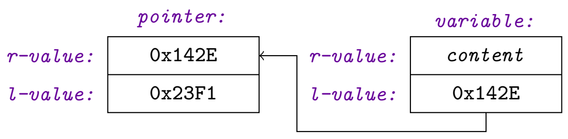
Pointers

Introduction

A *pointer* is a reference variable; its content is the address of a variable of the same type.



It is used as an alias of a variable or an array, or holds *dynamic memory* (explicitly allocated memory), eliminates copying overhead with function invocations, and is ideal for implementing data structures such as linked lists and trees.

Pointer Syntaxes

The pointer syntaxes are

(1) Declaration	`data-type * identifier ;`
(2) Initialization	`data-type * identifier = pointer-argument ;`
(3) Read-Only	`const data-type * identifier ;`
	`data-type const * identifier ;`
(4) Constant	`data-type * const identifier = pointer-argument ;`
(5) Permanent	`const data-type * const identifier = pointer-argument ;`
	`data-type const * const identifier = pointer-argument ;`

where `*` is called the *dereference operator* and the *pointer-argument* formats are

a. `&variable`

b. `array`

c. `pointer`

d. `new data-type[(argument-list)]` (**dynamic variable**)

e. `new data-type[size]` (**dynamic array**)

where & and new are called the *address-of operator* (or *reference operator*) and new operator, respectively, the *argument-list* is used for invoking overloaded constructors, *data-type* matches the data type of *pointer*, and *size* is a positive integer in any format.

Implicit Pointer Assignments

The three initial pointer arguments are of a variable, an array, and a pointer, respectively. For these implicit assignments, the pointer behaves as an alias of either its argument or the content of its argument.

When a variable is assigned to a pointer, the pointer behaves as an alias for the variable. The address-of operator (&) in its syntax is used to access a variable's address.

For an array assignment, the pointer is an alias of the first element of the array since an array without the subscript operator provides the address of its first element; hence, the following array argument syntax is equivalent to the one above:

```
&array[0];
```

Any element of an array can be assigned to a pointer by changing the index in the recent syntax. Regardless of which element is assigned to a pointer, the pointer can access all its elements using pointer arithmetic.

Assigning a pointer to another pointer follows the assignment standard rules, which means the pointer receives a copy of the argument's content; ultimately, making both pointers point to the same address.

Example:

```
int a, b[4], *c, *d, *e, *f, *g, *h;
c = &a; //variable assignment
d = b; //array assignment
e = &b[2]; //array assignment
f = c; //pointer assignment
```

Dynamic Memory

The last pointer arguments use dynamic memory. Dynamic memory does not have an object to access it; therefore, it must be assigned to a pointer; otherwise, it is inaccessible. The new operator is always used to allocate dynamic memory.

Once allocated, the pointer will be used to modify the content of the memory. Consequently, since a pointer can be assigned a new address, it is possible to lose access to dynamically allocated memory, which causes a *memory leak*; thus, it is essential to assign the dynamic memory to another pointer before any new assignment; otherwise, the memory must be deallocated before the new assignment.

Dynamic memory exists indefinitely until it is explicitly deallocated. To deallocate dynamic memory, the *delete* operator (`delete`) is used as follows:

Dynamic Variable: `delete` *`pointer`*`;`

Dynamic Array: `delete[]` *`pointer`*`;`

where *pointer* is assigned the dynamic memory.

After memory is deallocated, it is common to make the pointer a *null pointer*, a pointer assigned a *null value* that is the macro `NULL`, the literal 0, or the keyword `nullptr` (preferred), to indicate that it is freed; otherwise, the pointer will continue to point to the same address. In general, when dealing with dynamic memory, it is good practice to set a pointer to a null pointer before allocation and after deallocation to indicate that it is safe to use.

Example: (continuation)

```
g = nullptr; //null pointer
g = new int; //dynamic variable
h = new int[3]; //dynamic array
delete g; //deallocate dynamic variable
delete[] h; //deallocate dynamic array
```

Constant Pointers

The last three pointer formats are pointer constants. The *read-only pointer* prohibits modifications to its content but allows reassignments, which means it does not have to be initialized when created, and, unlike regular pointers, it can be assigned constant arguments.

The *constant pointer* prohibits reassignment; thus, it must be initialized. However, it permits modifications to its content; hence, it cannot be assigned constant arguments.

Last, the *permanent pointer* prohibits both reassignment and modifications to its content. Thus, it must be initialized and can accept constant arguments.

Example: (continuation)

```
const int i = 5;
const int * j, * const k = &i; //read-only and permanent
int * const l = &a; //constant
j = &a;
j = &i;
```

Pointer Access

Once a pointer is assigned an address, it accesses the content of the address by preceding its name with the dereference operator as follows:

```
*pointer
```

For array arguments, the above syntax only accesses an element of an array argument (the first element if standard format is used). To access any element of an array argument through a pointer, use either of the following syntaxes:

subscript:	`pointer[index]`
arithmetic:	`*(pointer + index)`

where *index* is a standard index of an array for a standard array assignment or an offset of the standard range if an element other than the first element is assigned to the pointer.

Example: (continuation)

```
*c = 23; //a = 23
d[0] = 4; //b[0] = 4
d[3] = 9; //b[3] = 9
e[0] = 5; //b[2] = 5
e[-1] = 10; //b[1] = 10
```

Passing By Address

A function can accept addresses with a *pass by address* parameter. Its syntax is:

```
[const] data-type * [const]identifier
```

where its argument is a pointer argument.

Within the function, the parameter operates as a pointer variable; thus, it can be assigned a new argument, which means it can lose its association with the caller's argument; however, that behavior can be prevented by using a constant pointer parameter.

Example:

```
void print(int * const a) { std::cout << *a << "\n";}

int main()
{
  int a = 8, b[] = {6,8,3};
  print(&a); //prints 8
  print(b); //prints 6
}
```

To ensure that the argument is a pointer, which is essential for manipulation of some data structures, use a *pointer reference parameter* that has the syntax:

```
[const] data-type * [const] & identifier
```

Without the constant quantifiers, only regular pointer arguments are accepted. Including both constant quantifiers allows all types of pointer arguments; meanwhile, a single constant quantifier only allows pointers with the same format as the argument.