C++ Pointers

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax:

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

How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.
- The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

Pointers in C++

```
// C++ program to illustrate Pointers
#include <iostream>
using namespace std;
void main_program()
{
   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
   cout << "Value at ptr = " << ptr << "\n";
   cout << "Value at var = " << var << "\n";
   cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
```

```
int main()
{
  main_program();
  return 0;
}
```

Output

```
Value at ptr = 0x7ffe454c08cc
Value at var = 20
Value at *ptr = 20
```

References and Pointers

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

- Call-By-Value
- Call-By-Reference with a Pointer Argument
- Call-By-Reference with a Reference Argument

```
// C++ program to illustrate call-by-methods
#include <iostream>
using namespace std;
// Pass-by-Value
int square1(int n)
{
   // Address of n in square1() is not the same as n1 in
   // main()
   cout << "address of n1 in square1(): " << &n << "\n";

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

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

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

// Pass-by-Reference with Reference Arguments
void square3(int& n)
```

```
{
    // Address of n in square3() is the same as n3 in main()
    cout << "address of n3 in square3(): " << &n << "\n";

    // Implicit de-referencing (without '*')
    n *= n;
}
void main_program()
{
    // Call-by-Value
    int n1 = 8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

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

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

Output

```
address of n1 in main(): 0x7fffa7e2de64
address of n1 in square1(): 0x7fffa7e2de4c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7fffa7e2de68
address of n2 in square2(): 0x7fffa7e2de68
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7fffa7e2de6c
address of n3 in square3(): 0x7fffa7e2de6c
Square of n3: 64
```

Change reflected in n3: 64

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function. If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

Function pointers in C
Pointer to a Function
Array Name as Pointers
An array name contains the address of the first element of the array which acts like a constant pointer. It means, the address stored in the array name can't be changed. For example, if we have an array named val then val and &val[0] can be used interchangeably.

```
// C++ program to illustrate Array Name as Pointers
#include <ioistream>
using namespace std;
void main_program()
{
   // Declare an array
   int val[3] = { 5, 10, 20 };

   // declare pointer variable
   int* ptr;

   // Assign the address of val[0] to ptr
   // We can use ptr=&val[0];(both are same)
   ptr = val;
   cout << "Elements of the array are: ";
   cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
}
// Driver program
int main() { main_program(); }
```

Output
```
Elements of the array are: 5 10 20
```
Representation of data in memory

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