, and the cout line is skipped for that iteration.

```
#include <iostream>

using namespace std;

int main()

{

   for (int i = 1; i <= 10; i++)

   {

      if (i == 5)

      {

         continue;

      }

      cout << i << "\n";

   }

   return 0;

}
```

Output:

```
1 2 3 4 6 7 8 9 10
```

C++ Comments:

C++ comments are statements that are not executed by the compiler. They are used to provide explanations of the code.

Types of Comments:

1. **Single-Line Comment:** Starts with //.
2. **Multi-line Comment:** Starts with /* and ends with */.

Example (Single-Line Comment):

```
#include <iostream>
using namespace std;
int main()
{
  int x = 11; // x is a variable.
  cout << x << "\n";
  return 0;
}
```

Output:

```
11
```

Multi-line Comment:

This page provides an example of a multi-line comment.

Example (Multi-line Comment):

```
#include <iostream>
using namespace std;
int main()
{
  /* declare and
  print variable in c++. */
  int x = 35;
  cout << x << "\n";
  return 0;
}
```

Output:

```
35
```

A note on this page clarifies that the multi-line comment is used to comment out multiple lines of code and is enclosed by /* and */.

C++ Arrays :

An array in C++ is a group of elements of the same type that are stored in contiguous memory locations. In C++, **std::array** is a container that encapsulates fixed-size arrays. Array indexes begin at 0.

Advantages of C++ Arrays:

- Code optimization
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data

Disadvantages of C++ Arrays:

- **Fixed size:** Arrays have a fixed size, which means you cannot change their size after creation.

C++ Array Types:

- Single-dimensional array.
- Multi-dimensional array.

Single-Dimensional Array Example:

This code snippet shows how to declare and traverse a single-dimensional array. The program initializes an integer array named arr and then prints its elements using a for loop.

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


int main () {
   int arr[5] = {10, 0, 20, 0, 30};
   for (int i = 0; i < 5; i++) {
       cout << arr[i] << " ";
   }
}
```

Output:

```
10 0 20 0 30.
```

Passing an Array to a Function :

This page explains how to pass an array to a function in C++ to reuse array logic. You only need to provide the array's name as an argument.

Example:

The following example defines a function print Array that accepts an array and then calls it twice with two different arrays.

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

void printArray(int arr[5]);

void printArray(int arr[5]) {
   cout << "printing array Elements:" << endl;
   for (int i = 0; i < 5; i++) {
      cout << arr[i] << "\n";
   }
}

int main() {
   int arr1[5] = {10, 20, 30, 40, 50};
   int arr2[5] = {5, 15, 25, 35, 45};
   printArray(arr1);
   printArray(arr2);
}
```

Output :

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Arrays:

Multidimensional arrays are also known as rectangular arrays and can be two- or three-dimensional5. Data in these arrays is stored in a tabular format6.

Example:

This code declares a 3x3 two-dimensional array named test.

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

int main() {
   int test[3][3];
   test[0][0] = 5;
   test[0][1] = 10;
   test[1][1] = 15;
   test[1][2] = 20;
   test[2][0] = 30;
   test[2][2] = 10;
}
for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 3; ++j) {
      cout << test[i][j] << " ";
   }
   cout << "\n";
}
return 0;
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Functions :

A function in C++ is a block of code that performs a specific task and can be called multiple times[8888].

Advantages of Functions:

- **Code Reusability:** Functions allow you to reuse code logic by calling the same function multiple times, eliminating the need to write the same code again.
- **Code Optimization:** Functions optimize code by reducing the amount of code you need to write and allowing you to use the same logic multiple times after writing it just once.

Types of Functions and Syntax :

This page outlines the two main types of functions in C++:

- **Library Functions:** These are predefined functions available in C++ header files, such as ceil(x) and cos(x).
- **User-Defined Functions:** These are created by the programmer and can be used multiple times, helping to reduce the complexity of a large program.

Syntax for Creating Functions:

```
type function_name (parameters) {
   // code to be executed
}
```

Function Example:

The following code defines a function Func() to demonstrate the use of static and local variables.

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


void Func() {
   void Func() {
   static int i = 0;
   int j = 0;
   i++;
   j++;
   cout << "i=" << i << " &j=" << j << endl;
}
int main() {
   Func();
   Func();
   Func();
}
```

Output:

```
i=1 & j=1

i=2 & j=1

i=3 & j=1
```

The output demonstrates that i (a static variable) retains its value across function calls, while j (a local variable) is re-initialized each time.

Call by Value & Call by Reference :

This page introduces two methods for passing data to functions: call by value and call by reference.

- In **call by value**, a copy of the original value is passed, so the original value is not modified[14].
- In **call by reference**, the address of the value is passed, so the original value is modified[15].

Call by Value:

The page notes that with call by value, the original value of a variable is not changed by the function.

Examples:

This page provides code examples for both "call by value" and "call by reference."

Call by Value Example:

This program demonstrates that the change() function receives a copy of data, so the original value in main() remains unchanged.

```
#include <iostream>

using namespace std;


void change(int data) {

   data = 5;

}


int main() {

   int data = 3;

   change(data);

   cout << "Value of data is " << data << endl;

   return 0;

}
```

Output:

```
Value of data is 3.
```

Call by Reference Example (Cont.) :

This page provides the code for the call-by-reference example. A swap() function is used to exchange the values of two variables by passing their memory addresses (pointers).

Example :

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

void swap(int *x, int *y) {
    int swap;
    swap = *x;
    *x = *y;
    *y = swap;
}

int main() {
    int x = 500;
    int y = 100;
    swap(&x, &y);
    cout << "Value of x is: " << x << endl;
    cout << "Value of y is: " << y << endl;
    return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Pointers vs. References :

The main difference between a pointer and a reference is how they handle memory and assignment. A **pointer** is a variable that holds the memory address of another variable. It can be reassigned to point to a different variable, and it can be assigned a NULL value. Pointers have their own memory address and size on the stack.

In contrast, a **reference** is an alias, or another name, for an already existing variable. It cannot be reassigned after its initial declaration. A reference shares the same memory address as the variable it references and cannot be assigned NULL.

Pointers	References
A pointer is a variable that holds the memory address of another variable.	A reference is an alias, or another name, for an existing variable.
A pointer can be reassigned.	A reference cannot be reassigned.
A pointer has its own memory address and size on the stack.	A reference shares the same memory address but also takes up some space on the stack.
A pointer can be assigned	NULL directly.
A reference cannot be assigned	NULL directly
int a=10; int *p=&a; or int *p; p=&a;	int a=10; int &p=a; (correct) but int &p; p=a; (incorrect).

C++ Recursion :

Recursion is when a function calls itself A function that does this is called a recursive function.

Example:

This code calculates the factorial of a number using a recursive function.

```
#include <iostream>

using namespace std;

int factorial(int);

int main()

{

   int fact, value;

   cout << "Enter any number: ";

   cin >> value;

   fact = factorial(value);

   cout << "Factorial of a number is: " <<
fact << endl;

   return 0;

}

int factorial(int n)

{
```

```
{
   if (n < 0)
      return (1);
   else
      return (n * factorial(n - 1));
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

The diagram visually represents the recursive calls for factorial(5), where each call multiplies the number by the factorial of the number minus one, until it reaches factorial(0).

C++ Storage Classes :

Storage classes define the lifetime and visibility of a variable or function in a C++ program[11]. There are five types of storage classes.

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	Class	Mutable	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

C++ Pointers :

A pointer is a variable in C++ that holds the memory address of another variable. It is also known as a locator or an indicator.

Advantages of Pointers:

- They reduce code and improve performance.
- They can be used to return multiple values from a function.

- They allow access to any memory location in the computer's memory.

Usage of Pointers:

Pointers are used for dynamic memory allocation, arrays, functions, and structures17.

Pointer Symbols and Declaration :

This page defines the two main symbols used with pointers.

Symbol	Name	Description
&	Address operator	Determines the address of a variable[18].
*	Indirection operator	Accesses the value at an address[19].

Declaring a Pointer:

To declare a pointer, use the asterisk (*) symbol followed by the variable name.

- int *p; (pointer to an integer)
- char *c; (pointer to a character)

Pointer Example :

This page provides a complete code example that demonstrates how to declare a pointer, assign the address of a variable to it, and then print the addresses and the value.

```
#include <iostream>

using namespace std;

int main() {

   int number = 30;

   int *p;

   p = &number;

   cout << "Address of number variable is: " << &number <<
endl;

   cout << "Address of p variable is: " << p << endl;

   cout << "value of p variable is: " << *p << endl;

   return 0;

}
```

Output:

The output shows that the address of the number variable is the same as the value of the pointer p, and *p dereferences the pointer to show the value of number.

C++ Array of Pointers :

This page discusses the close relationship between arrays and pointers. The name of an array acts as a pointer to its first element.

Example:

This program demonstrates that both a pointer (ptr) and an array name (marks) can point to the same memory location, accessing the same value.

```
#include <iostream>
using namespace std;
int main() {
   int *ptr;
   int marks[10];
   cout << "Enter the elements of an array." <<
endl;
   for (int i = 0; i < 10; i++) {
      cin >> marks[i];
   }
   ptr = marks;
   std::cout << "Value of *ptr is: " << *ptr <<
std::endl;
   std::cout << "value of *marks is: " << *marks <<
std::endl;
}
```

Output:

```
Enter the elements of an array:
1
2
3
4
5
6
7
8
9
The value of *ptr is: 1
The value of *marks is: 1
```

The output confirms that both *ptr and *marks access the same value (the first element of the array), proving that the array name itself stores the address of its first element.

C++ Void Pointer :

A void pointer is a generic pointer that can hold the address of any data type, but it is not associated with a specific type.

Syntax:

void *ptr;

Example:

This code shows how a void pointer can store the address of an int variable.

```
#include <iostream>

using namespace std;

int main()

{

   void *ptr;

   int a = 9;

   ptr = &a;

   std::cout << a << std::endl;

   std::cout << ptr << std::endl;

   return 0;

}
```

Output:

The output shows the address of the variable a is the same as the value of the void pointer ptr.

```
9

0x7ffcf1da5004
```

Void Pointer Purpose:

This page explains the purpose of a void pointer. A void pointer can hold the address of any data type but is not associated with any specific data type. You cannot assign the address of a variable to a variable of a different data type in C++.

Syntax of Void Pointer:

The page gives an example of a void pointer assignment that would not be possible with other pointer types:

```
int *ptr;

float a = 10.2;

ptr = &a; // This is not possible in C++
```

The example explains that the integer pointer ptr cannot hold the address of the float variable a because they are of different data types.

Function Pointers:

This page introduces the concept of function pointers, which are used to point to functions[14]. They are primarily useful for event-driven applications and callbacks.

What is the address of functions?

The page includes a table and diagram to illustrate how source code is compiled into machine code with specific memory addresses for different parts of the program, such as the heap, stack, and code sections. The address of a function is located in the code section.

Syntax of Declaration:

int (*funcPtr)(int, int);

This declares a function pointer named funcPtr that can point to a function which takes two integers as arguments and returns an integer.

Function Pointer Example:

This page provides a code example of how to use a function pointer to call a function. The program defines an add() function and then assigns its address to a function pointer funcPtr, which is then used to call the function.

```
#include <iostream>

using namespace std;


int add(int a, int b)

{

   return a + b;

}

int main()

{

   int (*funcPtr)(int, int);

   funcPtr = add;

   int sum = funcPtr(5, 5);

   std::cout << "Value of sum is: " << sum << std::endl;

   return 0;

}
```

```
Output:

Value of sum is: 10.
```

C++ Object-Oriented Class Concepts :

This page introduces the core concepts of object-oriented programming (OOP) in C++. OOP is a paradigm that designs a program using classes and objects[20].

Key OOP Concepts:

- **Object:** A real-world entity, such as a chair or car, that has a state and behavior at runtime[21].
- **Class:** A group of similar objects; it serves as a template from which objects are created[22].
- **Inheritance:** A process where one object acquires the properties and behavior of its parent object[23].
- **Polymorphism:** The ability of an object to take on many forms[24].
- **Abstraction:** The process of hiding internal details and showing only functionality[25].
- **Encapsulation:** The bundling of data and methods that operate on the data into a single unit.

Advantages of OOP and Object/Class Definitions :

This page lists the advantages of using OOP over procedural programming and provides more detailed definitions of C++ objects and classes.

Advantages of OOP:

1. **Easier Maintenance:** OOP makes program development and maintenance easier, especially as a project's size grows.
2. **Data Hiding:** OOP provides data hiding, which is not easily achieved in procedural programming where global data can be accessed from anywhere.
3. **Real-World Simulation:** OOP allows for more effective simulation of the real world and provides solutions to real-world problems.

C++ Object:

- A real-world entity (e.g., chair, car, pen).
- A runtime entity that has a state and behavior.
- An instance of a class.

C++ Class:

- A group of similar objects.
- A template from which objects are created.

C++ Object and Class Example:

The program defines a Student class with id and name as public members. It then creates an object s1 of this class, assigns values to its members, and prints them to the console.

```
#include <iostream>

#include <string> // Added based on context of string usage

using namespace std;


class Student {

public:

   int id;

   string name;

};


int main() {

   Student s1;

   s1.id = 201;

   s1.name = "Sonoo Jaiswal";

   cout << s1.id << endl;

   cout << s1.name << endl;

   return 0;

}
```

Output:

```
201

Sonoo Jaiswal
```

C++ Constructors :

A constructor is a special method that is automatically invoked when an object is created[34]. Its primary purpose is to initialize the data members of the new object.

Types of Constructors:

- **Default Constructor:** Has no arguments and is automatically called if no constructor is defined by the programmer.
- **Parameterized Constructor:** Has one or more parameters and must be written by the programmer.

Default Constructor	Parameterized Constructor
Has no parameters[38].	Has one or more parameters[39].
Called automatically if no constructor is written[40].	Must be explicitly written by the programmer[41].

Default Constructor Example :

This page provides a code example demonstrating the use of a default constructor. The Employee() constructor is called automatically when objects e1 and e2 are created, printing a message to the console.

Example:

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

class Employee {
public:
   Employee() {
      cout << "Default Constructor Invoked" << endl;
   }
};

int main(void) {
   Employee e1;
   Employee e2;
   return 0;
}
```

Output:

```
Default Constructor invoked.
Default Constructor invoked.
```

Parameterized Constructor Example :

This page begins a code example for a parameterized constructor. The constructor is defined to take an integer, a string, and a float to initialize the employee's ID, name, and salary.

Example:

```
#include <iostream>
#include <string> // Added based on context of string usage
using namespace std;

class Employee {
public:
int id;
```

```
    float salary;

    string name;


    Employee(int i, string n, float s) {

        id = i;

        name = n;

        salary = s;

    }


    void display() {

        cout << id << " " << name << " " << salary << endl;

    }

};


int main(void) {

    Employee e1 = Employee(101, "Sonoo Jaiswal",
890000);

    Employee e2 = Employee(102, "Nakul", 59000);

    e1.display();

    e2.display();

    return 0;

}
```

Output:

```
101 Sonoo Jaiswal 890000

102 Nakul 59000
```

The output shows that two Employee objects are created and their details are displayed, confirming that the parameterized constructor successfully initialized their respective member variables.

C++ Copy Constructor:

A copy constructor is an overloaded constructor used to declare and initialize a new object as a copy of an existing object.

Types of Copy Constructors:

- **Default Copy Constructor:** Defined by the compiler if the user does not define their own.
- **User-Defined Copy Constructor:** Defined by the programmer.

Syntax of a Copy Constructor:

class_name(const class_name &old_object);

Copy Constructor Example :

This page provides a simple example of a copy constructor. The program creates an object a1 with a value of 20 and then creates a second object a2 as a copy of a1. The copy constructor ensures that a2 also holds the value 20.

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

class A {
public:
   int x;
   A(int a) { // Constructor
      x = a;
   }
   A(A &i) { // Copy Constructor
      x = i.x;
   }
};

int main() {
   A a1(20);
   A a2(a1);
   cout << a2.x;
   return 0;
}
```

Output:

```
20
```

Shallow vs. Deep Copy:

This page explains the difference between shallow and deep copies, which are two ways an object can be copied.

Shallow Copy:

- A shallow copy creates a new object by simply copying the data of all member variables as they are.
- The default copy constructor in C++ produces a shallow copy.

Deep Copy:

- A deep copy dynamically allocates new memory for the copy and then copies the actual value.
- Both the source and the copy have distinct memory locations.
- A deep copy requires a user-defined copy constructor.

Shallow Copy Example:

This page begins a code example for a shallow copy.

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


class Demo {
    int a;
    int b;
    int *p;
public:
    Demo() {
        p = new int;
    }
    void setdata(int x, int y, int z) {
        a = x;
        b = y;
        *p = z;
    }
    void showdata() {
        std::cout << "Value of a is " << a << std::endl;
        std::cout << "Value of b is: " << b << std::endl;
        std::cout << "Value of *p is: " << *p << std::endl;
    }
};
int main() {
    Demo d1;
    d1.setdata(4, 5, 7);

    Demo d2 = d1;

    d2.showdata();

    return 0;

}
```

Output:

```
Value of a is: 4

Value of b is: 5

Value of *p is: 7
```

The example shows a Demo class with a pointer member p. A shallow copy of an object of this class will result in both the original and the new object pointing to the same memory location for p. The output shows that the copied object d2 has the same values as the original object d1, including the value pointed to by p, because it's a shallow copy.

Deep Copy Example:

This page begins a code example for a deep copy, showing the definition of a class with a user-defined copy constructor.

```
#include <iostream>

using namespace std;


class Demo {

public:

    int a;

    int b;

    int *p;

    Demo() { // Default constructor

        p = new int;

    }

    Demo(Demo &d) { // User-defined copy
constructor

        a = d.a;

        b = d.b;

        p = new int;

        *p = *(d.p);

    }

};
```

```
void showdata() {
    std::cout << "Value of a is: " << a << std::endl;
    std::cout << "Value of b is: " << b << std::endl;
    std::cout << "Value of *p is: " << *p << std::endl;
}

int main() {
    Demo d1;
    d1.setdata(4, 5, 7);
    Demo d2 = d1;
    d2.showdata();
    return 0;
}
```

Output:

```
Value of a is 4
Value of b is: 5
Value of *p is: 7
```

The output confirms that the values are correctly copied, and because it is a deep copy, changes to d1 would not affect d2.

C++ Destructor :

A destructor is a special method that works in the opposite way of a constructor. It is invoked automatically when an object is destroyed.

Key characteristics of a destructor:

- It is defined only once per class.
- It must have the same name as the class, preceded by a tilde (~).
- It does not take any parameters.
- No modifiers can be applied to it.

Key Difference between Constructor and Destructor:

A constructor initializes an object when it's created, while a destructor performs cleanup tasks when the object is destroyed. A class can have multiple constructors but only one destructor, and it is identified by a tilde (~) before the class name. The constructor is called automatically when an object is instantiated, and the destructor is called automatically when it goes out of scope.

Example of Constructor and Destructor:

This page begins a code example that includes both a constructor and a destructor to show when each is invoked.

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

class Employee {
public:
   Employee() {
      cout << "Constructor Invoked" << endl;
   }
   ~Employee() {
   cout << "Destructor Invoked" << endl;
   }
};
int main(void) {
   Employee e1;
   Employee e2;
   return 0;
}
```

Output:

```
Constructor Invoked.
Constructor Invoked.
Destructor Invoked.
Destructor Invoked.
```

The output demonstrates that the constructor is called twice when the two Employee objects are created, and the destructor is also called twice when the objects go out of scope at the end of main().

C++ this Pointer :

The this pointer is a special pointer that points to the current object. It can be used for several purposes:

- To pass the current object as a parameter to another method.
- To refer to the current class's instance variables.
- To declare indexers.

Example:

This code demonstrates the use of this-> to distinguish between a member variable (this->id) and a parameter (id) with the same name.

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

class Employee {
public:
  int id;
  string name;
  float salary;

  Employee(int id, string name, float salary) {
    this->id = id;
    this->name = name;
    this->salary = salary;
  }

  void display() {
    cout << id << " " << name << " " << salary << endl;
  }
};

int main(void) {
  Employee e1 = Employee(101, "Sonoo Jaiswal", 8900);
  Employee e2 = Employee(102, "Nakul", 5900);
  e1.display();
  e2.display();
  return 0;
}
```

Output:

```
101 Sonoo Jaiswal 8900
102 Nakul 5900
```

This page continues the previous example, completing the main() function and showing the output. The output demonstrates that the constructor correctly assigns values to the object's members using the this pointer.

C++ friend Function:

A friend function is a function that, although not a member of a class, has access to the class's private and protected members. The friend keyword is used to declare it.

Declaration:

```
class class-name {
   // ...
   friend data-type function-name(argument/s);
   // ...
};
```

Example:

```
#include <iostream>
using namespace std;
class Box {
private:
   int length;
public:
   Box() : length(0) {}
   friend int printlength(Box);
};
int printlength(Box b) {
   b.length += 10;
   return b.length;
}
int main() {
   Box b;
   cout << "Length of box: " << printlength(b) << endl;
   return 0;
}
Output:
Length of Box: 10.
```

C++ Inheritance :

Inheritance is an OOP process where one object (the derived class) acquires all the properties and behaviors of another object (the base class).

Advantages:

- **Code Reusability:** It allows you to reuse members of the parent class without redefining them.

Terminology:

- **Derived Class:** The class that inherits members.
- **Base Class:** The class whose members are inherited.

Notes:

- The default visibility mode in C++ is private.
- Private members of a base class are never inherited.

Types of Inheritance and Example:

This page lists the different types of inheritance and begins a single inheritance example.

Types:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Example:

The Programmer class inherits from the Account class, gaining access to its salary member.

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

class Account {
public:
    float salary = 60000;
};

class Programmer : public Account {
    public:
    float bonus = 5000;
};
```

```
int main(void) {
   Programmer p1;
   cout << "Salary: " << p1.salary << endl;
   cout << "Bonus: " << p1.bonus << endl;
   return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

Visibility Modes and Inheritance :

This page defines the three visibility modes and provides a table showing how they affect inherited members.

Visibility Modes:

- **Public:** Members are accessible to all functions in the program.
- **Private:** Members are accessible only within their own class.
- **Protected:** Members are accessible within their own class and by derived classes.

Visibility of Inherited Members Table:

Base Class Visibility	Derived Class: public	Derived Class: private	Derived Class: protected
private	Not Inherited	Not Inherited	Not Inherited
protected	protected	private	protected
public	public	private	protected

C++ Polymorphism :

Polymorphism, from the Greek words "poly" (many) and "morphism" (forms), means "many forms."

Types of Polymorphism:

- **Compile-time polymorphism:** Achieved through **overloading** (function and operator).
- **Runtime polymorphism:** Achieved through **virtual functions**.

Example:

This page begins a polymorphism example with two classes, Animal and Dog, to show how a member variable can have different values in different class contexts.

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

class Animal {
public:
   string color = "Black";
};

class Dog : public Animal {
   public:
   string color = "Gray";
};
int main(void) {
   Animal d;
   cout << d.color << endl;
   return 0;
}
```

Output:

```
Black.
```

The output is "Black" because the main function creates an Animal object, not a Dog object, demonstrating that the object's type at creation determines the value of the color variable.

C++ Overloading:

C++ overloading allows you to create two or more members with the same name, as long as they differ in the number or type of parameters. This concept can be applied to methods, constructors, and operators. The page also lists the types of overloading:

1. Function Overloading
2. Operator Overloading.

Operator Overloading:

This page discusses **operator overloading**, a type of polymorphism. However, it notes that certain operators cannot be overloaded:

- The scope operator (::).
- sizeof.
- The member selector (.).
- The member pointer selector (.*).
- The ternary operator (?:).

The page also provides the syntax for operator overloading:

return_type class_name::operator op (parameters) { body of function }.

Operator Overloading Example:

This page provides a code example for operator overloading, demonstrating how the

++ operator can be overloaded to increment a member variable by two.

```
#include <iostream>

using namespace std;

class Test {

private:

   int num;

public:

   Test() : num(8) {}

   void operator++() {

      num = num + 2;

   }

   void Print() {

      cout << "The Count is: " << num;

   }

};

int main() {

   Test tt;

   ++tt;

   tt.Print();

   return 0;

}
```

C++ Function Overriding:

Function overriding occurs when a derived class defines the same function as its base class.

Function overloading is a C++ feature that allows you to define multiple functions with the same name but with different numbers or types of parameters. The compiler determines which specific function to call based on the arguments provided during the function call.

Function Overloading Example:

```
#include <iostream>

using namespace std;

class Animal {

public:

   void eat() {

      cout << "Eating...";

   }

};

class Dog : public Animal {

public:

   void eat() {

      void eat() {

      cout << "Eating bread...";

   }

};

int main(void) {

   Dog d = Dog();

   d.eat();

   return 0;

}
```

Output:

```
Eating bread...
```

The program prints "Eating bread..." because the overridden function in the

Dog class is called.

Abstraction:

Abstraction is the process of hiding internal details and showing only the necessary functionality. This can be achieved using **abstract classes** and **interfaces**. The page provides an example of an abstract class named Shape. The virtual void draw() = 0; line declares a pure virtual function, making Shape an abstract class.

This page shows the continuation of the abstraction example with Rectangle and Circle classes that inherit from Shape and provide their own implementations of the draw() function. The main() function creates objects of these classes and calls their draw() method, demonstrating polymorphism and abstraction·

Example:

```
#include <iostream>

using namespace std;

class Shape {

public:

   virtual void draw() = 0;

};
```

C++ Strings :

The std::string class represents a sequence of characters. It provides functions for operations like concatenation and comparison. The page shows a simple string concatenation example.

```
#include <iostream>

#include <string>

using namespace std;

int main() {

   string s1 = "Hello";

   string s2 = "C++";

   string s3 = s1 + s2;

   cout << s3 << endl;

   return 0;

}
```

Output:

```
HelloC++
```

C++ Exception Handling:

Exception handling is the process of managing runtime errors to ensure the normal flow of a program continues. The benefit is that the rest of the code can still execute even after an exception occurs.

The page lists several exception classes, including :

std::exception, std::bad_alloc, and std::runtime_error.

Exception Classes and Descriptions:

This page provides a table that describes some of the C++ exception classes.

- **std::exception**: The base class for all standard C++ exceptions.
- **std::logic_error**: An exception that is typically detected in the code.
- **std::runtime_error**: An exception detected during runtime.
- **std::bad_exception**: Used to handle unexpected exceptions.
- **std::bad_cast**: An exception generally thrown by dynamic_cast.
- **std::bad_alloc**: An exception typically thrown by new.

Exception Handling Keywords:

Exception handling in C++ is performed using the

try and catch statements. The page defines three key keywords:

- **try**: A block of code that may cause an exception.
- **catch**: A block of code that handles an exception thrown from the try block.
- **throw**: A statement that throws an exception when a problem occurs.

C++ Exception Classes:

The following exception classes are listed in the document as part of C++'s exception handling framework[1]:

- std::exception
- std::bad_alloc
- std::bad_cast
- std::bad_exception
- std::bad_typeid
- std::bad_function_call
- std::logic_error
- std::runtime_error
- std::overflow_error
- std::underflow_error
- std::range_error

- std::domain_error
- std::invalid_argument
- std::length_error
- std::out_of_range

Descriptions of Key Exception Classes:

The document provides descriptions for several of these classes.

Exception Class	Description
std::exception	It is the base class for all standard C++ exceptions.
std::logic_error	It is an exception that is typically detected in the code.
std::runtime_error	It is an exception that is detected during runtime.
std::bad_exception	It is used to handle unexpected exceptions.
std::bad_cast	This exception is generally thrown by dynamic_cast
std::bad_alloc	This exception is generally thrown by the new operator when memory allocation fails