Programming Design

Pointers

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Basics of pointers**
- Using pointers in functions
- Dynamic memory allocation (DMA)
- Arrays and pointer arithmetic

Pointers

- A **pointer** is a variable which stores a **memory address**.
  - An **array** variable also stores a memory address.
- To declare a pointer, use `*`.

```
type pointed* pointer name;
```

```
type pointed *pointer name;
```

- Examples:

```
int *ptrInt;
```

```
double* ptrDou;
```

  - These pointers will store addresses.
  - These pointers will store addresses of **int**/**double** variables.
- We may point to **any** type.
- To point to different types, use different types of pointers.

Sizes of pointers

- All pointers have the same size.
  - In a 32-bit computer, a pointer is allocated 4 bytes.
  - In a 64-bit computer, a pointer is allocated 8 bytes.

```
int* p1 = 0;
cout << sizeof(p1) << "\n"; // 8
double* p2 = 0;
cout << sizeof(p2) << "\n"; // 8
```

- The length of pointers decides the maximum size of the memory space.
  - 32 bits: 2^{32} bytes = 4GB.
  - 64 bits: 2^{64} bytes = ?

Pointer assignment

- We use the **address-of operator `&`** to obtain a variable's address:

```
pointer name = &variable name
```

- The address-of operator `&` returns the (beginning) **address** of a variable.
- Example:
  - `ptr` points to `a`, i.e., `ptr` stores **the address of `a`**.

```
int a = 5;
int* ptr = &a;
```

- When assigning an address, the two types must **match**.

```
int a = 5;
double* ptr = &a; // error!
```

Variables in memory

- `int a = 5;`
- `double b = 10.5;`
- `int* aPtr = &a;`
- `double* bPtr = &b;`
- `cout << &a; // 0x20c644`
- `cout << &b; // 0x20c660`
- `cout << &aPtr; // 0x20c658`
- `cout << &bPtr; // 0x20c64c`

Address	Identifier	Value
0x20c644	a	5
0x20c64c	bPtr	0x20c660
0x20c650		
0x20c658	aPtr	0x20c644
0x20c65c		
0x20c660	b	10.5
0x20c664		

Memory

Address operators

- There are two address operators.
  - `&`: The **address-of operator**. It returns a variable's address.
  - `*`: The **dereference operator**. It returns the pointed variable.
- For `int a = 5`:
  - `a` equals 5.
  - `&a` returns an address (e.g., `0x22ff78`).
- For `int* ptrA = &a`:
  - `ptrA` stores an address (e.g., `0x22ff78`).
  - `&ptrA` returns the pointer's address (e.g., `0x21aa74`). This has nothing to do with the pointed variable `a`.
  - `*ptrA` returns `a`, **the variable** pointed by the pointer.

Address operators

- Example:

```
int a = 10;
int* p1 = &a;
cout << "value of a = " << a << "\n";
cout << "value of p1 = " << p1 << "\n";
cout << "address of a = " << &a << "\n";
cout << "address of p1 = " << &p1 << "\n";
cout << "value of the variable pointed by p1 = " << *p1 << "\n";
```

Address operators

- `&` returns **a variable's address**.
  - We cannot use `&100`, `&(a++)` (because `a++` returns the value of `a`).
  - We can only perform `&` on a **variable**.
  - We cannot assign a value to `&x` (`&x` is a value!).
  - We can get a usual variable's or a pointer variable's address.
- `*` returns **the pointed variable**.
  - We can perform `*` on a pointer variable.
  - We cannot perform `*` on a usual variable.
  - We cannot change a variable's address. No operation can do this.

Address operators

- What is `*&x` if `x` is a variable?
  - `&x` is the address of `x`.
  - `*(&x)` is the variable stored in that address.
  - So `*(&x)` is `x`.
- What is `&*x` if `x` is a pointer?
  - If `x` is a pointer, `*x` is the variable stored at `x` (`x` stores an address!).
  - `&*x` is the address of `*x`, which is exactly `x`.
- `&` and `*` **cancel each other**.
- What is `&*x` if `x` is not a pointer?

Address operators: examples

```
int a = 10;
int* ptr = &a;
cout << *ptr; // ?
*ptr = 5;
cout << a;    // ?
a = 18;
cout << *ptr; // ?
```

```
int a = 10;
int* ptr1;
int* ptr2;
ptr1 = ptr2 = &a;
cout << *ptr1; // ?
*ptr2 = 5;
cout << *ptr1; // ?
(*ptr1)++;
cout << a;     // ?
```

Null pointers

- What will happen?

```
int* ptr;
cout << *ptr; // ?
```

- If we dereference a pointers of unknown value, the outcome is unpredictable.
  - The pointers points to **somewhere**... And we do not know where it is!
- A pointer **pointing to nothing** should be assigned **nullptr, NULL, or 0**.
  - Dereferencing a null pointer shutdowns the program (a run-time error).

```
int* p2 = nullptr;
cout << "value of p2 = " << p2 << "\n";
cout << "address of p2 = " << &p2 << "\n";
cout << "the variable pointed by p2 = " << *p2 << "\n";
```

Null pointers

- As a bad example:

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

int main()
{
  int* ptrArray[10000];
  for(int i = 0; i < 10000; i++)
    cout << i << " " << *ptrArray[i] << "\n";
  return 0;
}
```

Good programming style

- Initialize a pointer as **`nullptr`**, `0`, or **`NULL`** if no initial value is available.
  - **`nullptr`** is the current standard in C++, but they are all the same for representing a "null pointer".
  - By using **`nullptr`** (instead of `0`), everyone knows the variable must be a pointer, and you are not talking about a number or character.
- In general, when you get **a run time error** or different outcomes for multiple executions, check your arrays and pointers.
- When we use `*` in **declaring** a pointer, that `*` is not a dereference operator.
  - It is just a special syntax for declaring a pointer variable.
- When we use `&` in **declaring** a reference, that `&` is not an address-of operator.
  - It is just a special syntax for declaring a reference variable.

Good programming style

- I prefer to view **`int*`** as a type, which represents an "integer pointer".
  - I prefer "**`int* p`**" to "**`int *p`**".
- The other way is also common. It views **`*p`** as an integer.
  - They prefer "**`int *p`**" to "**`int* p`**".
- Be consistent throughout your program.
- Be careful:

```
int b = 5;
int *ptr1 = &b; // int, int, addr
*ptrB = 12;
cout << b << "\n";
int* ptr2 = &b; // addr, addr, addr
```

```
int* p, q;    // p is int*, q is int
int *p, *q;   // two pointers
int* p, *q;   // two pointers
int* p, * q;  // two pointers
```

Outline

- The basics of pointers
- **Using pointers in functions**
  - Call by reference
  - Call by pointer
  - Returning a pointer
- Dynamic memory allocation (DMA)
- Arrays and pointer arithmetic

References and pointers

- Recall this example:
- When invoking a function and passing parameters, the default scheme is to "**call by value**" (or "pass by value").
  - The function declares its own local variables, using a copy of the arguments' values as initial values.
  - Thus we swapped the two local variables declared in the callee, not the two in the caller that we want to swap.
- To solve this, we can use "**call by reference**" or "call by pointer."

```
void swap(int x, int y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << "\n";
  swap(a, b);
  cout << a << " " << b << "\n";
}
void swap(int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

References

- A **reference** is a variable's **alias**.
- The reference is another variable that refers to the variable.
- Thus, using the reference is the same as using the variable.

```
int c = 10;
int& d = c; // declare d as c's reference
d = 20;
cout << c << "\n"; // 20
```

- `int& d = c` is to declare `d` as `c`'s reference.
  - This `&` is different from the `&` operator which returns a variable's address.
- `int& d = 10` is an error.
  - A literal cannot have an alias!

Call by reference

- Now we know how to change a parameter's value:
  - Instead of declaring a usual local variable as a parameter, declare a **reference** variable.
- This is to "call by reference".

```
void swap(int& x, int& y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << "\n";
  cout << &a << "\n";
  swap(a, b);
  cout << a << " " << b << "\n";
}
void swap(int& x, int& y)
{
  cout << &x << "\n";
  int temp = x;
  x = y;
  y = temp;
}
```

Call by reference

- Thus we can call by reference and modify our arguments' values.
- When calling by reference, the only thing you have to do is to add an **&** in the parameter declaration in the function header.
- Mostly people use references only to call by reference.
- View the **&** in declaration as a part of type.
  - I use **int& a = b** instead of **int &a = b**.
  - Be consistent of your choice about **int& a = b** and **int &a = b**.

```
void swap(int& x, int& y);
int main()
{
  int a = 10, b = 20;
  swap(a, b);
}
```

Call by pointers

- To call by pointers:
  - Declare a **pointer** variable as a parameter.
  - Pass a pointer variable or an address (e.g., returned by `&`) at invocation.
- For the `swap()` example:

```
void swap(int* ptrA, int* ptrB)
{
  int temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
}
```

- Invocation becomes `swap(&a, &b);`

Address	Identifier	Value
0x20c644		
0x20c648		
0x20c64c		
0x20c650		
0x20c654		
0x20c658		
0x20c65c		
0x20c660	a	20
0x20c664	b	10

Memory

Call by pointers

- How about the following implementation?

```
void swap(int* ptrA, int* ptrB)
{
  int* temp = ptrA;
  ptrA = ptrB;
  ptrB = temp;
}
```

  – Invocation: `swap(&a, &b);`

Address	Identifier	Value
0x20c644		
0x20c648		
0x20c64c		
0x20c650		
0x20c654		
0x20c658		
0x20c65c		
0x20c660	a	10
0x20c664	b	20

Memory

Call by pointers

- The principle behind calling by reference and calling by pointer is the same.
- You can view calling by reference as a special tool made by using pointers.
- Do not mix references and pointers!
  - E.g., we cannot pass a pointer variable or an address to a reference!
- You can call by reference in most situations, and it is clearer and more convenient than to call by pointer.
  - When you just want to modify arguments or return several values, call by reference.
  - When you really have to do something by pointers, call by pointer.

Returning a pointer

- May a function **return a pointer**? Yes!
  - We simply **returns an address**.
- Why returning an address?
  - `p` records the address of the first negative number in the array `a`.
  - With the address, we **also know** the value of that negative number.
  - If we only have the value, we **do not know** its address (and index).
- To obtain the index, we need **pointer arithmetic**.

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  } // what if a[i] >= 0 for all i?
}
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = firstNeg(a, 5);
  cout << *p << " " << p << "\n";
  return 0;
}
```

Outline

- The basics of pointers
- Using pointers in functions
- **Dynamic memory allocation (DMA)**
- Arrays and pointer arithmetic

Static memory allocation

- We declare an array by specifying it's length as a constant variable or a literal.

```
const int ARRAY_LEN = 100;
int a[ARRAY_LEN];
```

- Memory allocation to an array can be determined during the compilation time.
  - 400 bytes will be allocated for the above statements.
- This is called "**static memory allocation**".
- We may decide the length of an array "**dynamically**".
  - That is, during the **run** time.
- To do so, we must use a different syntax.
  - All types of variables may also be declared in this way.

Dynamic memory allocation

- The operator **`new`** allocates a memory space **and** returns the address.
  - In C, we use a different keyword **`melloc`**.
- **`new int`** allocates 4 bytes, and the returned address is not recorded.
- **`int* a = new int`** makes **`a`** store the address of the 4-byte space.
- **`int* a = new int(5)`** makes the space contain 5 as the value.
- **`int* a = new int[5]`** allocates 20 bytes (for 5 integers).
  - **`a`** points to the first integer.
  - **`a`** can be viewed as an array. It is a **dynamic array**.
- Dynamically allocated arrays **cannot be initialized** with a single statement.
  - A loop, for example, is needed.

Dynamic memory allocation

- Memory allocation (the size and location of the space) is determined during the **run time**.
- So we may write

```
int len = 0;
cin >> len;
int* a = new int[len];
```

- This allocates space according to the input from users.

Dynamic memory allocation

- Space allocated during the run time has **no name**!
  - On the other hand, every space allocated during the compilation time has a name.
- To access a dynamically-allocated space, we use a **pointer** to store its address.

```
int len = 0;
cin >> len; // 3
int* a = new int[len];
for(int i = 0; i < len; i++)
  a[i] = i + 1;
```

Address	Identifier	Value
0x20c644	N/A	1
0x20c648		2
0x20c64c		3
0x20c650		
0x20c654		
0x20c658	len	3
0x20c65c		
0x20c660	a	0x20c644
0x20c664		

Memory

Example: Fibonacci sequence

- Recall the repetitive implementation of generating the Fibonacci sequence.
- After we get the value of sequence length *n*, we dynamically declare an array of length *n*.
- Then just use that array!

```
double fibRepetitive(int n)
{
  if(n == 1)
    return 1;
  else if(n == 2)
    return 1;
  double* fib = new double[n];
  fib[0] = 1;
  fib[1] = 1;
  for(int i = 2; i < n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
  double result = fib[n - 1];
  delete[] fib; // to be explained
  return result;
}
```

Memory leak

- For space allocated during the **compilation** time, the system will **release this space** automatically when the corresponding variables no longer exist.

```
void func(int a)
{
  double b;
} // 4 + 8 bytes are released
int main()
{
  func(10);
  return 0;
}
```

Address	Identifier	Value
0x20c644		
0x20c648		
0x20c64c	a	10
0x20c650		
0x20c654		
0x20c658	b	?
0x20c660		
0x20c664		

Memory

Memory leak

- For space allocated during the **run** time, the system will **not** release this space unless it is asked to do so.
  - Because the space has no name!

```
void func()
{
  int* bPtr = new int[3];
}
// 8 bytes for bPtr are released
// 12 bytes for integers are not
int main()
{
  func( );
  return 0;
}
```

Address	Identifier	Value
0x20c644		
0x20c648	N/A	?
0x20c64c	N/A	?
0x20c650	N/A	?
0x20c654		
0x20c658		
0x20c65c		
0x20c660		
0x20c664		

Memory

Memory leak

- Programmers must keep a record for all space allocated dynamically.

```
double* b = new double;
*b = 5.2;
double c = 10.6;
b = &c; // now no one can access
        // the space containing 5.2
```

- This problem is called **memory leak**.
  - We lose the control of allocated space.
  - These space are **wasted**.
  - They will not be released unit the program ends.

Address	Identifier	Value
0x20c644		
0x20c648	b	0x20c660
0x20c650		
0x20c654	N/A	5.2
0x20c65c		
0x20c660	c	10.6

Memory

Memory leak

- Try this carefully!
  - The outcome may be different on your computer.

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

int main()
{
  for(int i = 0; ; i++)
  {
    int* ptr = new int[100000];
    cout << i << "\n";
    // delete [] ptr;
  }
  return 0;
}
```

Releasing space manually

- The **delete** operator will release a dynamically-allocated space.

```
int* a = new int;
delete a; // release 4 bytes
int* b = new int[5];
delete b; // release only 4 bytes!
          // Unpredictable results may happen
delete [] b; // release all 20 bytes
```

- The **delete** operator will do nothing to the pointer. To avoid reusing the released space, set the pointer to **nullptr**.

```
int* a = new int;
delete a;    // a is still pointing to the address
a = nullptr; // now a points to nothing
int* b = new int[5];
delete [] b; // b is still pointing to the address
b = nullptr; // now b points to nothing
```

Good programming style

- Use DMA for arrays with **no predetermined** length.
  - Even though Dev-C++ (and some other compilers) converts

```
int a = 10;
int b[a];
```

to

```
int a = 10;
int* b = new int[a];
// ...
delete [] b;
```

- To avoid memory leak:
  - Whenever you write a **`new`** statement, add a **`delete`** statement below immediately (unless you know you really do not need it).
  - Whenever you want to change the value of a pointer, check whether memory leak occurs.
  - Whenever you write a **`delete`** statement, set the pointer to **`nullptr`**.

Two-dimensional dynamic arrays

- With static arrays, we may create matrices as two-dimensional arrays.
- An *m* by *n* two-dimensional array has:
  - *m* rows (single-dimensional arrays).
  - Each row has *n* elements.
- With dynamic arrays, we now may create matrices **with different row lengths**.
  - We may still have *m* rows.
  - Now each row may have different number of elements.
  - E.g., a **lower triangular matrix**.

Example: lower triangular arrays

- `int* array = new int[10]` declares an array of integers.
- `int** array = new int*[10]` declares **an array of integer pointers**!
  - The type of `array[0]` is `int*`.
  - The type of `array[1]` is `int*`.
- Then each of these integer pointers may store the address of a dynamic integer array.
  - And their lengths can be different.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r); // later
  // some delete statements
  return 0;
}
```

Example: lower triangular arrays

- Let's visualize the memory events.
- In general, the space of the three 1-dim dynamic arrays may be **separated**.
- However, the space of the array elements in each array are **contiguous**.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r); // later
  // some delete statements
  return 0;
}
```

Address	Identifier	Value
0x20c644	r	3
0x20c648	Array	0x20c654
0x20c650		
0x20c654	N/A	0x20c66c
0x20c65c	N/A	0x20c670
0x20c664	N/A	0x20c678
0x20c66c	N/A	1
0x20c670	N/A	1
0x20c674	N/A	2
0x20c678	N/A	1
0x20c67c	N/A	2
0x20c680	N/A	3

Memory

Example: lower triangular arrays

- To pass a two-dimensional dynamic array, just pass that pointer.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r);
  // some delete statements
  return 0;
}
```

```
int print(int** arr, int r)
{
  for(int i = 0; i < r; i++)
  {
    for(int j = 0; j <= i; j++)
      cout << arr[i][j] << " ";
    cout << "\n";
  }
}
```

Example: lower triangular arrays

- An alternative:

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r);
  // some delete statements
  return 0;
}
```

```
int print1D(int* arr, int n)
{
  for(int i = 0; i < n; i++)
    cout << arr[j] << " ";
  cout << "\n";
}
int print(int** arr, int r)
{
  for(int i = 0; i < r; i++)
  {
    print1D(arr[i], i + 1);
  }
}
```

Outline

- The basics of pointers
- Using pointers in functions
- Dynamic memory allocation (DMA)
- **Arrays and pointer arithmetic**

Pointers and arrays

- An array variable stores an address, **just like** a pointer!
  - It records the address of the **first** element of the array.
  - When passing an array, we pass an address.
  - The array indexing operator **[]** indicates **offsetting**.
- To further understand this issue, let's study **pointer arithmetic**.
  - Using **+**, **−**, **++**, and **−−** on pointers.

Pointer arithmetic: ++ and --

- **++**: Increment the pointer variable's value by the number of bytes occupied by a variable in this type (i.e., point to the **next** variable).
  - E.g., for integer pointers, the value (an address) increases by 4 (bytes).
- **--**: Decrement the pointer variable's value by the number of bytes a variable in this type occupies (i.e., point to the **previous** variable).

```
int a = 10;
int* ptr = &a;
cout << ptr++;
  // just an address
  // we don't know what's here
cout << *ptr;
  // dangerous!
```

Pointer arithmetic

- Usually, one arbitrary address returned by performing arithmetic on a pointer variable is useless.
- The arithmetic is useful (and should be used) only when you can predict a variable's address.
  - In particular, when variables are stored **consecutively**.

```
double a[3] = {10.5, 11.5, 12.5};
double* b = &a[0];
cout << *b << " " << b << "\n";  // 10.5
b = b + 2; // b++ and then b++
cout << *b << " " << b << "\n";  // 12.5
b--;
cout << *b << " " << b << "\n";  // 11.5
```

Pointer arithmetic: –

- We cannot add two address.
- However, we can find the difference of two addresses.

```
double a[3] = {10.5, 11.5, 12.5};
double* b = &a[0];
double* c = &a[2];
cout << c - b << "\n"; // 2, not 16!
```

Pointers and arrays

- Changing the value stored in a pointer is dangerous:

```
int y[3] = {1, 2, 3};
int* x = y;
for(int i = 0; i < 3; i++)
  cout << *(x + i) << " "; // 1 2 3
for(int i = 0; i < 3; i++)
  cout << *(x++) << " "; // 1 2 3
for(int i = 0; i < 3; i++)
  cout << *(x + i) << " "; // unpredictable
```

Indexing and pointer arithmetic

- The array indexing operator `[]` is just an **interface** for doing pointer arithmetic.
  - Interface: a (typically safer and easier) way of completing a task.

```
int x[3] = {1, 2, 3};
for(int i = 0; i < 3; i++)
  cout << x[i] << " "; // x[i] == *(x + i)
for(int i = 0; i < 3; i++)
  cout << *(x + i) << " "; // 1 2 3
```

  - `x[i]` and `*(x + i)` are identical, but using the former is safer and easier.
- The address stored in an array variable (e.g., `x`) cannot be modified.

```
int x[3] = {1, 2, 3};
for(int i = 0; i < 3; i++)
  cout << *(x++) << " "; // error!
```

Example 1: incrementing array elements

- What does the following program do?

```
#include <iostream>
using namespace std;
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = a;
  for(int i = 0; i < 5; i++) {
    *p += 3;
    p++;
  }
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

Example 2: insertion sort

- Consider the **insertion sort** taught last time.
  - Given a unsorted array A of length n, we first sort $A[0:(n-2)]$, and then insert $A[n-1]$ to the sorted part.
  - To complete this task, we do this **recursively**.
- What if we want to **first sort $A[1:(n-1)]$**, and then insert $A[0]$?
- We will need to implement a function:

```
void insertionSort(int array[], const int n);
```

  - Given **`array`**, each time when we (recursively) invoke it, we pass a **shorter** array formed by elements from **`array[1]`** to **`array[n - 1]`**, the **second** element to the last element.

Example 2: insertion sort

```
void insertionSort(int array[], const int n) {
  if(n > 1) {
    insertionSort(array + 1, n - 1);
    int num1 = array[0];
    int i = 1;
    for(; i < n; i++) {
      if(array[i] < num1)
        array[i - 1] = array[i];
      else
        break;
    }
    array[i - 1] = num1;
  }
}
```

Example 3: returning a pointer

- Recall that we want to find the first negative number in an array.
  - We want its **value** and **index**.
  - We return its address.
- Three issues remain.
  - Why not return its index?
  - What if all elements in `a` are nonnegative?
  - Why not `const int a[]`?

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  } // what if a[i] >= 0 for all i?
}
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = firstNeg(a, 5);
  cout << *p << " " << p - a << "\n";
  return 0; // what is p - a?
}
```

Example 3: returning a pointer

- To take the possibility of having no negative number into consideration:

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  }
  return nullptr;
}
```

```
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = firstNeg(a, 5);
  if(p != nullptr)
    cout << *p << " " << p - a << "\n";
  return 0;
}
```

Example 3: returning a pointer

- Why not **`const int a[]`**?
  - We return the address of **`a[i]`**, which allows the caller to alter the element.
  - **`const int*`** and **`int*`** are different!

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  }
  return nullptr;
}
```

```
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = firstNeg(a, 5);
  if(p != nullptr)
    *p = -1 * *p; // *p at the LHS of =
  return 0;
}
```

Example 3: returning a pointer

- To use **`const int a[]`**, we need to change the return type.
  - We should also **`return const int*`**.
  - This is an **`int*`** that cannot be put at the LHS of an assignment operator.

```
#include <iostream>
using namespace std;
const int* firstNeg
           (const int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  }
  return nullptr;
}
```

```
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  const int* p = firstNeg(a, 5);
  if(p != nullptr)
    cout << *p << "\n"; // OK
  return 0;
}
```

Remarks

- When should we use pointers?
  - Call by reference/pointer.
  - Dynamic memory allocation and dynamic arrays.
  - Dynamic data structures (to be introduced later in this semester).
  - C strings (to be introduced later in this semester).
- If not needed, avoid using pointers.
  - In the past, using pointers may enhance the run-time efficiency (at the implementation level).
  - Modern compilers are good at implementation-level efficiency optimization.
  - Readability is more important.