POINTERS

Bùi Tiến Lên

2023

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Pointers

Objects, Sizes, and Addresses

Concept 1

Each **variable** (object) has an address and a size. The address is where it sits and the size is how many memory locations it takes up.

address	...	100	101	102	103	104	105	...
memory	...							...

Size Operator

- The size can be retrieved using the size operator `sizeof`

```c
#include <stdio.h>
int main(void) {
  char c;
  printf("%zu %zu\n", sizeof(char), sizeof c);
  int i;
  printf("%zu %zu\n", sizeof(int), sizeof i);
  double d;
  printf("%zu %zu\n", sizeof(double), sizeof d);
  return 0;
}
```

Address Operator

- The address can be retrieved using the address operator &

```c
#include <stdio.h>
int main(void) {
  char c = 1;
  printf("%d %p\n", c, &c);
  int i = 2;
  printf("%d %p\n", i, &i);
  double d = 3.0;
  printf("%f %p\n", d, &d);
  return 0;
}
```

What is Pointer

Concept 2

A pointer is a variable that stores addresses of memory locations (addresses of other objects).

- The null pointer does not point to anything

Pointer's Usage

Pointers have several uses, including:

- Creating fast and efficient code
- Providing a convenient means for addressing many types of problems
- Supporting dynamic memory allocation
- Making expressions compact and succinct

Declaring Pointers

- A pointer being a variable needs to be declared like all variables do

```
PointedType * PointerVariableName;
```

- A pointer to void is a general-purpose pointer used to hold references to any data type.

```
void *pv;
```

- Declare pointer type

```
typedef PointedType * PointerTypeName;
```

- Example

```
typedef int * IntPointer;
int *p1;          // pointer to an integer
IntPointer p2;    // pointer to an integer
```

Pointer Operators

Operator	Name	Meaning
`*`	Dereference	Used to dereference a pointer
`->`	Point-to	Used to access fields of a structure referenced by a pointer
`+, +=, ++`	Addition, increment	Used to increment a pointer
`-, -=, --`	Subtraction, decrement	Used to decrement a pointer
`== !=`	Equality, inequality	Compares two pointers
`> >= < <=`	Greater than, greater than or equal, less than, less than or equal	Compares two pointers
(data type)	Cast	To change the type of pointer

Dereferencing a Pointer

```
int num = 5;
int *pi = &num;
*pi = *pi + 2;
printf("%d\n",*pi);
```

Adding an integer to a pointer

```
int vector[] = {28, 41, 7};
int *pi = vector;       // pi: 100

printf("%d\n",*pi);     // Displays 28
pi += 1;                // pi: 104
printf("%d\n",*pi);     // Displays 41
pi += 1;                // pi: 108
printf("%d\n",*pi);     // Displays 7
```

Subtracting an integer from a pointer

```
int vector[] = {28, 41, 7};
int *pi = vector + 2;  // pi: 108

printf("%d\n",*pi);    // Displays 7
pi--;                  // pi: 104
printf("%d\n",*pi);    // Displays 41
pi--;                  // pi: 100
printf("%d\n",*pi);    // Displays 28
```

Subtracting two pointers

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2-p0:  %d\n",p2-p0);     // p2-p0:  2
printf("p2-p1:  %d\n",p2-p1);     // p2-p1:  1
printf("p0-p1:  %d\n",p0-p1);     // p0-p1:  -1
```

Comparing Pointers

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2>p0:  %d\n",p2>p0);     // p2>p0:  1
printf("p2<p0:  %d\n",p2<p0);     // p2<p0:  0
printf("p0>p1:  %d\n",p0>p1);     // p0>p1:  0
```

Multilevel Pointer

Concept 3

A pointer can store the address of another pointer variable.

```
int num = 100;
int *p1;
int **p2;
int ***p3;
p1 = &num;
p2 = &p1;
p3 = &p2;
```

Constants and Pointers

Const

Concept 4

Using the `const` keyword to protect variables from changing

Pointer Type	Pointer Modifiable	Data Pointed to Modifiable
Pointer to a nonconstant	✓	✓
Pointer to a constant	✓	X
Constant pointer to a nonconstant	X	✓
Constant pointer to a constant	X	X

Pointer to a constant

```
int num = 5;
const int limit = 500;
int *pi;                    // Pointer to an integer
const int *pci;             // Pointer to a constant integer

pi = &num;
pci = &limit;
```

Constant pointer to a nonconstant

```
int num = 5;
int *const cpi = &num;
```

Constant pointer to a constant

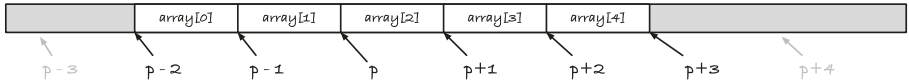
```
const int limit = 500;
const int * const cpci = &limit;
```

Pointers and Arrays

One-Dimensional Arrays

- Example

```
int array[5];
int *p = array + 2;
```



- Technique

```
T * range_begin = array;
T * range_end = array + n; // n is the length of the array
for (T *p = range_begin; p < range_end; p++) {
  // do something
}
```

Pointer to Arrays

- Syntax

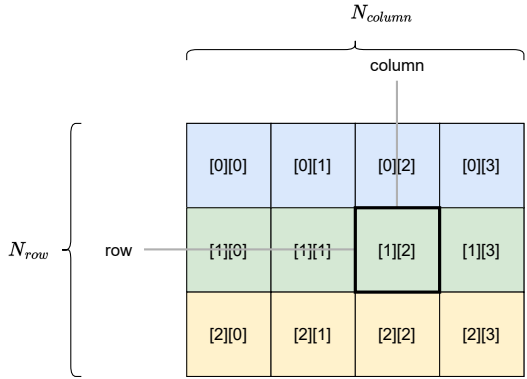
```
DataType (*PointerVariableName)[SizeOfArray];
```

- Example

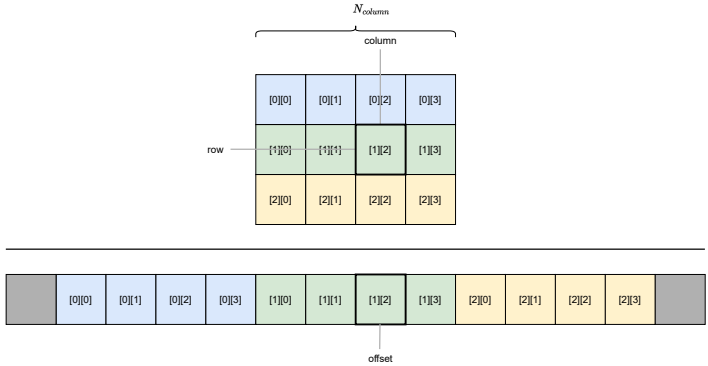
```
int (*p1)[10];
double (*p2)[50];
```

Two-Dimensional Arrays

- Row major
- Column major



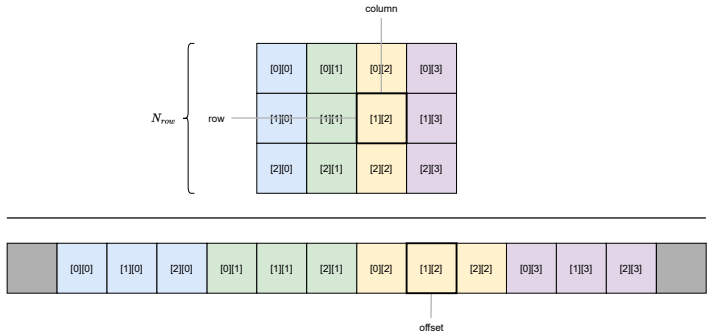
2D row major



$$offset \longleftrightarrow (row, column)$$

$$offset = row \times N_{column} + column$$

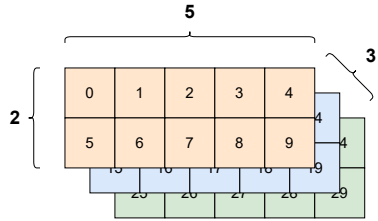
2D column major



$$offset \longleftrightarrow (row, column)$$

$$offset = column \times N_{row} + row$$

Multidimensional Arrays



0	1	2	3	4
5	6	7	8	9

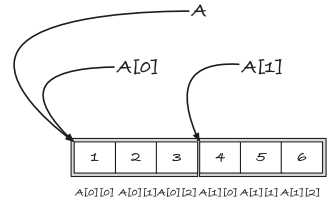
10	11	12	13	14
15	16	17	18	19

20	21	22	23	24
25	26	27	28	29

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	

Multidimensional Arrays (cont.)

```
int A[2][3] = {
  { 1, 2, 3 },
  { 4, 5, 6 }
};
```



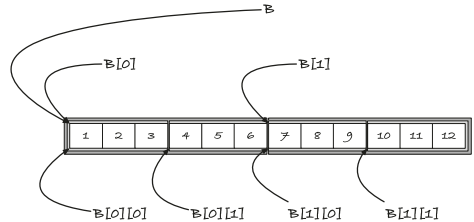
Zero dimensions, A[i][j] int int int int int int

One dimension, A[i] int[3] int[3]

Two dimensions, A int[2][3]

Multidimensional Arrays (cont.)


```
int B[2][2][3] = {
  { { 1, 2, 3 }, {  4,  5,  6 } },
  { { 7, 8, 9 }, { 10, 11, 12 } }
};
```



	int	int	int	int	int	int	int	int	int	int	int	int
One dimension, B[i][j]	int[3]			int[3]			int[3]			int[3]		
Two dimensions, B[i]	int[2][3]						int[2][3]					
Three dimensions, B	int[2][2][3]											

Pointers and Functions

Passing Data by Value

```
void swap(int num1, int num2) {
    int tmp;
    tmp = num1;
    num1 = num2;
    num2 = tmp;
}
int main() {
    int n1 = 5;
    int n2 = 10;
    swap(n1, n2);
    return 0;
}
```

Passing Data Using a Pointer

```
void swapWithPointers(int* pnum1, int* pnum2) {
    int tmp;
    tmp = *pnum1;
    *pnum1 = *pnum2;
    *pnum2 = tmp;
}
int main() {
    int n1 = 5;
    int n2 = 10;
    swapWithPointers(&n1, &n2);
    return 0;
}
```

Passing a Pointer to a Constant

```
void passingAddressOfConstants(const int* num1, int* num2) {
    *num2 = *num1;
}
int main() {
    const int limit = 100;
    int result = 5;
    passingAddressOfConstants(&limit, &result);
    return 0;
}
```

Returning a Pointer

```
int* allocateArray(int size, int value) {
    int* arr = new int[size];
    for(int i=0; i<size; i++) {
        arr[i] = value;
    }
    return arr;
}
```

Returning a Pointer (cont.)

several potential problems can occur when returning a pointer from a function, including:

- Returning an uninitialized pointer
- Returning a pointer to an invalid address
- Returning a pointer to a local variable
- Returning a pointer but failing to free it

Function Pointers

Concept 5

A function pointer is a pointer that holds the address of a function.

```
ReturnType (*PointerVariableName)(...);
```

```
int (*f1)(double);          // Passed a double and
                            //    returns an int
void (*f2)(char*);          // Passed a pointer to char and
                            //    returns void
double* (*f3)(int, int); // Passed two integers and
                            //    returns a pointer to a double
```

- Declare function pointer type

```
typedef ReturnType (*PointerTypeName)(...);
```

Using a Function Pointer

```
int (*fptr1)(int);
int square(int num) {
    return num*num;
}
int main() {
    int n = 5;
    fptr1 = square;
    printf("%d squared is %d\n",n, fptr1(n));
}
```

Pointers and Structures

Pointers and Structures

- Declare a structure

```
struct Person {
    char firstName[100];
    char lastName[100];
    char title[10];
    unsigned int age;
};
```

- Declare a pointer

```
Person *p;
```

Point-to Operator

- Access the fields of a structure variable

```
cout << p->title;
cout << (*p).age;
```

- The awkwardness of this expression `(*p).age` shows the necessity of the `->` operator.
- Remember that the operators `->` and `.` for selecting members of structures have higher precedence than the dereferencing operator `*`.

Expression	Meaning
`s->m`	
`*a.p`	
`(*s).m`	
`*s->p`	
`*(*s).p`	

Workshop

Quiz

1. What is a pointer?

Exercises

- Write a program

References

Deitel, P. (2016).
C++: How to program.
Pearson.

Gaddis, T. (2014).
Starting Out with C++ from Control Structures to Objects.
Addison-Wesley Professional, 8th edition.

Jones, B. (2014).
Sams teach yourself C++ in one hour a day.
Sams.