Introduction to Pointers

Pointers are one of the most important aspects of C++. Pointers are another type of variable in CPP, and these variables store addresses of other variables.

While creating a pointer variable, we need to mention the type of data whose address is stored in the pointer. e.g., to create a pointer that stores the address of an integer, we need to write:

int* p;

This means that p will contain the address of an integer. So, if a pointer is going to store the address of datatype X, it will be declared like this:

int* p;

Address of Operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as the *address-of operator*.

Example:

cout << (&var) << endl;

This would print the address of variable *var*; by preceding the name of the variable var with the *address-of operator* (&), we are no longer printing the content of the variable itself but its address.

Example:

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

int main() {
    int i = 10;
    int * p;
    p = & i;
    cout << "Address of the variable i is " << p << endl;
    cout << "Address of the pointer p is " << & p;
    return 0;
}
```

Output:
Address of the variable i is 0x7fff32eb4bb4
Address of the pointer p is 0x7fff32eb4bb8

Here, we have an integer i and an integer pointer p. The address of(&) operator is used to address i in p that returns the variable's address. e.g., &i will give us the address of variable i.

Dereference Operator

As just seen, a variable that stores the address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store.

An exciting property of pointers is that they can access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by."

The reference and dereference operators are thus complementary:

- & is the address-of operator and can be read simply as "address of."
- * is the dereference operator and can be read as "value pointed to by."

Note: The asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier) and should not be confused with the dereference operator seen above, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.

Example:

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

int main() {
   int firstvalue = 5, secondvalue = 15;
   char thirdvalue = 'a';
   int * p1, * p2;
   char * p3;
   p1 = & firstvalue; // p1 = address of firstvalue
   p2 = & secondvalue; // p2 = address of secondvalue
   p3 = & thirdvalue; // p3 = address of thirdvalue
   * p1 = 10; // value pointed to by p1 = 10
   * p2 = * p1; // value pointed to by p2 = value pointed to by p1
   p1 = p2; // p1 = p2 (value of pointer is copied)
   * p1 = 20; // value pointed to by p1 = 20
   * p3 = 'b'; // value pointed to by p3 = ‘b ’
   cout << "firstvalue is " << firstvalue << endl;
   cout << "secondvalue is " << secondvalue << endl;
   cout << "thirdvalue is " << thirdvalue << endl;
   return 0;
}
```

Output:
firstvalue is 10
secondvalue is 20
thirdvalue is b

Note: While solving pointers questions, you should use pen and paper and draw better ideas.

Pointer Arithmetic

Arithmetic operations on pointers behave differently than they do on simple data types we studied earlier. Only addition and subtraction operations are allowed; the others aren't allowed on pointers. But both addition and subtraction have slightly different behavior with pointers, according to the size of the data type to which they point.

For example, char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these depends on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Suppose now that we define three-pointers:

```
char *mychar;
short *myshort;
long *mylong;
```

and they point to the memory locations 1000, 2000, and 3000, respectively.

Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.

This is applicable both when adding and subtracting any number to a pointer. It would happen the same if we wrote:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same applies to the decrement operator).

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables related to each other, such as the same array of elements, then p1 and p2 can be meaningfully compared.