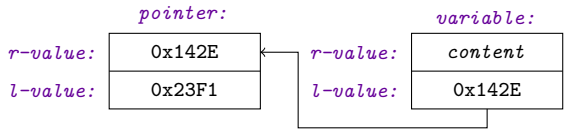
Lecture 5: Pointers

Introduction

A *pointer* is a reference variable; this means its r-value (content) is an address (l-value) of a variable or an array of the same data type as the pointer.



Typically, it acts as an alias for variables and arrays. However, it is also used for *dynamic memory*, explicitly allocated memory.

Pointer Declaration

The declaration of a pointer is very similar to the declaration of a regular variable. Its syntax is

```
data-type *identifier;
```

where * is called the *dereference operator*.

Example:

```
int *alpha; //an int pointer declaration
char *beta; //a char pointer declaration
double *gamma; //a double pointer declaration
long a, b[20], *c, d[100], *f, *g, h; //c, f, and g are pointers
```

For the last statement c, f, and g are pointers, a and h are variables, and b and d are arrays.

Pointer Assignment

An assignment to a pointer depends on its argument's type. The assignment syntaxes are

- **Variable**:

  ```
  pointer = &variable;
  ```

  where & is called the *address-of operator* (or *reference operator*), which is used to access the variable argument's address.

- **Array**:

  ```
  pointer = array;
  ```

  The pointer points to the first element of the array argument.

- **Pointer**:

  ```
  pointer = pointer;
  ```

  A pointer-to-pointer assignment is equivalent to a variable-to-variable argument; the content (r-value) of the argument becomes the r-value of the pointer; hence, another alias of the content of its argument.

Once a pointer is assigned a value, it can modify the content (r-value) of its content by preceding its name with the dereference operator (*) as follows

```
*pointer
```

If the pointer points to an array, the elements of the array can be accessed in one of two ways

version I	version II
pointer[*index*]	*(*pointer* + *index*)

The first element of the array can also be accessed using **pointer*.

Example:

```
int a, b[20], *c, *d;
c = &a; //variable assignment
d = b; //array assignment
*c = 7; //variable a assigned 7
d[4] = 10; //the fifth element of b is assigned 10
*d = 19; //the first element of b is assigned 19
*(d + 19) = 8; //the last element of b is assigned 8
d = c; //d points to a
cout << *d; //a is displayed
```

Pointer Initialization

A pointer can be initialized as well. The initialization syntaxes are

- **Variable**:

data-type ***identifier* = &*variable*;

- **Array**:

data-type ***identifier* = *array*;

- **Pointer**:

data-type ***identifier* = *pointer*;

Constant Pointer

There are two interpretations of a constant pointer. A regular pointer variable **cannot point to a constant variable**; an error will be thrown if attempted. A *read-only pointer* can reference a constant variable (or array). Its syntax is

const *data-type* ***identifier*; or *data-type* const ***identifier*;

A read-only pointer does not have to be initialized, can be assigned a constant or nonconstant variable (or array), and cannot be used to change the content of its content.

The other type of constant pointer, operates as an actual constant variable. Its syntax is

data-type * const *identifier* = *pointer-argument*;

where a *pointer-argument* is a variable reference, array, or pointer. This is the official *constant pointer*. It cannot be assigned a new argument after it is initialized and the arguments must be nonconstant; however, it can modify its argument's content.

Last, a pointer can be both constant and read-only. Its syntax is

const *data-type* * const *identifier*; or *data-type* const * const *identifier*;

It is a constant pointer that can accept both constant and nonconstant arguments. but it can no longer modify its argument's content.

Dynamic Memory

Although a pointer is normally used as an alias of an existing object, it can be used to hold *dynamic memory*, which is explicitly allocated memory. To allocate dynamic memory, the *new* operator (`new`) is used. It can be used to create a dynamic variable or array. Their syntaxes are

Dynamic Variable	Dynamic Array
pointer = new *data-type*;	*pointer* = new *data-type*[*size*];

where *data-type* must match the data type of *pointer* that can be constant or nonconstant and *size* is a positive integer in any format. Afterward, the pointer is used to modify the memory's content.

Since dynamic memory is explicitly allocated, it exists indefinitely until it is deallocated explicitly; a memory leak occurs if it is not deallocated. Thus, to deallocate dynamic memory, the *delete* operator (`delete`) is used. Their syntaxes are

Dynamic Variable	Dynamic Array
delete *pointer*;	delete[] *pointer*;

where *pointer* is assigned the dynamic memory.

The pointer assigned dynamic memory is the only access to the memory; hence, it needs to assign the memory to another pointer before its content can change; otherwise, the memory is no longer accessible. par Additionally, an *null pointer*, a pointer that is assigned an *null value*, which is the macro `NULL`, the literal 0, or the keyword `nullptr`, is used to indicate it points to nothing. Whenever dealing with dynamic memory, null pointers are made before allocation and after deallocation to represent the state of the pointer for safe usage.

Example:

```
double *a[3];
a[0] = new double; //dynamic variable
a[1] = new double[20]; //dynamic array
a[2] = nullptr; //null pointer
*a[0] = 18;

for(int i = 0;i < 20;i += 1) {a[1][i] = 2.0 * i;}
delete a[0]; //deallocate dynamic variable
delete[] a[1]; //deallocate dynamic array
```

Passing By Address

Another format of a function parameter is *pass by address* that accepts addresses. Its syntax is

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

where its argument is a pointer argument.

The parameter operates as a pointer variable in the function; thus, it is similar to a reference parameter except it can point to new content, losing its association with the argument. To prevent argument association loss, make the parameter a constant pointer.

Example:

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

int main()
{
  int a = 8, b[] = {6,8,3}, *c;
  c = &b[2];
  print(&a); //prints 8
  print(b); //prints 6
  print(c); //prints 3
}
```