C++ FOR ABSOLUTE BEGINNERS

educative

C++ Hello World

```cpp
#include <iostream>

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

Getting Started with C++

Header Types

Introduction to headers in C++

In C++, header files are crucial in organizing code, facilitating modularity, and promoting code reusability. They typically contain declarations of functions, classes, and other constructs without providing the full implementation.

Common header files

Here are some commonly used header files:

```cpp
// Input and output operations
#include <iostream>
// Mathematical functions
#include <cmath>
// String manipulation
#include <string>
// Dynamic array implementation
#include <vector>
// File input and output
#include <fstream>
```

Comments in C++

Single-line comment

To indicate a single-line comment, use two forward slashes //.

```cpp
// This is a single-line comment
std::cout << "Educative";
```

Multiline comment

To indicate a multiline comment, use /* to start a multiline comment and */ to end it.

```cpp
/* This is a multiline comment.
The cout statement below will print Educative.*/
std::cout << "Educative";
```

Variables, Data Types, and Operators

Understanding Variables

A variable is a named container that holds data.

```cpp
int count = 10; // 'count' is a variable holding the value 10
```

Data Types

C++ supports various data types.

```cpp
int integerVar = 5; // integer
float floatVar = 2.5; // float
char charVar = 'a'; // character
double doubleVar = 1.2345; // double
bool boolVar = false; // bool
```

Operators in C++

Operators are symbols or keywords that perform operations on values or variables.

Name	Operator	Example (x = 6, y = 2)	Output
Addition	+	x + y	8
Subtraction	-	x - y	4
Multiplication	*	x * y	12
Division	/	x/y	3
Modulo	%	x % y	0
Increment	++	++x	7
Decrement	--	--x	5

Comparison operators

Name	Operator	Example (x = 6, y = 2)	Output
Equal to	==	x == y	Evaluates to 0 (false) because 6 is not equal to 2
Not equal to	!=	x != y	Evaluates to 1 (true) because 6 is not equal to 2
Greater than	>	x > y	Evaluates to 1 (true) because 6 greater than 2
Less than	<	x < y	Evaluates to 0 (false) because 6 is not less than 2
Greater than or equal to	>=	x >= y	Evaluates to 1 (true) because 6 is greater than or equal to 2
Less than or equal to	<=	x <= y	Evaluates to 0 (false) because 6 is neither less than or equal to 2

Logical operators

Name	Operator	Example (x = 6)	Output
Logical AND	&&	x > 3 && x < 10	Evaluates to 1 (true) because 6 is greater than 3 AND 6 is less than 10
Logical OR	||	x > 3 || x < 4	Evaluates to 1 (true) because one of the conditions are true (6 is greater than 3, but 6 is not less than 4)
Logical NOT	!	!(x > 3 && x < 10)	Evaluates to 0 (false) because ! is used to reverse the result

Assignment operators

Name	Operator	Example (x = 6, y = 2)	Output
Add and assign	+=	x += y	8
Subtract and assign	-=	x -= y	4
Multiply and assign	*=	x *= y	12
Divide and assign	/=	x /= y	3
Modulo and assign	%=	x %= y	0
Exponent and assign	&=	x &= y	2
Bitwise OR and assign	|=	x |= y	6
Bitwise XOR and assign	^=	x ^= y	4
Right shift and assign	>>=	x >>= y	1
Left shift and assign	<<=	x <<= y	24

Conditional Statements

Based on certain conditions, users can control the flow of the program using conditional statements.

The if statement

The `if` statement instructs the compiler to execute the block of code only if the condition holds true.

```cpp
int num = 10;

if (num >= 8) {
    std::cout << "The number is greater than 8." << std::endl;
}
```

The if-else statement

The `if-else` statement instructs the compiler to execute the `if` block only if the specified condition is true. Otherwise, it executes the `else` block.

```cpp
int num = 10;

if (num % 2 == 0) {
    std::cout << "Even number." << std::endl;
}
else {
    std::cout << "Odd number." << std::endl;
}
```

Nested if statement

Nested if statements refer to an `if` statement inside another `if` statement.

```cpp
int x = 5, y = 3;

if (x > y) {
    if (x > 0) {
        std::cout << "x is positive." << std::endl;
    }
}
```

The else-if statement

The `else-if` statement allows us to check for multiple conditions sequentially.

```cpp
int score = 60;

if (score >= 90) {
    std::cout << "A grade" << std::endl;
}
else if (score >= 80) {
    std::cout << "B grade" << std::endl;
}
else {
    std::cout << "C grade" << std::endl;
}
```

C++ FOR ABSOLUTE BEGINNERS

educative

Switch statement

A **switch** statement allows us to check an expression against multiple cases. If a match is found, the code within that case is executed. A case can be ended with the **break** keyword. When no case matches, the **default** case is executed.

```
int marks = 88;

switch (marks/10) {
    case 10:
    case 9:      // Range of marks 90 - 100
        std::cout << "Perfect Grade!" << std::endl;
        break;
    case 8:
        std::cout << "Good work!" << std::endl;
        break;
    default:
        std::cout << "Try working hard!" << std::endl;
}
```

Loops

Loops are used to repeatedly execute a block of code multiple times.

For loop

The **for** loop is used to execute a block of code for a set number of times.

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

While loop

The **while** loop repeatedly executes a block of code till a given condition is true.

```
int counter = 3;

while (counter > 0) {
    std::cout << counter << std::endl;
    counter--;
}
```

Do-while loop

The **do-while** loop is similar to the **while** loop. In the **do-while** loop, the condition is checked after certain tasks have been performed, whereas in the **while** loop, the condition is checked first.

```
int num;

do {
    std::cout << "Enter a positive number: ";
    std::cin >> num;
} while (num <= 0);
```

Input and Output in C++

Input using `cin`

We take input from the user using **cin**

```
int age;
cin >> age; // Take input using cin
```

Output using `cout`

We can print output to the console using **cout**

```
std::cout << "The age you entered is: " << age << std::endl; // Print
output to the console
```

Buffering

Buffering refers to the storage and manipulation of data in a temporary location (buffer) before it is processed or displayed. Buffers are used to efficiently handle input and output operations by storing data in memory. In the example below, we read input characters one by one and count the number of uppercase letters (‘A' to ‘Z') encountered until a period ('.'). The program aims to count the number of uppercase letters in the given input string.

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

int main()
{
    int countCapAlphas = 0;
    char sym;

    cout << "Enter a string (terminate with '.'): ";
    cin >> sym;    // sjdkfJHAAAH.N2671637A

    while (sym != '.')
    {
        if (sym >= 'A' && sym <= 'Z')
            countCapAlphas++;

        cin >> sym;
    }

    cout << "Number of uppercase letters: " << countCapAlphas << endl;

    return 0;
}
```

Arrays

An array is a data structure that stores a fixed number of elements (which must be known at compile-time) of the same data type in contiguous memory locations.

Basics of arrays

Creation

```
// Declaring an array to store up to 5 integers
int nums[5];

// Declaring and initializing an array with values 1, 2, 3, 4, and 5
int nums1[5]= {1, 2, 3, 4, 5};

// Initializing an array with the values 1 and 2, and the remaining
elements are set to 0
int nums2[5] = {1,2};

// Initializing an array with all elements set to 0
int nums3[5] = {0};

// Automatically determining the size and initializing the array with
values 1, 2, 3, 4, and 5
int nums4[] = {1,2,3,4,5};
```

Another example of creation is by initializing an integer array with a maximum capacity and allowing users to input the size of the array at runtime and then iterate through the array elements up to the user-defined size, potentially operating on each element.

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

const int capacity = 100; // Maximum capacity of the array

int main() {
    int Array[capacity] = {}; // Initialize array with all elements set to 0
    int size; // To hold the size of the array, to be input by the user

    cout << "Enter the size of the array (up to " << capacity << "): ";
    cin >> size; // Input the size of the array

    for(int i = 0; i < size; i++) {
        Array[i] += 1;
    }

    cout << "Array elements are:\n";
    for(int i = 0; i < size; i++) {
        cout << Array[i] << " "; // Output each element of the array
    }
    cout << endl;

    // The output in the console is 1,1,1,1,1

    return 0;
}
```

Access

```
int firstElement = nums[0]; // accessing the first element in the array
```

Iterations

Example 1

```
int sum = 0;

// loop over every element to calculate the sum of elements in an array
for (int i = 0; i < 5; i++) {
    sum += nums[i];
}
```

Example 2

```
#include <iostream>

int main() {
    int A[] = {1, 2, 3};

    // Iterate over each element 'r' of array 'A'
    // Using a range-based loop, increment each element by 1
    for(int &r : A) {
        r++; // Incrementing each element by 1
    }

    // Iterate over each element 'i' of array 'A'
    // Using a range-based loop, print each element separated by a space
    for(int &i : A) {
        std::cout << i << " "; // Output: 2, 3, 4
    }

    return 0;
}
```

Example 3

Multi-dimensional arrays

A multi-dimensional array is an array of arrays that stores similar types of data in tabular form.

```
// Declaring a 2D array with 3 rows and 4 columns and initializing it with
specific values
int 2dArray[3][4] =  {{ 1, 2, 3, 4 }, { 5, 6, 7, 8 }, {9, 7, 1, 3}};

// Declaring a 2D array with 3 rows and 4 columns, and initializing it with
zero values for all elements
int 2dArray[3][4] = {};
```

Access

```
int 2dArray[3][4] =  {{ 1, 2, 3, 4 }, { 5, 6, 7, 8 }, {9, 7, 1, 3}};

// Output: 1
std::cout << "Element at 2dArray[0][0]: " << 2dArray[0][0] << std::endl;

// Output: 8
std::cout << "Element at 2dArray[1][3]: " << 2dArray[1][3] << std::endl;
```

C++ FOR ABSOLUTE BEGINNERS

educative

Operations

```
int 2dArray[3][4] =  {{ 1, 2, 3, 4 }, { 5, 6, 7, 8 }, {9, 7, 1, 3}};

2dArray[1][3] = 20;
// Output: 20
std::cout << "Modified element at arr[1][3]: " << 2dArray[1][3] << std::endl;

// Nested loops for iterating through the 2D array
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        std::cout << 2dArray[i][j] << " ";
    }
    std::cout << std::endl;
}
```

Function syntax & structure

```
return_type function_name( parameter list ) {
    // Function body
    // ...
    // return statement (if applicable)
}
```

- **`return_type`:** The type of value being returned by the function
- **`function_name`:** The name of the function
- **`parameter`:** The input values to the function

Note: A function body comprises a set of instructions that compute the result based on the input parameters.

Practical examples of functions

Example 1: Pass arguments by value

```
#include <iostream>

// Function to compute the cube of a number
double cube(double number) {
    return number * number * number;
}

int main() {
    double num = 4;

    double result = cube(num);
    // Output: 64
    std::cout << "The cube of " << num << " is: " << result << std::endl;

    return 0;
}
```

Example 2: Pass arguments by reference

```
#include <iostream>

// Function to compute the cube of a number using reference (include the &
operator before the parameter name)
void cube(double& number) {
    number = number * number * number;
}

int main() {
    double num = 4;

    cube(num);
    // Output: 64
    std::cout << "The cube of 4 is:" << num << std::endl;

    return 0;
}
```

Example 3: Function calls with arrays

When we pass an array to a function, what's actually passed is the base address or the memory address of the first element of the array.

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

void funWithArrays(int A[], int size) {
    // Function to print elements of the array
    cout << "Array elements: ";
    for(int i = 0; i < size; ++i) {
        cout << A[i] << " ";
    }
    cout << endl;
}

int main() {
    int A[] = {1,2,3};
    int sizeA = 3;
    funWithArrays(A, sizeA); // 1,2,3

    int B[] = {1,2,3,4};
    int sizeB = 4;
    funWithArrays(B, sizeB); 1,2,3,4
}
```

Pointers

A pointer is a variable that stores the memory address of another variable. To create a pointer, we use the type keyword followed by an asterisk (*) symbol followed by the name of the pointer variable.

Basics of pointers

General syntax

```
type * pointer_name;
```

Here, **`type`** is the type of memory the pointer points to, and **`pointer_name`** is the name of the pointer variable.

Initialize memory address of a variable to a pointer

```
type *pointer_name = &variable_name;
// OR
pointer_name = &variable_name; // if pointer_name is already declared
```

Declaring and initializing pointers

```
// Declaration
int* ptr;

// Initialization
int* ptr = nullptr;
```

Real-life example

```
int a = 10;
int* ptr = &a; // Pointer to the variable a
```

Pointer Arithmetic

Pointer arithmetic refers to the mathematical operations performed on pointers. It allows us to navigate through memory efficiently. Here's an in-depth explanation of pointer arithmetic:

```
// Translates to
p++;     // p = p + sizeof(Type);
p--;     // p = p - sizeof(Type);
p+=2;    // p = p + n*sizeof(Type);
int numbers[] = {1,2,3,4};
int n = &numbers[3] - numbers[1]; // translates to (&(numbers[3] - &(num-
bers[1]))) / sizeof(Type)
```

Incrementing/Decrementing a Pointer

When a pointer is incremented (e.g., p++), it moves to the next memory location based on the size of the data type it points to. Similarly, decrementing a pointer (e.g., p--) moves it to the previous memory location by subtracting the size of the data type. Here's an example demonstrating pointer increment/decrement:

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

int main() {
  int numbers[] = {10, 20, 30, 40};
  int* p = numbers;  // Pointer to the first element of the array

  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

  p++;  // Moves the pointer to the second element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

 p++;  // Moves the pointer to the third element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

 p++;  // Moves the pointer to the fourth element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

 p--;  // Moves the pointer to the third element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

 p--;  // Moves the pointer to the second element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

 p--;  // Moves the pointer to the first element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

  return 0;
}
```

Pointer arithmetic with integral value

We can perform arithmetic operations with integral values. Adding an integer value to a pointer adjusts its position based on the size of the data type it points to. Here's an example that demonstrates pointer arithmetic with an integral value:

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

int main() {
  int numbers[] = {1, 2, 3, 4};
  int* p = numbers;  // Pointer to the first element of the array

  p = p + 2;  // Moves the pointer to the third element of the array
  cout << "Address of p: "<< &p << " pointing to: "<< p << " have value
at: "<<*p << endl;

  return 0;
}
```

Pointer subtraction

We can also subtract two pointers of the same type to determine the number of elements between them. The result of the subtraction is divided by the size of the data type.

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

int main()
{
  int numbers[] = {1, 2, 3, 4, 5};
  int* p1 = &numbers[1];  // Pointer to the second element of the array
  int* p2 = &numbers[4];  // Pointer to the fifth element of the array

  int n = p2 - p1;

  cout << "Offset difference: " << n << endl;  // should display 3

  return 0;
}
```

C++ FOR ABSOLUTE BEGINNERS

Common Errors

Dangling pointers

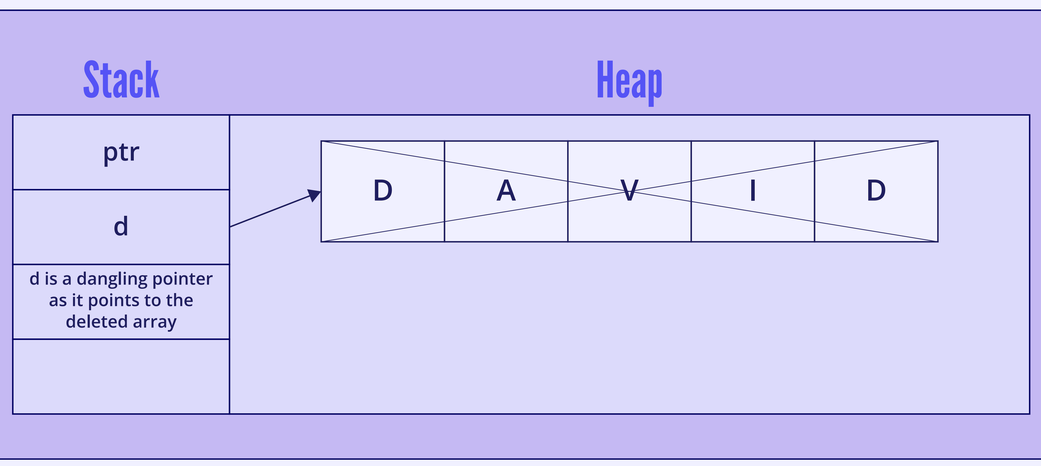
A dangling pointer is a pointer that points to a memory location that has been deallocated.

```
#include <iostream>

int main() {
    int* ptr = new int;
    *ptr = 42;
    int* ptr2 = ptr;

    // Deallocate memory, but the pointer still points to the same address
    delete ptr;

    std::cout << *ptr2 << std::endl; // Dangling pointer dereferencing
}
```



Dangling pointer

Memory leaks

Memory leaks occur when allocated memory is not properly deallocated.

```
#include <iostream>
#include <string.h>

int main()
{
  char * ptr = new char[6] {'D', 'A','V', 'I', 'D'};
  std::cout << "ptr: " << ptr <<'\n';

  ptr = new char[10]{'H', 'E','L', 'L', 'O',' ', 'C', '+', '+', '\0'};
  std::cout << "ptr: " << ptr <<'\n';

  return 0;
}
```

Memory is allocated on the heap for the character array `'DAVID'` using **`new`**, but the memory is not deallocated before assigning a new memory block to the same pointer, **`ptr`**.

Null pointer/illegal address dereferencing

A null pointer is a pointer that doesn't point to a valid memory location. Dereferencing it or attempting to access memory using an illegal address can lead to runtime errors and program crashes. Similarly, if a pointer is not initialized or explicitly set to a valid memory address, dereferencing it can lead to undefined behavior.

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

int main() {
  int* ptr;

  // Dereferencing a pointer with a garbage address will lead to undefined behavior
  *ptr = 10; // The program will crash

  ptr = nullptr;

  // Dereferencing a null pointer should lead to undefined behavior
  *ptr = 10; // The program will crash

  ptr = new int;

  // Check if memory allocation was successful
  if (ptr != nullptr) {
    *ptr = 42;
    std::cout << "Dynamically allocated value: " << *ptr << std::endl;
  }

  // Deallocate the memory
  delete ptr;
  ptr = nullptr;

  // Accessing the memory after deallocation
  if (ptr != nullptr) {
    std::cout << "This should not be executed.";
  }

  return 0;
}
```

Structs

Structs are user-defined data types that can be used to group items of possibly different types into a single type.

Basics of Structs

The **`struct`** keyword creates a structure, and each of its members is declared inside curly braces. The general syntax to create a structure is as follows:

```
struct structureName {
    type1 myIntVar; // Member 1
  type2 myStringVar; // Member 2
}
```

A real example of a structure is as follows:

```
struct Person {
    int age;
    string name;
}
```

Here a structure **`Person`** is defined which has two members: **`age`** and **`name`**.

Declaring and Initializing a Structure Variable

A structure is simply a new data type that is user-defined. We must declare a variable of this data type to initialize its members and access them.

Syntax to declare a structure variable

A structure can either be declared with a structure declaration or as a separate declaration similar to basic types as follows:

```
// Structure variable declaration with structure declaration
struct Person {
    int age;
    string name;
} person1; // The variable person1 is declared with 'Person'


// Structure variable declaration similar to basic data types
struct Person {
    int age;
    string name;
}

int main() {
    Person person1; // The variable person1 is declared like a normal variable in the main function
}
```

Structs initialization

Structure variables can be initialized at the time of declaration or later using assignment statements. There are two methods that we can use to initialize them.

Direct initialization

Direct initialization is done via the curly bracket ({}) notation, which contains the initialized values.

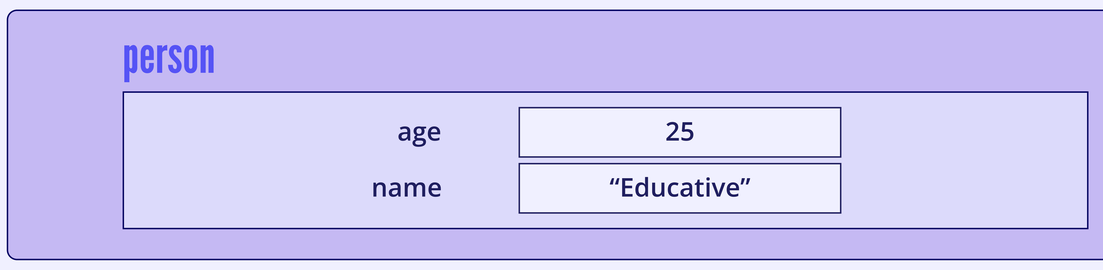
```
Person person = { 25, "Educative" };
```

Separate initialization

Separate initialization means assigning values to the structure variable members individually after the structure variable is declared using the . (dot) operator.

```
Person person;

person.age = 25;
person.name = "Educative";
```



Accessing structure members

Structure members are accessed using the dot (.) operator.

```
#include <iostream>

struct Person {
    int age;
    string name;
}

int main() {
    Person person1 = { 25, "Educative" };

    // Accessing Struct Members
    std::cout << "Age = " << person1.age << ", Name = " << person1.name;

    return 0;
}
```

C++ Built-In Toolbox

C++ provides a variety of built-in functions for common tasks such as mathematical operations, string manipulation, etc.

Built-in functions for strings

Several string functions are present in C++ that are used to perform operations on strings. Some of them are as follows:

- **`append:`** Appends a string at the end of the given string
- **`length:`** Returns the length of a string
- **`empty:`** Used to check if a string is empty
- **`substr:`** Extracts a substring from a given string
- **`compare:`** Compares two strings lexicographically

Built-in math functions

Several math functions are present in C++. Some of them are as follows:

- **`max:`** Returns the larger value among two values
- **`min:`** Returns the smaller value among two values
- **`sqrt:`** Returns the square root of a number
- **`ceil:`** Returns the value of the number rounded up to its nearest integer
- **`floor:`** Returns the value of the number rounded down to its nearest integer
- **`pow:`** Returns the value of num1 to the power of num2

Debugging: Identifying & Resolving Errors

Errors in programming are unexpected issues that disrupt the proper execution of a program.

Syntax

These are also called compile-time errors. For example, in the code below, we are missing a semicolon at the end of the `cout` statement, resulting in a syntax error.

```
#include <iostream>

int main() {
    // missing semicolon
    std::cout << "Geeks for geeks!"

    return 0;
}
```

Runtime

These are errors that occur during execution. An example can be dividing a number by `0`.

```
#include <iostream>

int main() {
    int x = 10;
    int y = 0;

    int result = x / y; // Division by zero

    return 0;
}
```

Logical

Logical errors run without crashing but produce incorrect results even though syntax and other factors are correct. For example, we perform integer division in the code below, and the result will be an integer. Even though we store it in a **`float`** variable, the output will be **`1`** instead of **`1.8`**.

```
#include <iostream>

int main() {
    int num1 = 2, num2 = 3, num3 = 4;
  float avg = (num1+num2+num3)/5; // this will store 1 in the avg variable

  std::cout<< avg << std::endl;

    return 0;
}
```

Linker

Linker errors occur when the program is successfully compiled and attempts to link the different object files with the main object file. These are generated when the executable of a program cannot be generated. For example, if **`main()`** is written as **`Main()`**, a linked error will be generated.

```
#include <iostream>
// main() is written as Main()
int Main() {
    std::cout << "Educative";

    return 0;
}
```

Semantic

Semantic errors occur when a statement is syntactically correct but has no meaning for the compiler. For example, if an expression is entered on the left side of the assignment operator, a semantic error may occur.

```
#include <iostream>

int main() {
    int a = 10;
    int b = 20;
    int c;

    // Semantic error here
    a + b = c;

    return 0;
}
```