C Pointers

Difficulty Level : Easy • Last Updated : 23 Mar, 2023

Pointers in C are used to store the address of variables or a memory location. This variable can be of any data type i.e, int, char, function, array, or any other pointer. Pointers are one of the core concepts of C programming language that provides low-level memory access and facilitates dynamic memory allocation.

What is a Pointer in C?

A pointer is a derived data type in C that can store the address of other variables or a memory. We can access and manipulate the data stored in that memory location using pointers.

Syntax of C Pointers

```
datatype * pointer_name;
```

The above syntax is the generic syntax of C pointers. The actual syntax depends on the type of data the pointer is pointing to.

How to Use Pointers?

To use pointers in C, we must understand below two operators:

1. Addressof Operator

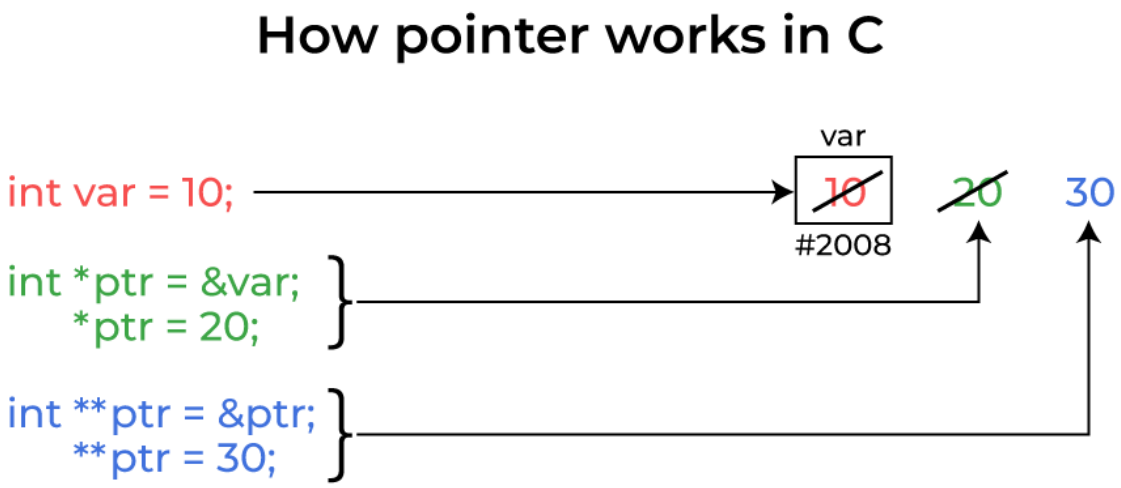
The addressof operator (&) is a unary operator that returns the address of its operand. Its operand can be a variable, function, array, structure, etc.

Syntax of Address of Operator

```
&variable_name;
```

2. Dereferencing Operator

The dereference operator (*), also known as the indirection operator is a unary operator. It is used in pointer declaration and dereferencing.



C Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the * dereference operator before its name.

```
data_type * pointer_name;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are also called wild pointers that we will study later in this article.

C Pointer Initialization

When we assign some value to the pointer, it is called Pointer Initialization in C. There are two ways in which we can initialize a pointer in C of which the first one is:

Method 1: C Pointer Definition

```
datatype * pointer_name = address;
```

The above method is called Pointer Definition as the pointer is declared and initialized at the same time.

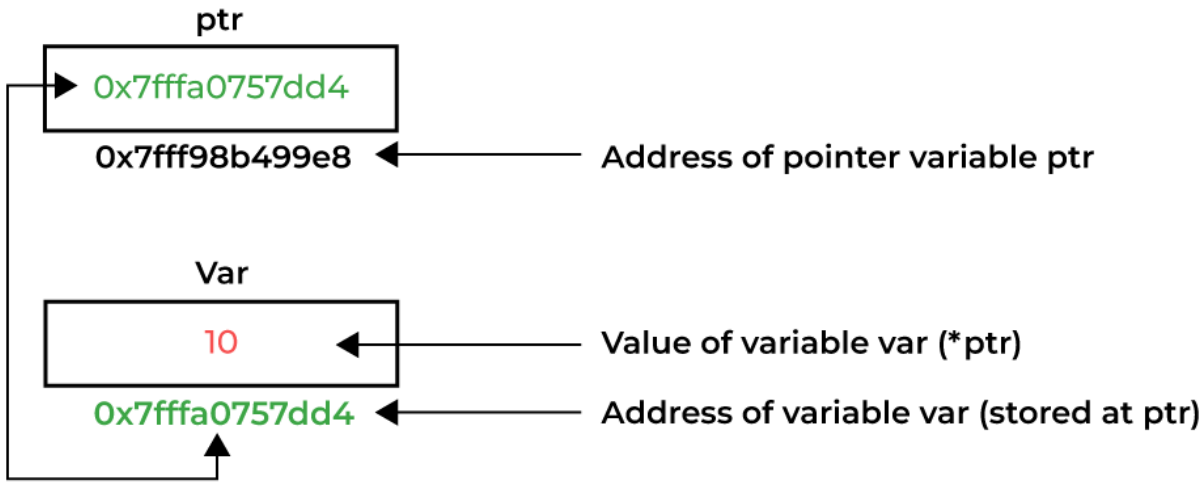
Method 2: Initialization After Declaration

The second method of pointer initialization in C the assigning some address after the declaration.

```
datatype * pointer_name;
pointer_name = addresss;
```

Dereferencing a C Pointer

Dereferencing is the process of accessing the value stored in the memory address specified in the pointer. We use dereferencing operator for that purpose.



Dereferencing a Pointer in C

Example: C Program to demonstrate how to use pointers in C.

C

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

void geeks()
{
    int var = 20;

    // declare pointer variable
```

```
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}

// Driver program
int main()
{
    geeks();
    return 0;
}
```

Output

```
Value at ptr = 0x7ffd15b5deec
Value at var = 20
Value at *ptr = 20
```

Types of Pointers

Pointers can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax of Integer Pointers

```
int *pointer_name;
```

These pointers are pronounced as Pointer to Integer.

Similarly, a pointer can point to any primitive data type. The syntax will change accordingly. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an

array using the given syntax.

Syntax of Array Pointers

```
char *pointer_name = &array_name;
```

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax of Structure Pointer

```
struct struct_name *pointer_name;
```

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the function pointer for this function will be

Syntax of Function Pointer

```
int (*pointer_name)(int, int);
```

Keep in mind that the syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax of Double Pointer in C

```
datatype ** pointer_name;
```

Dereferencing in Double Pointer

```
*pointer_name; // get the address stored in the inner level pointer
**pointer_name; // get the value pointed by inner level pointer
```

Note: *In C, we can create multi-level pointers with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on.*

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax of NULL Pointer in C

```
data_type *pointer_name = NULL;
        or
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax of Void Pointer

```
void * pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

Example of Wild Pointers

```
int *ptr;
char *str;
```

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax of Constant Pointer

```
const data_type * pointer_name;
```

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax to Pointer to Constant

```
data_type * const pointer_name;
```

Other Types of Pointers in C:

There are also the following types of pointers available to use in C apart from those specified above:

- **Far pointer:** A far pointer is typically 32-bit that can access memory outside the current segment.
- **Dangling pointer:** A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- **Huge pointer:** A huge pointer is 32-bit long containing segment address and offset address.
- **Complex pointer:** Pointers with multiple levels of indirection.
- **Near pointer:** Near pointer is used to store 16-bit addresses means within the current segment on a 16-bit machine.
- **Normalized pointer:** It is a 32-bit pointer, which has as much of its value in the segment register as possible.
- **File Pointer:** The pointer to a FILE data type is called a stream pointer or a file pointer.

Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

How to find the size of pointers in C?

We can find the size of pointers using the **sizeof** operator as shown in the following program:

Example: C Program to find the size of different pointer types.

C

```c
// C Program to find the size of different pointers types
#include <stdio.h>

// dummy structure
struct str {
};

// dummy function
void func(int a, int b){};

int main()
{
    // dummy variables definitions
    int a = 10;
    char c = 'G';
    struct str x;

    // pointer definitions of different types
    int* ptr_int = &a;
    char* ptr_char = &c;
    struct str* ptr_str = &x;
    void (*ptr_func)(int, int) = &func;
    void* ptr_vn = NULL;

    // printing sizes
    printf("Size of Integer Pointer  \t:\t%d bytes\n",
           sizeof(ptr_int));
    printf("Size of Character Pointer\t:\t%d bytes\n",
           sizeof(ptr_char));
    printf("Size of Structure Pointer\t:\t%d bytes\n",
           sizeof(ptr_str));
    printf("Size of Function Pointer\t:\t%d bytes\n",
           sizeof(ptr_func));
    printf("Size of NULL Void Pointer\t:\t%d bytes",
           sizeof(ptr_vn));

    return 0;
}
```

Output

```
Size of Integer Pointer      :    8 bytes
Size of Character Pointer    :    8 bytes
Size of Structure Pointer    :    8 bytes
Size of Function Pointer    :    8 bytes
Size of NULL Void Pointer    :    8 bytes
```

As we can see, no matter what the type of pointer it is, the size of each and every pointer is the same.

Now, one may wonder that if the size of all the pointers is the same, then why do we need to declare the pointer type in the declaration? The type declaration is needed in the pointer for dereferencing and pointer arithmetic purposes.

Pointer Arithmetic

Only a limited set of operations can be performed on pointers. The Pointer Arithmetic refers to the legal or valid operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations. The operations are:

- Increment in a Pointer
- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.

C

```c
// C program to illustrate Pointer Arithmetic

#include <stdio.h>

int main()
{

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

    // Declare pointer variable
    int* ptr;

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

```
    for (int i = 0; i < 3; i++) {

        // print value at address which is stored in ptr
        printf("Value of *ptr = %d\n", *ptr);

        // print value of ptr
        printf("Value of ptr = %p\n\n", ptr);

        // Increment pointer ptr by 1
        ptr++;
    }
    return 0;
}
```

Output

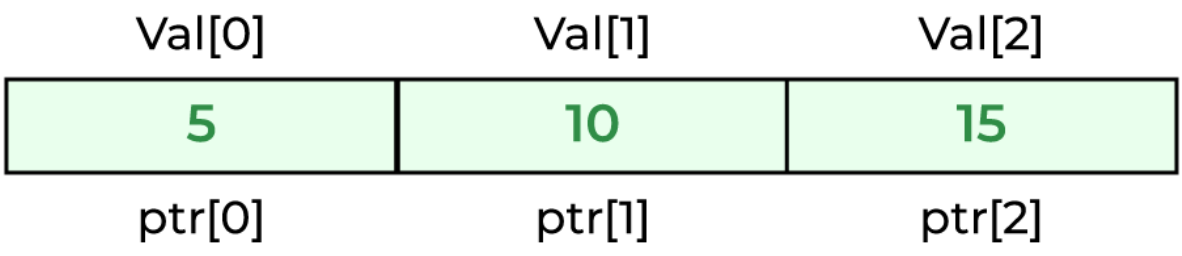
```
Value of *ptr = 10
Value of ptr = 0x7ffe8ba7ec50

Value of *ptr = 100
Value of ptr = 0x7ffe8ba7ec54

Value of *ptr = 200
Value of ptr = 0x7ffe8ba7ec58
```

C Pointers and Arrays Relation

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant pointer to the array, then we can access the elements of the array using this pointer.

Example 1: Accessing Array Elements using Pointer with Array Subscript



C

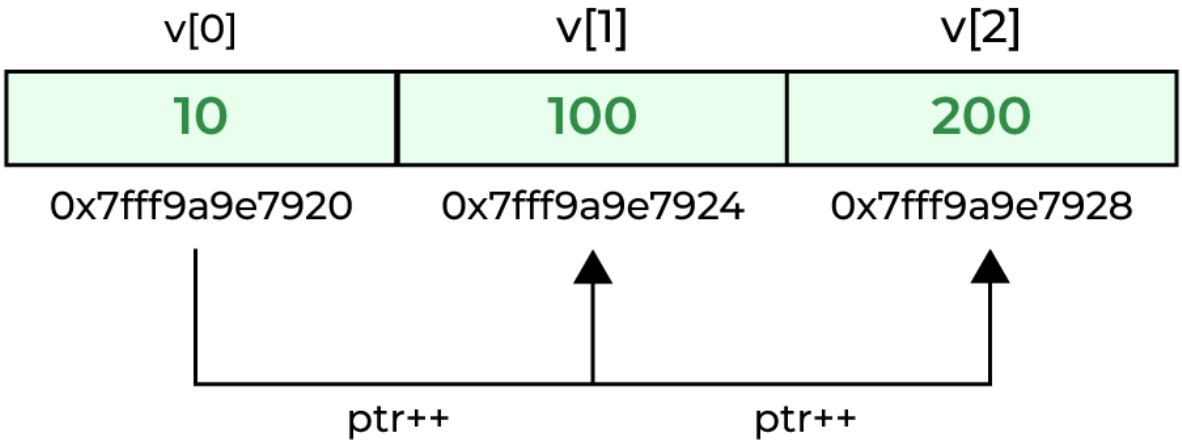
```c
// C Program to access array elements using pointer
#include <stdio.h>

void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 15 };

    // Declare pointer variable
    int* ptr;

    // Assign address of val[0] to ptr.
    // We can use ptr=&val[0];(both are same)
    ptr = val;

    printf("Elements of the array are: ");

    printf("%d, %d, %d", ptr[0], ptr[1], ptr[2]);

    return;
}

// Driver program
int main()
{
    geeks();
    return 0;
}
```

Output

```
Elements of the array are: 5 10 15
```

Not only that, as the array elements are stored continuously, we can pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

Example 2: Accessing Array Elements using Pointer Arithmetic



C

```c
// C Program to access array elements using pointers
#include <stdio.h>

int main()
{

    // defining array
    int arr[5] = { 1, 2, 3, 4, 5 };

    // defining the pointer to array
    int* ptr_arr = &arr;

    // traversing array using pointer arithmetic
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr_arr++);
    }
    return 0;
}
```

Output

```
 1 2 3 4 5
```

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element perfectly fine using this concept.

To know more about pointers to an array, refer to this article – Pointer to an Array

Uses of Pointers

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It finds its use in operations such as

1. Pass Arguments by Reference
2. Accessing Array Elements
3. Return Multiple Values from Function
4. Dynamic Memory Allocation
5. Implementing Data Structures
6. In System-Level Programming where memory addresses are useful.
7. In locating the exact value at some memory location.
8. To avoid compiler confusion for the same variable name.
9. To use in Control Tables.

Advantages of Pointers

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Disadvantages of Pointers

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

Conclusion

In conclusion, pointers in C are very capable tools and provide C language with its distinguishing features, such as low-level memory access, referencing, etc. But as powerful as they are, they should be used with responsibility as they are one of the most vulnerable parts of the language.

FAQs on Pointers in C

1. Define pointers.

Pointers are the variables that can store the memory address of another variable.

2. What is the difference between a constant pointer and a pointer to a constant?

A constant pointer points to the fixed memory location, i.e. we cannot change the memory address stored inside the constant pointer.

On the other hand, the pointer to a constant point to the memory with a constant value.

3. What is pointer to pointer?

A pointer to a pointer (also known as a double pointer) stores the address of another pointer.

4. Does pointer size depends on its type?

No, the pointer size does not depend upon its type. It only depends on the operating system and CPU architecture.

5. What are the differences between an array and a pointer?

The following table list the differences between an array and a pointer:

Pointer	Array
A pointer is a derived data type that can store the address of other variables.	An array is a homogeneous collection of items of any type such as int, char, etc.
Pointers are allocated at run time.	Arrays are allocated at runtime.
The pointer is a single variable.	An array is a collection of variables of the same type.
Dynamic in Nature	Static in Nature.

6. Why do we need to specify the type in the pointer declaration?

Type specification in pointer declaration helps the compiler in dereferencing and pointer arithmetic operations.

Quizzes:

Opaque Pointer in C++

Difficulty Level : Medium • Last Updated : 20 Jan, 2023

Opaque as the name suggests is something we can't see through. e.g. wood is opaque. An opaque pointer is a pointer that points to a data structure whose contents are not exposed at the time of its definition.

The following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

Why Opaque pointer?

There are places where we just want to hint to the compiler that "Hey! This is some data structure that will be used by our clients. Don't worry, clients will provide its implementation while preparing the compilation unit". Such a type of design is robust when we deal with shared code. Please see the below example:

Let's say we are working on an app to deal with images. Since we are living in a world where everything is moving to the cloud and devices are very affordable to buy, we want to develop apps for windows, android, and apple platforms. So, it would be nice to have a good design that is robust, scalable, and flexible as per our requirements. We can have shared code that would be used by all platforms and then different end-point can have platform-specific code. To deal with images, we have a CImage class exposing APIs to deal with various image operations (scale, rotate, move, save, etc).

Since all the platforms will be providing the same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have a different mechanism to access pixels of an image than Windows does. This means that APIs might demand different sets of info to perform operations. So to work on shared code, this is what we would like to do:

Image.h: A header file to store class declaration. We will create a header file that will contain a class CImage which will provide an API to handle the image operations.

```
// This class provides API to deal with various
// image operations. Different platforms can
// implement these operations in different ways.
class CImage
{
public:
    CImage();

    ~CImage();


// Opaque pointer


    struct SImageInfo* pImageInfo;

    void Rotate(double angle);

    void Scale(double scaleFactorX,
double scaleFactorY);

    void Move(int toX, int toY);

private:
```

```
    void InitImageInfo();
};
```

Image.cpp: Code that will be shared across different endpoints. This file is used to define the constructor and destructor of CImage class. The constructor will call the InitImageInfo() method and the code inside this method will be written in accordance with the Operating System on which it will work on.

```
// Constructor for CImage

CImage::CImage() {
    InitImageInfo();
}



// destructor for CImage class

CImage::~CImage()
{
    // Destroy stuffs here
}
```

Image_windows.cpp : Code specific to Windows operating System will reside in this file.

```
struct SImageInfo {
    // Windows specific DataSet
};

void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;
    // Initialize windows specific info here
}

void CImage::Rotate()
{
    // Make use of windows specific SImageInfo
}
```

Image_apple.cpp : Code specific to Mac Operating System will reside in this file.

```
struct SImageInfo {
    // Apple specific DataSet
};
void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;

    // Initialize apple specific info here
}
void CImage::Rotate()
{
    // Make use of apple specific SImageInfo
}
```

As it can be seen from the above example **while defining the blueprint of the CImage class we are only mentioning that there is a SImageInfo data structure.**

The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that data structure and use it as per their requirements. If in the future we want to develop an app for a new end-point 'X', the design is already there. We only need to define SImageInfo for end-point 'X' and use it accordingly.

Please note that the above-explained example is one way of doing this. Design is all about discussion and requirements. A good design is decided to take many factors into account. We can also have platform-specific classes like CImageWindows, and CImageApple and put all platform-specific code there.

Downsides of Opaque pointers

It's also important to note that opaque pointers can have some downsides, as well. For example, because the client code cannot access the implementation details of the object, it may be more difficult to debug or troubleshoot issues that arise. Additionally, opaque pointers can make it more difficult to understand the relationships between objects and their dependencies, which can make it harder to maintain and evolve the codebase over time.

Another potential downside is that opaque pointers can increase the complexity of the codebase, and make it harder to understand how the library is implemented. If a library is large, it can be difficult to understand the relationships between different parts of the codebase, and how they interact.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface. It's also

important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In summary, opaque pointers are a technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. They are useful for hiding the implementation details of an object from the client code, and also for providing a level of abstraction. However, it's important to use them judiciously and to consider other options as well.

It is also important to note that opaque pointers can cause a performance overhead, as it requires extra memory to store the pointer, and an additional level of indirection when accessing the object. It also increases the complexity of memory management as the client code cannot directly deallocate the memory of the object. The library must provide functions to handle the memory management, such as creating and destroying the object, or allocating and deallocating memory for the object.

Additionally, opaque pointers can make it more difficult to write unit tests for the client code as the actual implementation of the object is hidden from the client code. This can make it more difficult to test the client code's interaction with the object.

To mitigate these downsides, it's important to use opaque pointers judiciously, and to provide clear and comprehensive documentation of the library's interface and memory management. It's also important to consider other options, such as PIMPL or Handle-Body idiom, and choose the one that best fits the needs of the project.

In conclusion, opaque pointers are a powerful technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. However, it's important to use them judiciously and to be aware of the potential downsides, such as performance overhead, memory management, and testing. It's also important to consider other options and choose the one that best fits the needs of the project.

Questions? Keep them coming. We would love to answer.

This article is contributed by **Aashish Barnwal**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

70

Related Articles

1. Difference between passing pointer to pointer and address of pointer to any function

2. C++ Pointer To Pointer (Double Pointer)

References in C++

Difficulty Level : Medium • Last Updated : 30 Mar, 2023

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

Also, we can define a reference variable as a type of variable that can act as a reference to another variable. '&' is used for signifying the address of a variable or any memory. Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.

Prerequisite: Pointers in C++

Syntax:


```
data_type &ref = variable;
```

Example:

C++

```
// C++ Program to demonstrate
// use of references
#include <iostream>
```

```
using namespace std;

int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

Output:

```
x = 20
ref = 30
```

Applications of Reference in C++

There are multiple applications for references in C++, a few of them are mentioned below:

1. Modify the passed parameters in a function
2. Avoiding a copy of large structures
3. In For Each Loop to modify all objects
4. For Each Loop to avoid the copy of objects

1. Modify the passed parameters in a function:

If a function receives a reference to a variable, it can modify the value of the variable. For example, the following program variables are swapped using references.

Example:

C++

```
// C++ Program to demonstrate
// Passing of references as parameters
#include <iostream>
using namespace std;
```

```
// Function having parameters as
// references
void swap(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

// Driver function
int main()
{
    // Variables declared
    int a = 2, b = 3;

    // function called
    swap(a, b);

    // changes can be seen
    // printing both variables
    cout << a << " " << b;
    return 0;
}
```

Output

```
3 2
```

2. Avoiding a copy of large structures:

Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes a waste of CPU time and memory. We can use references to avoid this.

Example:

```
struct Student {
    string name;
    string address;
    int rollNo;
}

// If we remove & in below function, a new
// copy of the student object is created.
// We use const to avoid accidental updates
// in the function as the purpose of the function
// is to print s only.
```

```
void print(const Student &s)
{
    cout << s.name << "  " << s.address << "  " << s.rollNo
         << '\n';
}
```

3. In For Each Loop to modify all objects:

We can use references for each loop to modify all elements.

Example:

C++

```
// C++ Program for changing the
// values of elements while traversing
// using references
#include <iostream>
#include <vector>

using namespace std;

// Driver code
int main()
{
    vector<int> vect{ 10, 20, 30, 40 };

    // We can modify elements if we
    // use reference
    for (int& x : vect) {
        x = x + 5;
    }

    // Printing elements
    for (int x : vect) {
        cout << x << " ";
    }
    cout << '\n';

    return 0;
}
```

Output

```
15 25 35 45
```

4. For Each Loop to avoid the copy of objects:

We can use references in each loop to avoid a copy of individual objects when objects are large.

Example:

C++

```
// C++ Program to use references
// For Each Loop to avoid the
// copy of objects
#include <iostream>
#include <vector>

using namespace std;

// Driver code
int main()
{
    // Declaring vector
    vector<string> vect{ "geeksforgeeks practice",
                         "geeksforgeeks write",
                         "geeksforgeeks ide" };

    // We avoid copy of the whole string
    // object by using reference.
    for (const auto& x : vect) {
        cout << x << '\n';
    }

    return 0;
}
```

Output

```
geeksforgeeks practice
geeksforgeeks write
geeksforgeeks ide
```

References vs Pointers

Both references and pointers can be used to change the local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain. Despite the above similarities, there are the following differences between references and pointers.

1. A pointer can be declared as void but a reference can never be void For example

```
int a = 10;
void* aa = &a; // it is valid
void& ar = a;  // it is not valid
```

2. The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer, double-pointer, triple-pointer. Whereas, the reference variable has only one/single level of indirection. The following code reveals the mentioned points:

3. Reference variables cannot be updated.

4. Reference variable is an internal pointer.

5. Declaration of a Reference variable is preceded with the '&' symbol (but do not read it as "address of").

Example:

C++

```
// C++ Program to demonstrate
// references and pointers
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    // simple or ordinary variable.
    int i = 10;

    // single pointer
    int* p = &i;

    // double pointer
    int** pt = &p;

    // triple pointer
    int*** ptr = &pt;

    // All the above pointers differ in the value they store
    // or point to.
    cout << "i = " << i << "\t"
         << "p = " << p << "\t"
         << "pt = " << pt << "\t"
         << "ptr = " << ptr << '\n';

    // simple or ordinary variable
    int a = 5;
    int& S = a;
    int& S0 = S;
    int& S1 = S0;
```

```
    // All the references do not differ in their
    // values as they all refer to the same variable.
    cout << "a = " << a << "\t"
         << "S = " << S << "\t"
         << "S0 = " << S0 << "\t"
         << "S1 = " << S1 << '\n';

    return 0;
}
```

Output

```
i = 10    p = 0x7ffecfe7c07c    pt = 0x7ffecfe7c080    ptr = 0x7ffecfe7c088
a = 5    S = 5    S0 = 5    S1 = 5
```

Limitations of References

1. Once a reference is created, it cannot be later made to reference another object; it cannot be reset. This is often done with pointers.
2. References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
3. A reference must be initialized when declared. There is no such restriction with pointers.

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have the above restrictions and can be used to implement all data structures. References being more powerful in Java is the main reason Java doesn't need pointers.

Advantages of using References

1. ***Safer****:* Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
2. ***Easier to use:*** References don't need a dereferencing operator to access the value. They can be used like normal variables. The '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with the dot operator ('.'), unlike pointers where the arrow operator (->) is needed to access members.

Together with the above reasons, there are a few places like the copy constructor argument where a pointer cannot be used. Reference must be used to pass the argument in the copy constructor. Similarly, references must be used for overloading some operators like ++.

Exercise with Answers

Question 1 :

C++

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

int& fun()
{
    static int x = 10;
    return x;
}

int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

Output

```
30
```

Question 2

C++

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

int fun(int& x) { return x; }

int main()
{
    cout << fun(10);
    return 0;
}
```

Output:

```
./3337ee98-ae6e-4792-8128-7c879288221f.cpp: In function 'int main()':
./3337ee98-ae6e-4792-8128-7c879288221f.cpp:8:19: error: invalid
initialization of non-const reference of type 'int&' from an rvalue of type
'int'
```

```
      cout << fun(10);
                    ^
./3337ee98-ae6e-4792-8128-7c879288221f.cpp:4:5: note: in passing argument 1
of 'int fun(int&)'
 int fun(int& x) { return x; }
```

Question 3

C++

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

void swap(char*& str1, char*& str2)
{
    char* temp = str1;
    str1 = str2;
    str2 = temp;
}

int main()
{
    char* str1 = "GEEKS";
    char* str2 = "FOR GEEKS";
    swap(str1, str2);
    cout << "str1 is " << str1 << '\n';
    cout << "str2 is " << str2 << '\n';
    return 0;
}
```

Output

```
str1 is FOR GEEKS
str2 is GEEKS
```

Question 4

C++

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

int main()
{
    int x = 10;
    int* ptr = &x;
    int&* ptr1 = ptr;
}
```

Output:

```
./18074365-ebdc-4b13-81f2-cfc42bb4b035.cpp: In function 'int main()':
./18074365-ebdc-4b13-81f2-cfc42bb4b035.cpp:8:11: error: cannot declare
pointer to 'int&'
     int&* ptr1 = ptr;
```

Question 5

C++

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

int main()
{
    int* ptr = NULL;
    int& ref = *ptr;
    cout << ref << '\n';
}
```

Output:

```
timeout: the monitored command dumped core
/bin/bash: line 1:    34 Segmentation fault      timeout 15s ./372da97e-
346c-4594-990f-14edda1f5021 < 372da97e-346c-4594-990f-14edda1f5021.in
```

Question 6

C++

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

int& fun()
{
    int x = 10;
    return x;
}

int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

Output

```
0
```

Related Articles

- Pointers vs References in C++
- When do we pass arguments by reference or pointer?
- Can references refer to an invalid location in C++?
- Passing by pointer Vs Passing by Reference in C++

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

291

Related Articles

1. lvalues references and rvalues references in C++ with Examples
2. Can References Refer to Invalid Location in C++?
3. Default Assignment Operator and References in C++
4. Pointers and References in C++
5. Overloads of the Different References in C++
6. Pointers vs References in C++
7. C++ Error - Does not name a type
8. Execution Policy of STL Algorithms in Modern C++
9. C++ Program To Print Pyramid Patterns
10. Jagged Arrays in C++

Previous Next

'this' pointer in C++

Difficulty Level : Medium • Last Updated : 09 Aug, 2019

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as 'this'.
The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).
For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *' (see this GFact)

In the early version of C++ would let 'this' pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.
C++ lets object destroy themselves by calling the following code :

```
delete this;
```

As Stroustrup said 'this' could be the reference than the pointer, but the reference was not present in the early version of C++. If 'this' is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

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

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Output:

```
  x = 20
```

For constructors, initializer list can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
 x = 10 y = 20
```

Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

class Test
{
private:
  int x;
public:
  Test(int x = 0) { this->x = x; }
  void change(Test *t) { this = t; }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj(5);
  Test *ptr = new Test (10);
  obj.change(ptr);
  obj.print();
  return 0;
}
```

Question 2

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

Question 3

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

class Test
{
private:
  int x;
  int y;
public:
  Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test setX(int a) { x = a; return *this; }
  Test setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1;
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

Question 4

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  void setX(int a) { x = a; }
  void setY(int b) { y = b; }
  void destroy()  { delete this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj;
  obj.destroy();
  obj.print();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

248

Related Articles

1. C - Pointer to Pointer (Double Pointer)
2. Difference between passing pointer to pointer and address of pointer to any function
3. C++ Pointer To Pointer (Double Pointer)
4. Pointer to an Array | Array Pointer
5. What is a Pointer to a Null pointer
6. Difference between Dangling pointer and Void pointer
7. How to declare a pointer to a function?
8. Declare a C/C++ function returning pointer to array of integer pointers
9. Type of 'this' Pointer in C++
10. Multidimensional Pointer Arithmetic in C/C++

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Smart Pointers in C++

Difficulty Level : Medium • Last Updated : 16 Mar, 2023

Prerequisite: Pointers in C++

Pointers are used for accessing the resources which are external to the program – like heap memory. So, for accessing the heap memory (if anything is created inside heap memory), pointers are used. When accessing any external resource we just use a copy of the resource. If we make any changes to it, we just change it in the copied version. But, if we use a pointer to the resource, we'll be able to change the original resource.

Problems with Normal Pointers

Some Issues with normal pointers in C++ are as follows:

- **Memory Leaks:** This occurs when memory is repeatedly allocated by a program but never freed. This leads to excessive memory consumption and eventually leads to a system crash.
- **Dangling Pointers:** A dangling pointer is a pointer that occurs at the time when the object is de-allocated from memory without modifying the value of the pointer.
- **Wild Pointers:** Wild pointers are pointers that are declared and allocated memory but the pointer is never initialized to point to any valid object or address.
- **Data Inconsistency:** Data inconsistency occurs when some data is stored in memory but is not updated in a consistent manner.
- **Buffer Overflow:** When a pointer is used to write data to a memory address that is outside of the allocated memory block. This leads to the corruption of data which can be exploited by malicious attackers.

Example:

C++

```
// C++ program to demonstrate working of a Pointers
#include <iostream>
using namespace std;

class Rectangle {
private:
    int length;
    int breadth;
};

void fun()
{
    // By taking a pointer p and
    // dynamically creating object
    // of class rectangle
    Rectangle* p = new Rectangle();
}

int main()
{
    // Infinite Loop
    while (1) {
        fun();
    }
}
```

Output:

```
Memory limit exceeded
```

Explanation: In function *fun*, it creates a pointer that is pointing to the *Rectangle* object. The object *Rectangle* contains two integers, *length,* and *breadth*. When the function *fun* ends, p will be destroyed as it is a local variable. But, the memory it consumed won't be deallocated because we forgot to use *delete p;* at the end of the function. That means the memory won't be free to be used by other resources. But, we don't need the variable anymore, we need the memory.

In function *main*, *fun* is called in an infinite loop. That means it'll keep creating *p*. It'll allocate more and more memory but won't free them as we didn't deallocate it. The memory that's wasted can't be used again. Which is a memory leak. The entire *heap* memory may become useless for this reason.

Smart Pointers

As we've known unconsciously not deallocating a pointer causes a memory leak that may lead to a crash of the program. Languages Java, C# has *Garbage Collection Mechanisms* to smartly deallocate unused memory to be used again. The programmer doesn't have to worry about any memory leaks. C++ comes up with its own mechanism that's *Smart Pointer*. When the object is destroyed it frees the memory as well. So, we don't need to delete it as Smart Pointer does will handle it.

A *Smart Pointer* is a wrapper class over a pointer with an operator like ***** and **->** overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.

The idea is to take a class with a pointer, destructor, and overloaded operators like ***** and **->**. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or the reference count can be decremented).

Example:

C++

```
// C++ program to demonstrate the working of Smart Pointer
#include <iostream>
using namespace std;

class SmartPtr {
    int* ptr; // Actual pointer
public:
    // Constructor: Refer https:// www.geeksforgeeks.org/g-fact-93/
    // for use of explicit keyword
    explicit SmartPtr(int* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    int& operator*() { return *ptr; }
};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
```

```
    cout << *ptr;

    // We don't need to call delete ptr: when the object
    // ptr goes out of scope, the destructor for it is automatically
    // called and destructor does delete ptr.

    return 0;
}
```

Output

```
20
```

Difference Between Pointers and Smart Pointers

Pointer	Smart Pointer
A pointer is a variable that maintains a memory address as well as data type information about that memory location. A pointer is a variable that points to something in memory.	It's a pointer-wrapping stack-allocated object. Smart pointers, in plain terms, are classes that wrap a pointer, or scoped pointers.
It is not destroyed in any form when it goes out of its scope	It destroys itself when it goes out of its scope
Pointers are not so efficient as they don't support any other feature.	Smart pointers are more efficient as they have an additional feature of memory management.
They are very labor-centric/manual.	They are automatic/pre-programmed in nature.

> ***Note:*** *This only works for int. So, we'll have to create Smart Pointer for every object?* ***No****, there's a solution,* ***Template****. In the code below as you can see T can be of any type.*

Example:

C++

```
// C++ program to demonstrate the working of Template and
// overcome the issues which we are having with pointers
#include <iostream>
using namespace std;

// A generic smart pointer class
template <class T> class SmartPtr {
    T* ptr; // Actual pointer
public:
    // Constructor
    explicit SmartPtr(T* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    T& operator*() { return *ptr; }

    // Overloading arrow operator so that
    // members of T can be accessed
    // like a pointer (useful if T represents
    // a class or struct or union type)
    T* operator->() { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}
```

Output:

```
20
```

Note: *Smart pointers are also useful in the management of resources, such as file handles or network sockets.*

Types of Smart Pointers

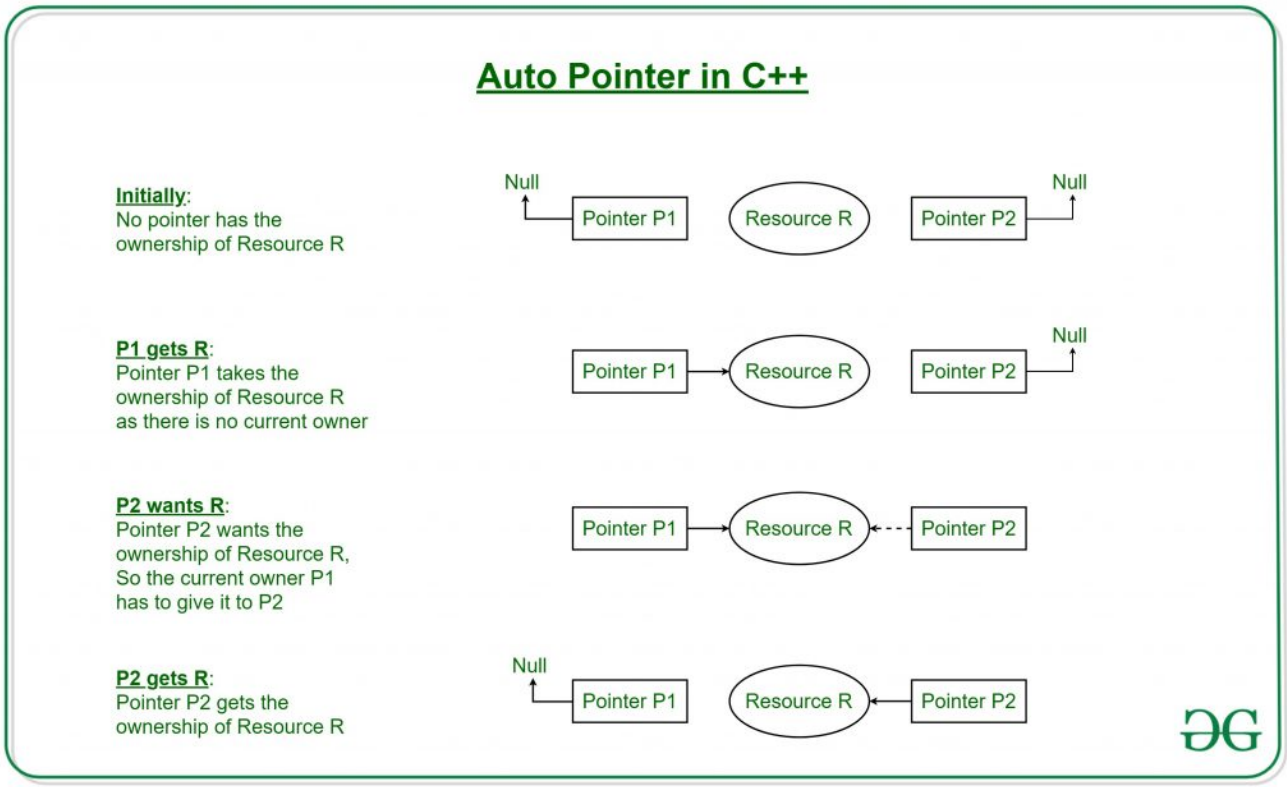
C++ libraries provide implementations of smart pointers in the following types:

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

auto_ptr

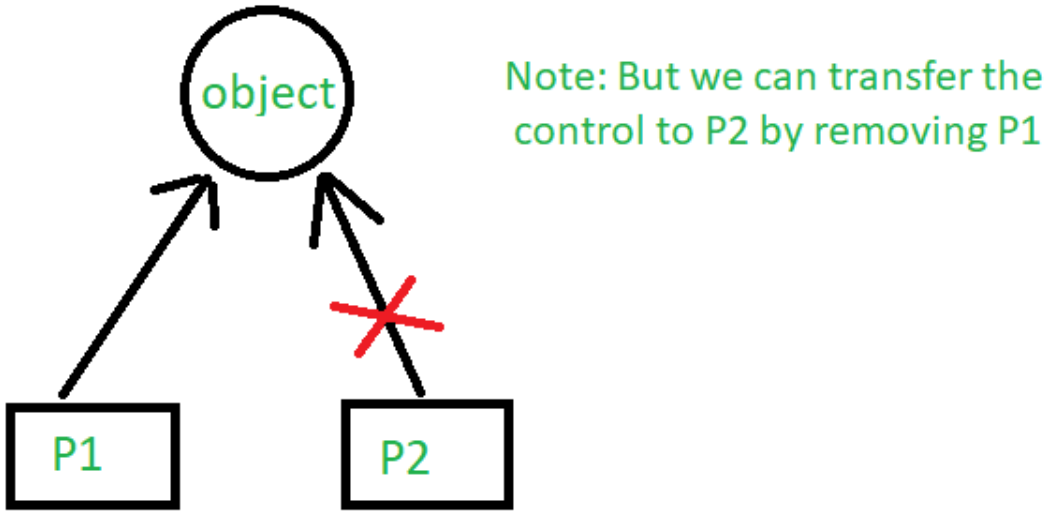
Using auto_ptr, you can manage objects obtained from new expressions and delete them when auto_ptr itself is destroyed. When an object is described through auto_ptr it stores a pointer to a single allocated object.

> ***Note:*** *This class template is deprecated as of C++11. unique_ptr is a new facility with a similar functionality, but with improved security.*



unique_ptr

unique_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer.



Example:

C++

```
// C++ program to demonstrate the working of unique_ptr
// Here we are showing the unique_pointer is pointing to P1.
// But, then we remove P1 and assign P2 so the pointer now
// points to P2.

#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
// --\/ Smart Pointer
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    // unique_ptr<Rectangle> P2(P1);
```

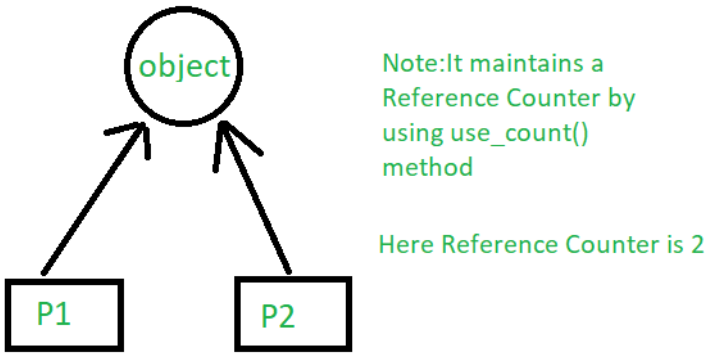
```
    unique_ptr<Rectangle> P2;
    P2 = move(P1);

    // This'll print 50
    cout << P2->area() << endl;

    // cout<<P1->area()<<endl;
    return 0;
}
```

Output

```
50
50
```

shared_ptr

By using *shared_ptr* more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using the ***use_count()* method.**



C++

```
// C++ program to demonstrate the working of shared_ptr
// Here both smart pointer P1 and P2 are pointing to the
// object Addition to which they both maintain a reference
// of the object
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
```

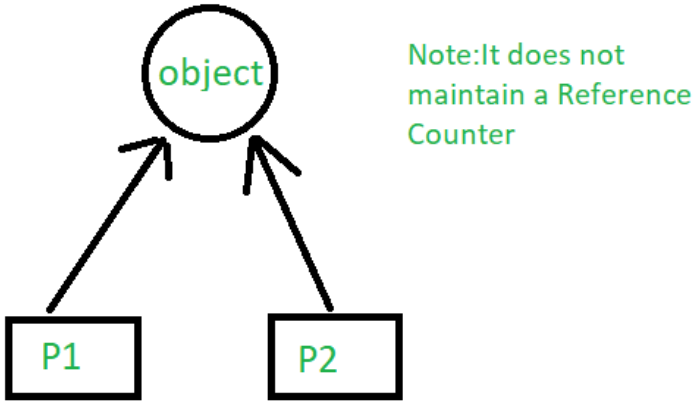
```
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
    //---\/ Smart Pointer
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));
    // This'll print 50
    cout << P1->area() << endl;

    shared_ptr<Rectangle> P2;
    P2 = P1;

    // This'll print 50
    cout << P2->area() << endl;

    // This'll now not give an error,
    cout << P1->area() << endl;

    // This'll also print 50 now
    // This'll print 2 as Reference Counter is 2
    cout << P1.use_count() << endl;
    return 0;
}
```

Output

```
50
50
50
2
```

weak_ptr

Weak_ptr is a smart pointer that holds a non-owning reference to an object. It's much more similar to shared_ptr except it'll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock.**



C++

```
// C++ program to demonstrate the working of weak_ptr
// Here both smart pointer P1 and P2 are pointing to the
// object Addition to which they both does not maintain
// a reference of the object
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
    //---\/ Smart Pointer
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));

    // This'll print 50
    cout << P1->area() << endl;

    auto P2 = P1;
```

```
    // P2 = P1;

    // This'll print 50
    cout << P2->area() << endl;

    // This'll now not give an error,
    cout << P1->area() << endl;

    // This'll also print 50 now
    // This'll print 2 as Reference Counter is 2
    cout << P1.use_count() << endl;
    return 0;
}
```

Output

```
50
50
50
2
```

C++ libraries provide implementations of smart pointers in the form of auto_ptr, unique_ptr, shared_ptr, and weak_ptr

152

Related Articles

1. Difference between constant pointer, pointers to constant, and constant pointers to constants
2. Trie Data Structure using smart pointer and OOP in C++
3. What are Wild Pointers? How can we avoid?
4. Declare a C/C++ function returning pointer to array of integer pointers
5. C++ Pointers
6. Computing Index Using Pointers Returned By STL Functions in C++
7. Applications of Pointers in C/C++
8. C++ Program to compare two string using pointers

Pointers vs References in C++

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

Prerequisite: Pointers, References

C and C++ support pointers, which is different from most other programming languages such as Java, Python, Ruby, Perl and PHP as they only support references. But interestingly, C++, along with pointers, also supports references.

On the surface, both references and pointers are very similar as both are used to have one variable provide access to another. With both providing lots of the same capabilities, it's often unclear what is different between these mechanisms. In this article, I will try to illustrate the differences between pointers and references.

Pointers: A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced with the * operator to access the memory location it points to.

References: A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.
A reference can be thought of as a constant pointer (not to be confused with a pointer to a

constant value!) with automatic indirection, i.e., the compiler will apply the * operator for you.

```
int i = 3;

// A pointer to variable i or "stores the address of i"
int *ptr = &i;

// A reference (or alias) for i.
int &ref = i;
```

Differences:

1. **Initialization:** A pointer can be initialized in this way:

```
int a = 10;
int *p = &a;
// OR
int *p;
p = &a;
```

We can declare and initialize pointer at same step or in multiple line.

2. While in references,

```
int a = 10;
int &p = a; // It is correct
// but
int &p;
p = a; // It is incorrect as we should declare and initialize references at
single step
```

NOTE: This difference may vary from compiler to compiler. The above difference is with respect to Turbo IDE.

3. **Reassignment:** A pointer can be re-assigned. This property is useful for the implementation of data structures like a linked list, a tree, etc. See the following example:

```
int a = 5;
int b = 6;
int *p;
p = &a;
p = &b;
```

4. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

```
int a = 5;
int b = 6;
int &p = a;
int &p = b; // This will throw an error of "multiple declaration is not
allowed"

// However it is valid statement,
int &q = p;
```

5. **Memory Address:** A pointer has its own memory address and size on the stack, whereas a reference shares the same memory address with the original variable but also takes up some space on the stack.

```
int &p = a;
cout << &p << endl << &a;
```

6. **NULL value:** A pointer can be assigned NULL directly, whereas a reference cannot be. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into an exception situation.

7. **Indirection:** You can have a pointer to pointer (known as a double pointer) offering extra levels of indirection, whereas references only offer one level of indirection. For example,

```
In Pointers,
int a = 10;
int *p;
int **q; // It is valid.
p = &a;
q = &p;

// Whereas in references,
int &p = a;
int &&q = p; // It is reference to reference, so it is an error
```

8. **Arithmetic operations:** Various arithmetic operations can be performed on pointers, whereas there is no such thing called Reference Arithmetic (however, you can perform pointer arithmetic on the address of an object pointed to by a reference, as in &obj + 5).

Tabular form of difference between References and Pointers in C++

	References	Pointers
Reassignment	The variable cannot be reassigned in Reference.	The variable can be reassigned in Pointers.
Memory Address	It shares the same address as the original variable.	Pointers have their own memory address.
Work	It is referring to another variable.	It is storing the address of the variable.
Null Value	It does not have null value.	It can have value assigned as null.
Arguments	This variable is referenced by the method pass by value.	The pointer does it work by the method known as pass by reference.

When to use What

The performances are exactly the same as references are implemented internally as pointers. But still, you can keep some points in your mind to decide when to use what:

- Use references:
  - In function parameters and return types.
- Use pointers:
  - If pointer arithmetic or passing a NULL pointer is needed. For example, for arrays (Note that accessing an array is implemented using pointer arithmetic).
  - To implement data structures like a linked list, a tree, etc. and their algorithms. This is so because, in order to point to different cells, we have to use the concept of pointers.

Quoted in C++ FAQ Lite: Use references when you can, and pointers when you have to. References are usually preferred over pointers whenever you don't need "reseating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value needs a "sentinel" reference – a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the "nullptr" value this special significance (references must always alias objects, not a dereferenced null pointer).

Related Article:

When do we pass arguments as Reference or Pointers?

This article is contributed by **Rishav Raj**. 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 you want to share more information about the topic discussed above.

217

Related Articles

1. lvalues references and rvalues references in C++ with Examples
2. Pointers and References in C++
3. Difference between constant pointer, pointers to constant, and constant pointers to constants
4. Can References Refer to Invalid Location in C++?
5. Default Assignment Operator and References in C++
6. C++ | References | Question 1
7. C++ | References | Question 6
8. C++ | References | Question 6
9. C++ | References | Question 4
10. C++ | References | Question 6

Previous Next

Article Contributed By :

GeeksforGeeks