References

Chapter 9 – Pointers

Starting Out With C++. From Control Structures through Objects (Eighth Edition)

The operator & and * are very useful when trying to obtain the address of a variable or the content (or value) of an address.

The & 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:

```
int ii = 0;

pp = &ii;
```

The first statement declares ii as an int variable and the second would assign the address of ii to pp. If we use cout to display pp, a 4-byte address (i.e. a hexadecimal number like 0x1122cc) is printed.

The * operator

The **dereference operator** *, on the other hand, reverse the operation of &. Consider the following code fragment:

```
int ii = 0;

pp = &ii;

int ii2 = *pp;
```

The operator * dereferences the address pp and returns the value, i.e. 0, pointed by the address. The ii2 will have an initial value 0 after it is declared.

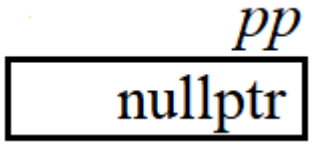
A pointer is a special type of variable which holds an address as its value. The variable pp above is a pointer which can point to an int variable. Before a pointer is used, it has to be declared first. If we use the code above as an example, pp can be declared like this:

```
int *pp;
```

or

```
int *pp = nullptr;
```

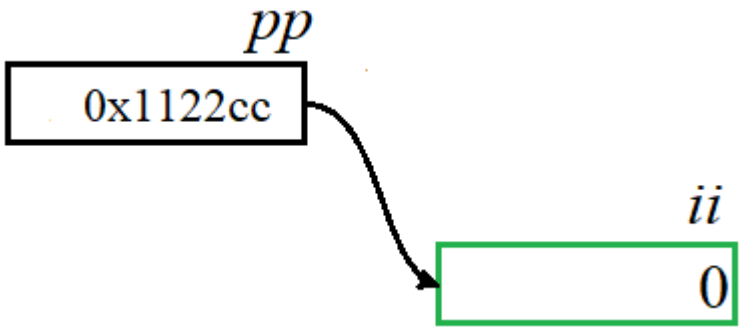
Once the statement is executed, a 4-byte memory box is allocated for the variable pp, and it can only point to an int type data.



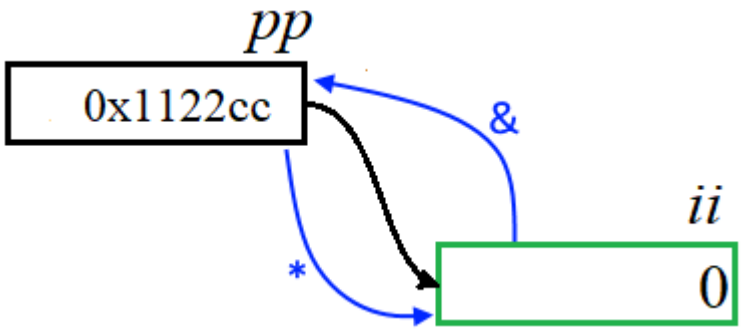
Just like what is shown in the beginning of this article, at a later time pp may be assign a address value like the following:

pp = ⅈ

After doing this, the address of ii is stored at pp. The following figure shows the relationship between pp, ii and their values.

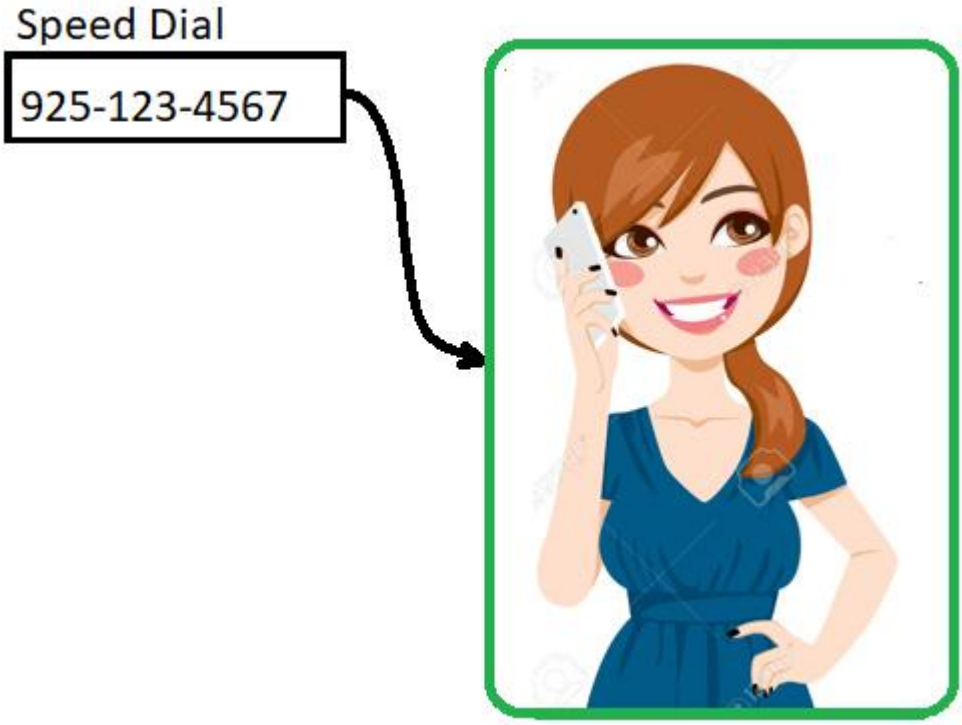


We can add two blue lines to indicate the operations of & and *. Applying & to ii gives us the address of ii which is the value of pp, and applying * to pp gives us the values pointed by the address, which is value of ii.



Analogy

Just like the address 0x1122cc points to the value 0, and the pointer variable *pp* is holding the address, a phone number can point to a person, and a speed dial is holding the phone number. When someone clicks the button of the speed dial, the stored phone number is dialed and the person owning this phone can be reached.



The three objects, i.e. person, the phone number, and the speed dial, in the figure above can be analogous to the value 0, the address, and the pointer, respectively. The speed dial stores the phone number and the phone number can reach the person. In a similar way, the pointer stores the address, and the address can reach the value.

	C++ Term	**Analogy**
Content	value 0	the person who is calling
Address	address	phone number
Pointer	pointer	speed dial

Exercise 1

(1) Run the following code segment and see what are displayed.
(2) Draw the figures for each line of the code to demonstrate that you fully understand the meanings of these statements.

```
int *pp = nullptr;
int ii = 0;
pp = &ii;
int ii2 = *pp;

cout << "The value of pp: " << pp << endl;
cout << "The value of ii2: " << ii2 << endl;
```

Exercise 2

Predict the output.

```
int y = 4;
int * yPtr = &y;


int x = 5;
yPtr = &x;

cout << "*yPtr = " << *yPtr << endl;

*yPtr = 10;
cout << "x = " << x << endl;
```

A Table Comparing Pointer and Reference:

There are differences between pointer and reference. The following table gives a brief introduction to them. More details will be provided in the **Pointer vs Reference** section.

	Pointer	**Reference**
Is a variable?	Yes	No

Must be initialized when it is declared?	No	Yes
Can be re-assigned?	Yes	No

Discussion:

How do we know that a variable is a pointer or just a regular variable? When a variable is declared by prefixing a * operator, we know that it is a pointer. For example we can interpret the following declaration

```
int *pp;
```

in this way:

Since "the content of pp" is an integer, we know that pp is a pointer which points to an integer variable.

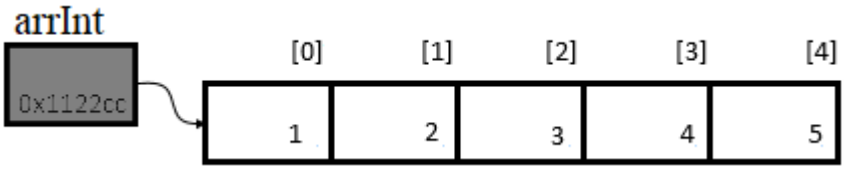
Pointer and array

After the statements:

```
const int SIZE = 5;

int arrInt[SIZE] = {1, 2, 3, 4, 5};
```

is executed, 20 bytes of memory is allocated for storing the five int values of the array.



The name arrInt is actually an address which points to the first element of the array. In a sense the array name is a pointer. (That is the reason why the array name can be passed to function as a parameter, and the function can modify the content in the array. If we declare a integer pointer like:

```
int * pp = nullptr;
```

we can assign the value (an address) of arrInt to it in the following way.

```
pp = arrInt;
```

In conclusion, the following two declarations give arrInt and pp very similar meaning. (The major difference is that pp can be resigned while arrInt cannot.)

```
int arrInt[SIZE];
int * pp;
```

This suggests that a prototype like:

```
void func1(int []) ;
```

can be replaced by

```
void func1(int *);
```

And in a similar way a prototype like:

```
void func1(int *[]) ;
```

can be replaced by

```
void func1(int **);
```

Pointer arithmetic

Using the same array arrInt from last section, we can get the address of each element in either one of these ways:

```
&arrInt[0], &arrInt[1], &arrInt[2] …     or
```

```
arrInt, arrInt+1, arrInt+2 …
```

The second approach, i.e. arrInt + n, is called pointer arithmetic. The C++ language allows us to perform integer addition or subtraction operations on pointers. If arrInt is the name of an array or a pointer for an integer array, (arrInt + 1) is the address of the next integer in memory after arrInt. And (arrInt – 1) is the address of the previous integer before arrInt.

How many bytes are added depends on the data type of the array. This means that if arrInt is an integer array and the address is 0x001110, then the address of (arrInt + 1) is 0x001114. However if the data type is double, then 8, not 4, is added to the address.

Exercise 3

The address of the array arrInt is Predict the output. Can you see which part in the code is incorrect?

```
const int SIZE = 6;
int arrInt[SIZE] = {3, 5, 9, 2, 6, 8};

cout << endl << "Part 1" << endl;
for (int i = 0; i < SIZE; i++)
cout << arrInt + i << " ";

cout << endl << "Part 2" << endl;
for (int i = 0; i < SIZE; i++)
cout << * arrInt + i << " ";
```

Exercise 4

```
#include <iostream>
using namespace std;
/// The getMax() function returns the maximum
```

```
/// integer in the sequence of n integers pointed
/// by p.
int getMax(int *p, int n)
{
        int maxInt = 0;
        /// TO DO: Add your code here
       return maxInt;
}

int main()
{
        const int SIZE = 6;
        int arr[SIZE] = {3, 5, 9, 2, 6, 8};

        cout << getMax(arr, SIZE);
       return 0;
}
```

See also:

Pointer vs. Reference