Script for the vhb lecture

Programming in C++
Part 2

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

2 References and Pointers

For beginners in C++ programming, references and pointers are not easy to understand. They are in fact one of the most difficult to learn elements of this programming language.

2.1 Definition of pointer

A pointer is a variable whose value is the address of another variable. Since the pointer does not establish a direct connection to the actual data, it is also referred to as indirection or referencing.

Syntax: Type ***Name**;

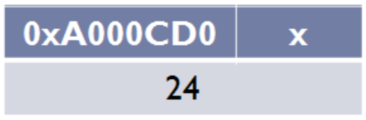
Example: int *ptr;   or   int* ptr;

You can declare pointers for any integral data type (int, char, long, short and so on) and also for objects (arrays, structures, classes, or instances).

To visualize pointers, let's first have a look at what a variable consists of. A variable is uniquely defined by four parameters:

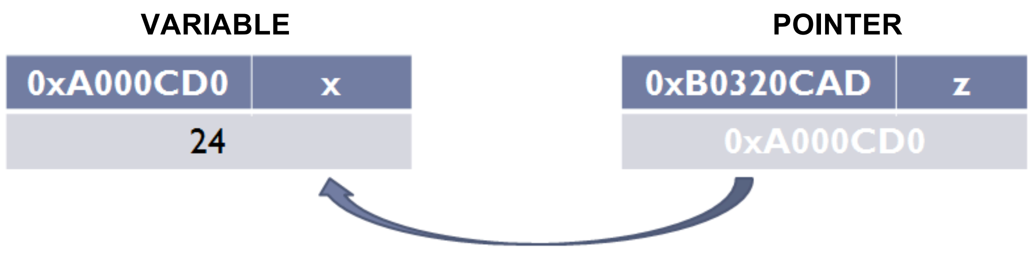
- Position (address)
- Size (required memory space)
- Name
- Content

Example of a typical variable.



In our example, the variable is named x, contains value 24 and is located at the memory address 0xA000CD0. The size of the variable is determined by the data type.

As mentioned before, a pointer is a variable that contains the address of another variable. Let's have a look at an example.



The pointer points to a variable.

Explanation: Dereferencing operator *

Example:

```
int x = 24, *z;
z = &x;
```

In this example, z is declared as a pointer variable and points to the x variable. The address of x is assigned to variable z. That means z contains the address of variable x.

Note:

- Pointers store **addresses**, whereas variables store **values**.
- Pointers which are not initialized contain random values like all other variables which are not initialized. This can crash a program.
- Variables and pointers that contain the address of the variable must be of the same data types.

2.2 Dereferencing of pointers

To dereference a pointer means to return the content of the memory position to which the pointer points.

Example: 2.2 Dereferencing a pointer

```
int x = 20;
int *ptrx = &x;    // the address of x is assigned to the pointer
int z = *ptrx;     // z is assigned the value of the variable pointing to ptrx (i.e. the value of x)
```

Explanation: The first line of this example declares an integer variable x and assigns value 20 to it. The next line declares a pointer to an integer value and assigns the address of variable x to the pointer. The *address-of operator* **&** is used. In the example, the address-of operator tells the compiler: “Pass the address of variable x to me but not the value of x itself“.
After the assignment, ptrx contains the memory address of x. You may need the value of the object to which ptrx assigns. Maybe the following code comes into your mind.

```
int z = ptrx; // WRONG!!!
```

Unfortunately, this is wrong and does not work. This is the reason: An attempt is made to assign a memory address to a regular variable. Then you get the following compiler error message *“Invalid conversion from int* to int“.* This is comprehensible since you are dealing with two completely different types of variables. Therefore, the *dereferencing operator* * has to be used to mark a variable as a pointer.

```
int z = *ptrx;
```

The dereferencing operator is virtually the opposite of the address-of operator. In this case, you do not want the actual value of ptrx since this is a memory address, but the value of the object to which the memory address refers. The dereferencing operator tells the compiler: “*Pass me the value of the object ptrx points to, not the actual value of ptrx“.*

Important:

To dereference a pointer means to determine the content of the memory area (of the object) to which the pointer points. The *operator is used to declare a pointer (int *x;) and to dereference it(int z = *x;).

Warning:

Pointers which are not initialized contain random values, just like all other variables which are not initialized. Attempting to use an uninitialized pointer can crash a program. In most cases, pointers are therefore initialized during their declaration.
Very often pointers are declared first and are initialized later in the course of the program. However, if the pointer is used before it is initialized, it will point to a random storage position. Changing this memory address can have many unforeseen consequences. Often the change of unknown memory only takes effect later so that the error can no longer be assigned. To be sure, you should therefore initialize a pointer with value 0 when declaring it.

```
Type *pointer = 0; // equal to      Type *pointer = NULL;
```

If you try to use a NULL pointer (any pointer set to NULL or 0), you receive an access violation error message from the operating system. Even though this does not sound particularly good, it is the lesser of two evils. It is much better to receive a message directly when an error occurs than to be confronted later with a problem that can no longer be assigned.

2.3 Pointer access options

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

int main()
{
    int x=18, *z;
    z=&x;
    cout << "Own address of pointer z: = " << &z << endl;
    cout << "Value of variable x: " << *z << endl;
    cout << " The address of variable x stored in z is" << z << endl;
    cout << "Address of variable x is: " << &x << endl;
    system("pause");
}
```

Explanation:

- int x = 18, *z; // Declaration of variable x and a pointer z
  z = &x; // z is assigned the address of x as value

- Access via name z // value of z is the address of x
  Example: cout << z; Output: 0x28ff44 // These addresses are only examples and are assigned by the compiler

- Access via dereferencing operator *z // Value of variable x
  Example: cout << *z; Output: 18

- Access via the address-of operator // Own address of the pointer
  Example: cout << &z; Output: 0x28ff66

2.4 Pointer to arrays

The use of pointers and the use of arrays are very similar. The differences are summarized below:

- A pointer has a memory location containing a value that can be used as an address.
- An array (i.e. array name) has no storage space in this sense. An array is a symbolic name for the address (= the beginning) of a memory area. If the name of an array variable is used without the index operator ([ ]), the address of the first array element is returned.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
int array[]={5,10,15,20,25};

cout << "The address of the first array is: " << array << endl;
// or
cout << "The address of the first array is: " << &array[0] << endl;

cout << "The value of the first array element is: " << *array << endl;
// or
cout << " The value of the first array element is: " << array[0] << endl;

system("pause");
}
```

You can access the individual elements of an array with the index operator ([ ]).

```
int array[]={5,10,15,20,25};
int aVariable = array[2]; // the value of the 3rd element (index starts at O!) is 15
```

We could do the same by using a pointer:

```
int array[]={5,10,15,20,25};
int *ptr = array;     // Declaration of a pointer with the name ptr and assignment of the
                      // address of the first field element
int aVariable = ptr[2]; // The value of the 3rd field element is assigned to the variable.
```

In this example, the memory location of the first array element is assigned to a pointer ptr. Please note that the pointer is of the data type *int* and a dereferencing operator (* symbol) is used to declare the pointer. After the assignment, the pointer contains the base address of the array in the memory and thus refers to it. With pointers to arrays, **no** dereferencing operator may be used when using the index.

2.5 Reference

A reference is a special type of pointer that allows you to treat a pointer like a regular object. References are declared using the address-of operator (&).

Syntax: Type &name; or Type& name;

The following example shows the declaration of the reference.

```
int x;
int &rx = x;       // equal to    int& rx = x;
rx = 10;           // has the same effect as    x = 10;
```

References have some peculiarities which make them inappropriate for many cases. For example, references cannot be declared and can only be assigned a value later. Therefore, **references must be initialized** right **at the time of the declaration**.

Note: The use of references (except for input and return parameters) should be largely avoided for the sake of comprehensibility.

2.6 Parameters as pointer or reference (call-by-reference)

When parameters are passed by "call-by-reference", a reference to the current parameter (which basically means the variable itself and not a copy) is used.

2.6.1 Parameter passing as reference

If you want to modify a passed object, you can use a **reference** to the object. The syntax of the call is the same as for passing parameters as a value (call-by-value). However, instead of working with a copy of the parameter, the original parameter is used. The changes made within the function have a direct effect on the original parameter. This means, no copy is created. If you do not wish to change the original and you do not work with call-by-value, you can pass an object as a reference to const. The parameter list could for instance be *const TYPE &object*. Within the function, this object can only be read. The compiler also validates this.

2.6.2 Parameter passing as pointer

The passing of parameters as pointer is basically the same as the passing as a reference. The difference between a pointer and a reference as parameter is first assumed to be very small. At first glance, we only see syntactic differences. In practice, it is often possible to replace one with the other. The difference is that you can always assign a different value to a pointer and thus refer to another variable. A reference, on the other hand, is the substitute for the passed variable and can no longer be referenced to another target variable after the passing of the parameter.

2.6.3 Example Call-By-Values vs Call-By-Reference

The following is a comparison between call-by-value and call-by-reference, as a reference and as a pointer.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
   int a = 10, b = 13;
   int *c, *d;
   c = &a;
   d = &b;
   cout << “a: “ << a << “, b: “ << b << endl;
   system("pause");
   return 0;
}
```

	call by value	**call by reference (as reference)**	**call by reference (as pointer)**

Function call	exchange(a, b);	exchange(a,b);	exchange(&a, &b);
			or
			exchange(c,d);

Function header and body	void exchange(int x, int y) {
	    int tmp = x;
	    x = y;
	    y = tmp;
	}
		void exchange(int &x,int &y) {
		    int tmp = x;
		    x = y;
		    y = tmp;
		}
			void exchange(int *x,int *y) {
			    int tmp = *x;
			    *x = *y;
			    *y = tmp;
			}

Output	a: 10, b: 13	a: 13, b: 10	a: 13, b: 10

2.7 Pointer to pointer

Apart from pointers to variables and functions, you will now learn about pointers to pointers.

Syntax: *TYPE* ***pname*;

Explanation: Pointer *pname* points to a pointer, which in turn points to a variable type *TYPE*.

Example: Pointer to pointer

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
   int x=10, y=10, *xptr, **ptrptr; // Declaration of variables x and y, of pointer xptr, and of the pointer to a pointer ptrptr

   cout << "x = " << x << endl; // Output of x as value

   xptr = &x; //  Pointer xptr is assigned the address of x as value
   *xptr = 20; // The variable which xptr points to is assigned value 20
   cout << "x = " << x << " and *xptr = " << *xptr <<endl;

   ptrptr = &xptr; // Pointer ptrptr is assigned the address of pointer xptr
   **ptrptr = 30; // Variable x is assigned value 30 via pointer **ptrptr
   cout << "x = " << x << " *xptr = "<< *xptr << " and **ptrptr = " << **ptrptr << endl;

   *ptrptr = &y; // The pointer now points to y
   **ptrptr = 40; // Variable y is assigned value 40, since ptrptr points to ptr and ptr is now set to y
   cout << "x = " << x << ", y= " << y << ", *xptr = " << *xptr << " and **ptrptr = " << **ptrptr << endl;
   system("pause");
}
```

2.8 Member access operators

The following member access operators are used for direct access to members (data members and functions) of a class: “.” and “->”. The difference between the two operators is that "." is applied to a variable of a class type, while "->" is used for pointers to class types.

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Birthday
{
   private:
      int year, month, day;
   public:
      Birthday(int j, int m, int t): year(j), month(m), day(t) {}       // Constructor with member initializer list
      Birthday() {}                                                     // Default constructor

      void setYear (int j) {  year = j; }
      void setMonth(int m) { month = m; }
      void setDay(int t) { day = t; }
      void displayData() { cout << day << "-" << month << "-" << year << endl; }
};
int main()
{
   Birthday Heinz;
   Birthday *Peter;

   Heinz.setYear(1964);
   Heinz.setMonth(1);
   Heinz.setDay(22);

   Peter = &Heinz;
   Peter->setMonth(12);
   Heinz.displayData();
   Peter->displayData();
   system("pause");
}
```

2.9 Example

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

int main()
{
   int x = 3;
   int *pointer = &x;
   cout << "Address which pointer points to: " << pointer << endl;
   cout << "Address of x: " << &x << endl;
   cout << "Address of pointer: " << &pointer << endl << endl;

   cout << "Value of x: " << x << endl;
   cout << "Value of pointer: " << *pointer << endl << endl;
   cout << "Enter value for x: ";

   cin >> x;
   cout << endl;
   cout << "Value of x: " << x << endl;
   cout << "Value of pointer: " << *pointer << endl << endl;

   cout << "Enter value for pointer: ";
   cin >> *pointer;
   cout << endl;
   cout << "Value of x: " << x << endl;
   cout << "Value of pointer: " << *pointer << endl << endl;

   system("pause");

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

   cout << "Address of the first array element: " << array << endl;
   cout << "Address of the first array element: " << &array << endl;
   cout << "Address of the first array element: " << &array[0] << endl;

   cout << endl;
   cout << "Value of the first array element: " << array[0] << endl;
   cout << "Value of the first array element: " << *array << endl << endl;

   cout << endl;
   cout << "Value of the third pointer array: " << array[2] << endl;
   //cout << "Value of the third pointer array: " << *arraypointer[2]; // Does not work!
   cout << "Value of the third pointer array: " << *(array+2) << endl;

   cout << "Display address of the array: "<<endl;
   for(int i = 0; i < 5; ++i)
   {
      cout << "&array[" << i << "] = " << &array[i] << endl;
   }

   cout << endl;
   cout << "Display value of the array: "<<endl;
   for(int i = 0; i < 5; ++i)
   {
      cout << "array[" << i << "] = " << array[i] << endl;
   }
   system("pause");

   // SUMMARY:
   // &array[0] is equal to array
   // *array is equal to array[0]
   // &array[1] is equal to array+1
   // *(arraypointer+1) is equal to array[1]
}
```