Operators in C

Difficulty Level : Easy • Last Updated : 04 Apr, 2023

C Operators are symbols that represent operations to be performed on one or more operands. C provides a wide range of operators, which can be classified into different categories based on their functionality. Operators are used for performing operations on variables and values.

What are Operators in C?

Operators can be defined as the symbols that help us to perform specific mathematical, relational, bitwise, conditional, or logical computations on operands. In other words, we can say that an operator operates the operands. For example, '+' is an operator used for addition, as shown below:

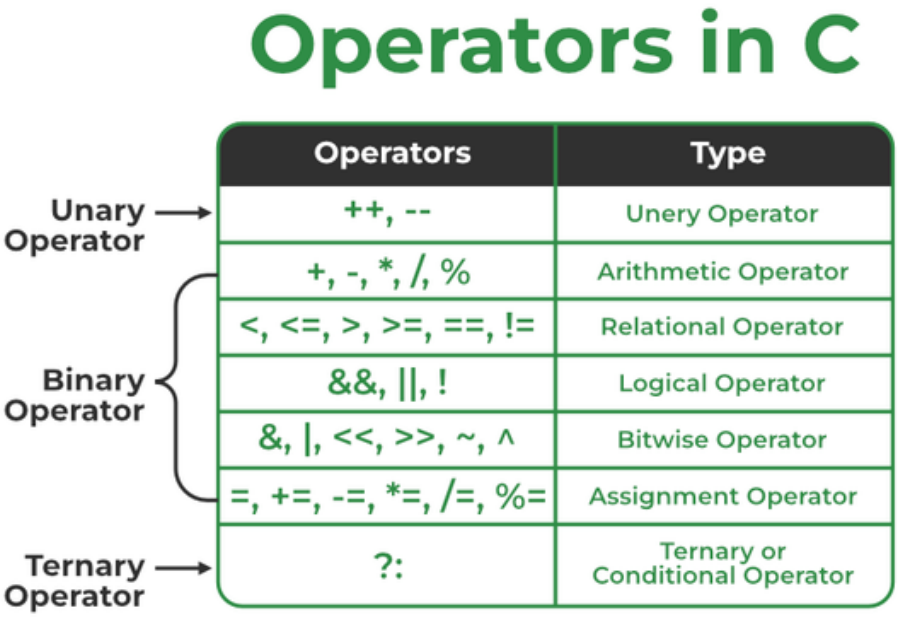
```
c = a + b;
```

Here, '+' is the operator known as the addition operator, and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'. The functionality of the C programming language is incomplete without the use of operators.

Types of Operators in C

C has many built-in operators and can be classified into 6 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators



Operators in C

The above operators have been discussed in detail:

1. Arithmetic Operations in C

These operators are used to perform arithmetic/mathematical operations on operands. Examples: (+, -, *, /, %,++,−). Arithmetic operators are of two types:

a) Unary Operators:

Operators that operate or work with a single operand are unary operators. For example: Increment(++) and Decrement(−) Operators

```
int val = 5;
cout<<++val;  // 6
```

b) Binary Operators:

Operators that operate or work with two operands are binary operators. For example: Addition(+), Subtraction(-), multiplication(*), Division(/) operators

```
int a = 7;
int b = 2;
cout<<a+b; // 9
```

2. Relational Operators in C

These are used for the comparison of the values of two operands. For example, checking if one operand is equal to the other operand or not, whether an operand is greater than the other operand or not, etc. Some of the relational operators are (==, >= , <=) (See this article for more reference).

```
int a = 3;
int b = 5;
cout<<(a < b);
// operator to check if a is smaller than b
```

3. Logical Operator in C

Logical Operators are used to combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

For example, the **logical AND** represented as the **'&&' operator in C** returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. Therefore, a && b returns true when both a and b are true (i.e. non-zero) (See this article for more reference).

```
cout<<((4 != 5) && (4 < 5));     // true
```

4. Bitwise Operators in C

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can

be performed at the bit level for faster processing. For example, the **bitwise AND** operator represented as **'&' in C** takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1 (True).

```
int a = 5, b = 9;   // a = 5(00000101), b = 9(00001001)
cout << (a ^ b);   //  00001100
cout <<(~a);       // 11111010
```

5. Assignment Operators in C

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

Different types of assignment operators are shown below:

a) "="

This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
Example:

```
a = 10;
b = 20;
ch = 'y';
```

b) "+="

This operator is the combination of the '+' and '=' operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
Example:

```
(a += b) can be written as (a = a + b)
If initially value stored in a is 5. Then (a += 6) = 11.
```

c) "-="

This operator is a combination of '-' and '=' operators. This operator first subtracts the value on the right from the current value of the variable on left and then assigns the result to the

variable on the left.

Example:

```
(a -= b) can be written as (a = a - b)
If initially value stored in a is 8. Then (a -= 6) = 2.
```

d) "*="

This operator is a combination of the '*' and '=' operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

Example:

```
(a *= b) can be written as (a = a * b)
If initially, the value stored in a is 5. Then (a *= 6) = 30.
```

e) "/="

This operator is a combination of the '/' and '=' operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

Example:

```
(a /= b) can be written as (a = a / b)
If initially, the value stored in a is 6. Then (a /= 2) = 3.
```

6. Other Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks. Some of them are discussed here:

i. sizeof operator

- sizeof is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by size_t.
- Basically, the sizeof the operator is used to compute the size of the variable.

To know more about the topic refer to this article.

ii. Comma Operator

- The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator.

To know more about the topic refer to this article.

iii. Conditional Operator

- The conditional operator is of the form ***Expression1 ? Expression2: Expression3***
- Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
- We may replace the use of if..else statements with conditional operators.

To know more about the topic refer to this article.

iv. dot (.) and arrow (->) Operators

- Member operators are used to referencing individual members of classes, structures, and unions.
- The dot operator is applied to the actual object.
- The arrow operator is used with a pointer to an object.

to know more about dot operators refer to this article and to know more about arrow(->) operators refer to this article.

v. Cast Operator

- Casting operators convert one data type to another. For example, int(2.2000) would return 2.
- A cast is a special operator that forces one data type to be converted into another.
- The most general cast supported by most of the C compilers is as follows – **[(type) expression]**.

To know more about the topic refer to this article.

vi. &,* Operator

- Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.
- The pointer operator * is a pointer to a variable. For example *var; will pointer to a variable var.

To know more about the topic refer to this article.

C Operators with Example

C

```c
// C Program to Demonstrate the working concept of
// Operators
#include <stdio.h>

int main()
{

    int a = 10, b = 5;
    // Arithmetic operators
    printf("Following are the Arithmetic operators in C\n");
    printf("The value of a + b is %d\n", a + b);
    printf("The value of a - b is %d\n", a - b);

    printf("The value of a * b is %d\n", a * b);
    printf("The value of a / b is %d\n", a / b);
    printf("The value of a % b is %d\n", a % b);
    // First print (a) and then increment it
    // by 1
    printf("The value of a++ is %d\n", a++);

    // First print (a+1) and then decrease it
    // by 1
    printf("The value of a-- is %d\n", a--);

    // Increment (a) by (a+1) and then print
    printf("The value of ++a is %d\n", ++a);

    // Decrement (a+1) by (a) and then print
    printf("The value of --a is %d\n", --a);

    // Assignment Operators --> used to assign values to
    // variables int a =3, b=9; char d='d';

    // Comparison operators
    // Output of all these comparison operators will be (1)
    // if it is true and (0) if it is false
    printf(
        "\nFollowing are the comparison operators in C\n");
    printf("The value of a == b is %d\n", (a == b));
```

```
    printf("The value of a != b is %d\n", (a != b));
    printf("The value of a >= b is %d\n", (a >= b));
    printf("The value of a <= b is %d\n", (a <= b));
    printf("The value of a > b is %d\n", (a > b));
    printf("The value of a < b is %d\n", (a < b));

    // Logical operators
    printf("\nFollowing are the logical operators in C\n");
    printf("The value of this logical and operator ((a==b) "
           "&& (a<b)) is:%d\n",
           ((a == b) && (a < b)));
    printf("The value of this logical or operator ((a==b) "
           "|| (a<b)) is:%d\n",
           ((a == b) || (a < b)));
    printf("The value of this logical not operator "
           "(!(a==b)) is:%d\n",
           (!(a == b)));

    return 0;
}
```

Output

```
Following are the Arithmetic operators in C
The value of a + b is 15
The value of a - b is 5
The value of a * b is 50
The value of a / b is 2
The value of a % b is 0
The value of a++ is 10
The value of a-- is 11
The value of ++a is 11
The value of --a is 10

Following are the comparison operators in C
The value of a == b is 0
The value of a != b is 1
The value of a >= b is 1
The value of a <= b is 0
The value of a > b is 1
The value of a < b is 0

Following are the logical operators in C
The value of this logical and operator ((a==b) && (a<b)) is:0
The value of this logical or operator ((a==b) || (a<b)) is:0
The value of this logical not operator (!(a==b)) is:1
```

Time and Space Complexity

Time Complexity: O(1)
Auxiliary Space: O(1)

Precedence of Operators in C

The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
1	**()**	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++/a–	Postfix increment/decrement (a is a variable)	left-to-right
2	**++a/–a**	Prefix increment/decrement (a is a variable)	right-to-left
	+/-	Unary plus/minus	right-to-left
	!~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*****	Dereference	right-to-left
	&	Address (of operand)	right-to-left

Precedence	Operator	Description	Associativity
	sizeof	Determine size in bytes on this implementation	right-to-left
3	***,/,%**	Multiplication/division/modulus	left-to-right
4	**+/-**	Addition/subtraction	left-to-right
5	**<< , >>**	Bitwise shift left, Bitwise shift right	left-to-right
6	**< , <=**	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7	**== , !=**	Relational is equal to/is not equal to	left-to-right
8	**&**	Bitwise AND	left-to-right
9	**^**	Bitwise exclusive OR	left-to-right
10	**|**	Bitwise inclusive OR	left-to-right
11	**&&**	Logical AND	left-to-right
12	**||**	Logical OR	left-to-right
13	**?:**	Ternary conditional	right-to-left
14	**=**	Assignment	right-to-left

Precedence	Operator	Description	Associativity
	+= , -=	Addition/subtraction assignment	right-to-left
	*= , /=	Multiplication/division assignment	right-to-left
	%= , &=	Modulus/bitwise AND assignment	right-to-left
	^= , |=	Bitwise exclusive/inclusive OR assignment	right-to-left
	<>=	Bitwise shift left/right assignment	right-to-left
15	,	expression separator	left-to-right

Conclusion

In this article, the points we learned about the operator are as follows:

- *Operators are symbols used for performing some kind of operation in C.*
- *The operation can be mathematical, logical, relational, bitwise, conditional, or logical.*
- *There are seven types of Unary operators, Arithmetic operator, Relational operator, Logical operator, Bitwise operator, Assignment operator, and Conditional operator.*
- *Every operator returns a numerical value except logical and conditional operator which returns a boolean value(true or false).*
- *'=' and '==' are not same as '=' assigns the value whereas '==' checks if both the values are equal or not.*
- *There is a Precedence in the operator means the priority of using one operator is greater than another operator.*

Frequently Asked Questions(FAQs)

1. What are operators in C?

Operators in C are certain symbols in C used for performing certain mathematical, relational, bitwise, conditional, or logical operations for the user.

2. What are the 7 types of operators in C?

There are 7 types of operators in C as mentioned below:

- Unary operator
- Arithmetic operator
- Relational operator
- Logical operator
- Bitwise operator
- Assignment operator
- Conditional operator

3. What is the difference between the '=' and '==' operators?

'=' is a type of assignment operator that places the value in right to the variable on left, Whereas '==' is a type of relational operator that is used to compare two elements if the elements are equal or not.

4. What is the difference between prefix and postfix operators in C?

Prefix operations are the operations in which the value is returned prior to the operation whereas in postfix operations value is returned after updating the value in the variable.

Example:

```
b=c=10;
a=b++;    // a==10
a=++c;    // a==11
```

5. What is the Modulo operator?

The Modulo operator(%) is used to find the remainder if one element is divided by another.

Example:

```
a % b (a divided by b)
5 % 2 == 1
```

Unary operators in C/C++

Difficulty Level : Basic • Last Updated : 03 Apr, 2023

Unary operators: are operators that act upon a single operand to produce a new value.

Types of unary operators:

1. unary minus(-)
2. increment(++)
3. decrement(- -)
4. NOT(!)
5. Addressof operator(&)
6. sizeof()

Time complexity of any unary operator is O(1).

Auxiliary Space of any unary operator is O(1).

1. unary minus: The minus operator changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive.

```
int a = 10;
int b = -a;  // b = -10
```

unary minus is different from the subtraction operator, as subtraction requires two operands.

Below is the implementation of **unary minus (-)** operator:

C++

```
// C++ program to demonstrate the use of 'unary minus'
// operator

#include <iostream>
using namespace std;

int main()
{
    int positiveInteger = 100;
    int negativeInteger = -positiveInteger;

    cout << "Positive Integer: " << positiveInteger << endl;
    cout << "Negative Integer: " << negativeInteger << endl;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

Output

```
Positive Integer: 100
Negative Integer: -100
```

2. increment: It is used to increment the value of the variable by 1. The increment can be done in two ways:

2.1 prefix increment: In this method, the operator precedes the operand (e.g., ++a). The value of the operand will be altered *before* it is used.

```
int a = 1;
int b = ++a;  // b = 2
```

2.2 postfix increment: In this method, the operator follows the operand (e.g., a++). The value operand will be altered *after* it is used.

```
int a = 1;
int b = a++;   // b = 1
int c = a;     // c = 2
```

3. decrement: It is used to decrement the value of the variable by 1. The decrement can be done in two ways:

3.1 prefix decrement: In this method, the operator precedes the operand (e.g., – -a). The value of the operand will be altered *before* it is used.

```
int a = 1;
int b = --a;  // b = 0
```

3.2 postfix decrement: In this method, the operator follows the operand (e.g., a- -). The value of the operand will be altered *after* it is used.

```
int a = 1;
int b = a--;   // b = 1
int c = a;     // c = 0
```

4. NOT(!): It is used to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.

```
If x is true, then !x is false
If x is false, then !x is true
```

Below is the implementation of the **NOT (!)** operator:

C++

```
// C++ program to demonstrate the use of '!(NOT) operator'

#include <iostream>
using namespace std;

int main()
{

    int a = 10;
    int b = 5;

    if (!(a > b))
        cout << "b is greater than a" << endl;
    else
        cout << "a is greater than b" << endl;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

Output

```
a is greater than b
```

5. Addressof operator(&): It gives an address of a variable. It is used to return the memory address of a variable. These addresses returned by the address-of operator are known as pointers because they "point" to the variable in memory.

```
& gives an address on variable n
int a;
int *ptr;
ptr = &a; // address of a is copied to the location ptr.
```

Below is the implementation of **Addressof operator(&)**:

C++

```
// C++ program to demonstrate the use of 'address-of(&)'
// operator

#include <iostream>
using namespace std;

int main()
{

    int a;
    int* ptr;

    ptr = &a;

    cout << ptr;

    return 0;
}

// This code is contributed by sarajadhav12052009
```

Output

```
0x7ffddcf0c8ec
```

6. sizeof(): This operator returns the size of its operand, in bytes. The *sizeof()* operator always precedes its operand. The operand is an expression, or it may be a cast.

Note: The **`sizeof()`** operator in C++ is machine dependent. For example, the size of an 'int' in C++ may be 4 bytes in a 32-bit machine but it may be 8 bytes in a 64-bit machine.

Below is the implementation of **sizeof()** operator:

C++

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

int main()
{
    float n = 0;
    cout << "size of n: " << sizeof(n);
    return 0;
}
```

Output

```
size of n: 4
```

167

Related Articles

1. Operators in C | Set 2 (Relational and Logical Operators)
2. What are the Operators that Can be and Cannot be Overloaded in C++?
3. Increment (Decrement) operators require L-value Expression
4. Order of operands for logical operators
5. Conversion Operators in C++
6. const_cast in C++ | Type Casting operators
7. How to sum two integers without using arithmetic operators in C/C++?
8. Execution of printf With ++ Operators in C
9. Conditionally assign a value without using conditional and arithmetic operators
10. new and delete Operators in C++ For Dynamic Memory

Pre-increment (or pre-decrement) With Reference to L-value in C++

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

Prerequisite: Pre-increment and post-increment in C/C++

In C++, pre-increment (or pre-decrement) can be used as l-value, but post-increment (or post-decrement) can not be used as l-value.
For example, following program prints *a = 20* (++a is used as l-value)

l-value is simply nothing but the memory location, which has an address.

CPP

```
// CPP program to illustrate
// Pre-increment (or pre-decrement)
#include <cstdio>

int main()
{
    int a = 10;

    ++a = 20; // works
    printf("a = %d", a);
    printf("\n");
    --a = 10;
    printf("a = %d", a);
    return 0;
}
```

Output:

```
a = 20
a = 10
```

Time Complexity: O(1)

The above program works whereas the following program fails in compilation with error *"non-lvalue in assignment"* (a++ is used as l-value)

CPP

```cpp
// CPP program to illustrate
// Post-increment (or post-decrement)
#include <cstdio>

int main()
{
    int a = 10;
    a++ = 20; // error
    printf("a = %d", a);
    return 0;
}
```

Error:

```
prog.cpp: In function 'int main()':
prog.cpp:6:5: error: lvalue required as left operand of assignment
 a++ = 20; // error
     ^
```

How ++a is Different From a++ as lvalue?

It is because ++a returns an *lvalue*, which is basically a reference to the variable to which we can further assign – just like an ordinary variable. It could also be assigned to a reference as follows:

```
int &ref = ++a; // valid
int &ref = a++; // invalid
```

Whereas if you recall how a++ works, it doesn't immediately increment the value it holds. For clarity, you can think of it as getting incremented in the next statement. So what basically happens is that, a++ returns an *rvalue*, which is basically just a value like the value of an expression that is not stored. You can think of a++ = 20; as follows after being processed:

```
int a = 10;

// On compilation, a++ is replaced by the value of a which is an rvalue:
10 = 20; // Invalid

// Value of a is incremented
a = a + 1;
```

That should help to understand why a++ = 20; won't work. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

144

Related Articles

1. lvalue and rvalue in C language
2. Output of the program | Dereference, Reference, Dereference, Reference....
3. When do we pass arguments by reference or pointer?
4. Reference to a pointer in C++ with examples and applications
5. Passing Reference to a Pointer in C++
6. Difference between Call by Value and Call by Reference
7. Can C++ reference member be declared without being initialized with declaration?
8. Different ways to use Const with Reference to a Pointer in C++

new and delete Operators in C++ For Dynamic Memory

Difficulty Level : Easy • Last Updated : 18 Oct, 2022

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack** (Refer to 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 for variable-length arrays.
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is it different from memory allocated to normal variables?

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 the programmer's responsibility to deallocate memory when no longer needed. If the programmer doesn't deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

How is memory allocated/deallocated in C++?

C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions

and also has two operators **new** and **delete,** that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a 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

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

Here, the 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 type 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 for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

```
pointer-variable = new data-type(value);
```

Example:

C++

```
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) {}
    cust() = default;
    //cust& operator=(const cust& that) = default;
};

int main()
{
    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    //OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);
    return 0;
}
```

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

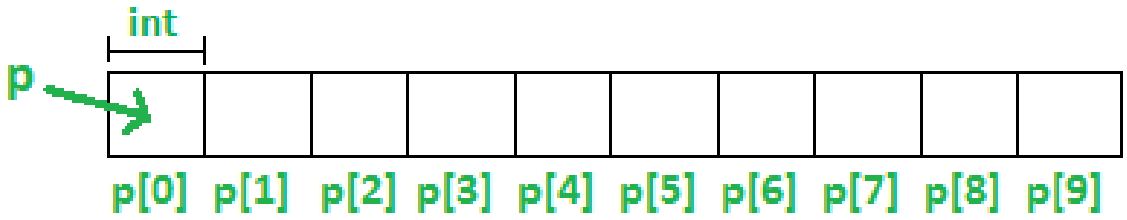
```
pointer-variable = new data-type[size];
```

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 a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.



Normal Array Declaration vs Using new

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

What if enough memory is not available during runtime?

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 a good idea to check for the pointer variable produced by the new before using its program.

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

delete operator

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

Syntax:

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

Here, the 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 the 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;
```

CPP

```cpp
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new(nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float *r = new float(75.25);

    cout << "Value of r: " << *r << endl;
```

```
    // Request block of memory of size n
    int n = 5;
    int *q = new(nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
```

Output

```
Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5
```

Time Complexity: O(n), where n is the given memory size.

Related Articles:

- Quiz on new and delete
- delete vs free

This article is contributed by **Akash Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

439

Related Articles