Coding Ninjas

C++ Foundation with Data Structures

Topic : Pointers

Address of Operator (&)

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

```
cout << (&var) << endl;
```

This would print 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.

What are Pointers?

Pointers are one of the most important aspects of C++. Pointers are another type of variables 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. in order to create a pointer which stores address of an integer, we need to write:

```
int* p;
```

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

```
X* p;
```

Now let's say we have an integer i & an integer pointer p, we will use addressof(&) operator in order to put address of i in p. Address of operator:& as studied abpve is a unary operator which returns address of a variable. e.g. &i will give us address of variable i. Here is the code to put address of i in p.

```
int i = 10;
int* p;
p = &i;
```

Dereference Operator

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

Coding Ninjas

An interesting property of pointers is that they can be used to 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".

Therefore, following with the values of the previous example, consider the following statement:

int a = *p;

So in this assignment we are assigning value pointed to by pointer p(i.e. value of to int i) to int variable a.

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"

Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with *.

Note that 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.

Following

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

int main ()
{
 int firstvalue = 5, secondvalue = 15;
char thirdvalue = ‘a’;
 int * p1, * p2;
char *p3;
```

```
firstvalue is 10
secondvalue is
20
thirdvalue is b
```

```
 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 << '\n';
 cout << "secondvalue is " << secondvalue << '\n';
 cout << "thirdvalue is " << thirdvalue << '\n';
 return 0;
}
```

Note: While solving pointers question, you should use pen and paper and draw things to get better idea.

Null Pointer

Consider the following statement –

int *p;

Here we have created a pointer variable that contains garbage value. In order to dereference the pointer, we will try reading out the value at the garbage stored in the pointer. This will lead to unexpected results or segmentation faults. Hence we should never leave a pointer uninitialized and instead initialize it to NULL, so as to avoid unexpected behavior.

int *p = NULL; // NULL is a constant with a value 0
int *q = 0;   // Same as above

So now if we try to dereference the pointer we will get segmentation fault as 0 is a reserved memory address.

Pointer Arithmetic

Arithmetic operations on pointers behave differently than they do on simple data types we studied earlier. Only addition and subtraction operations are

Coding Ninjas

allowed; the others aren't allowed on pointers. But both addition and subtraction have a 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 being dependent 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 in this compiler:

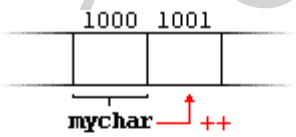
```
char *mychar;
short *myshort;
long *mylong;
```

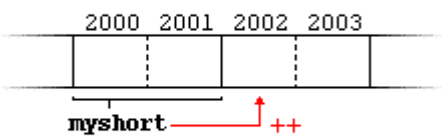
and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

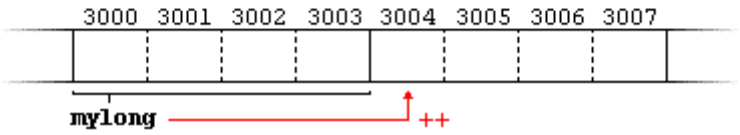
Therefore, if we write:

```
++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.







Coding Ninjas

This is applicable both when adding and subtracting any number to a pointer. It would happen exactly 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 being applicable also to the decrement operator):

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

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

Pointer and Arrays

Pointers and arrays are intricately linked. An Array is actually a pointer that points to the first element of the array. Because the array variable is a pointer, you can dereference it, which returns array element 0.

Consider the following code –

```
int a[] = {1,2,3,4,5};
int *b = &a[0];
cout << b << endl;
cout << a << endl;
cout << *b << endl; // This will print 1
```

Both b and a will print same same address as they are referring to first element of the array.
Also in arrays - *a[i]* is same as **(a + i)*.

Coding Ninjas

Consider an example for the same–

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

int main(){

  int a[5] = {1,2,3,4,5};
  cout << *(a + 2) << endl;

}
```

Output:

3

Differences between arrays and pointers:

1. **the sizeof operator:**
   sizeof(array) returns the amount of memory used by all elements in array whereas sizeof(pointer) only returns the amount of memory used by the pointer variable itself.

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

int main(){

  int a[5] = {1,2,3,4,5};
  int *b = &a[0];
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
}
```

Output :

```
20
8  // Size of pointer is compiler dependent. Here it is 8.
```

2. **the & operator**
   In the example above &a is an alias for &a[0] and returns the address of the first element in array
   &b returns the address of pointer.

Coding Ninjas

3. **Pointer variable can be assigned a value whereas array variable cannot be.**

```
int a[10];
int *p;
p=a;        //legal
a=p;        //illegal
```

4. **Arithmetic on pointer variable is allowed, but not allowed on array variable.**

```
p++;        //Legal
a++;        //illegal
```

Double Pointer

As we know by now that pointers are variables that store address of other variables, so we can create variables that store address of pointer itself i.e. a pointer to a pointer. Let's see how can we create one.

```
int a = 10;
int *p = &a;
int **q = &p;
```

Here q is a pointer to a pointer i.e. a double pointer,as indicated by **.

Consider the following code for better understanding –

```
#include<iostream>
using namespace std;
int main(){
 int a = 10;
 int *p = &a;
 int **q = &p;

// Next three statements will print same value i.e. address of a
 cout << &a << endl;
 cout << p << endl;
 cout << *q << endl;

// Next two statements will print same value i.e. address of p
 cout << &p << endl;
 cout << q << endl;
```

Coding Ninjas

```
// Next two statements will print same value i.e. value of a which is 10
  cout << a << endl;
  cout << *p << endl;
  cout << **q << endl;
 }
```

Void Pointer

A void pointer is a generic pointer, it has no associated type with it. A void pointer can hold address of any type and can be typcasted to any type. Void pointer is declared normally the way we do for pointers.

```
void *ptr;
```

This statement will create a void pointer.

Example:

```
void *v;
int *i;
int ivar;
char chvar;
float fvar;
v = &ivar; // valid
v = &chvar; //valid
v = &fvar; // valid
i = &ivar; //valid
i = &chvar; //invalid
i = &fvar; //invalid
```

Thus we can use void pointer to store address of any variable.

Coding Ninjas