CS2311 Computer Programming

LT10: Pointer II

Computer Science, City University of Hong Kong

Semester B 2022-23

Outline Today

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer Arithmetic

- You can perform arithmetic operations on a pointer with four operators
  - ++, --, +, and –
- When you do arithmetic with a pointer *p*, you consider *p* points to an array, and you perform arithmetic as it's an array index
- e.g.

```
int a[4] = {0, 1, 2, 3};
int *p = &a[3];
p -= 2; // now p points to a[1]
cout << *p << endl;
p++;    // now p points to a[2]
cout << *p << endl;
```

Pointer Arithmetic

```
int a[6] = {0, 1, 2, 3, 4, 5};
int *pa = &a[1];
cout << hex << pa    << endl;
cout << hex << ++pa << endl;
cout << &a[5]-pa     << endl;

long b[4] = {5, 4, 3, 2, 1, 0};
long *pb = &b[1];
cout << hex << pb    << endl;
cout << hex << ++pb << endl;
cout << &b[5]-pb     << endl;
```

Pointer Arithmetic

- Pointer arithmetic is equivalent to array index arithmetic

```
char str[] = "Hello World";
char *p = str;
cout << p+5      << endl;
cout << &str[5] << endl;
cout << str[2]  << endl;
cout << *(p+2)  << endl;
```

Pointer Arithmetic Summary

Equivalent representation		Remark
`num`	`&num[0]`	num is the address of the 0th element of the array
`num+i`	`&(num[i])`	Address of the i-th element of the array
`*num`	`num[0]`	The value of the 0-th element of the array
`*(num+i)`	`num[i]`	The value of the i-th element of the array
`(*num)+i`	`num[0]+i`	The value of the 0-th element of the array plus I

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer Array

- A pointer array's elements are all pointers.
- For example,

```
int a[6] = {0,1,2,3,4,5};
int *m[2] = {&a[0], &a[3]};
for (int row=0; row<2; row++) {
      for (int col=0; col<3; col++)
            cout << m[row][col] << " ";
      cout << "\n";
}
```

Pointer Array

- int main(int argc, char *argv[])
- Allows main to take parameter from user input
- int argc: number of arguments to take
- char *argv[]: array of arguments, each is a string

Pointer Array

```
// ./main apple banana orange peach pear

#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
	cout << "Have " << argc << " arguments: " << endl;
	for (int i = 0; i < argc; i++)
		cout << argv[i] << endl;
	return 0;
}
```

Array Pointer

- Pointer to a one-dimensional array can be declared as:

```
int arr[] = {1,2,3,4,5};
int *p; p = arr;
```

- Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
int *p[3] = arr;    // cannot declare as an array of two pointers
```

Array Pointer

- Pointer to a one-dimensional array can be declared as:

```
int arr[] = {1,2,3,4,5};
int *p; p = arr;
```

- Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
int *p[3] = arr;    // cannot declare as an array of two pointers
int (*p)[3] = arr; // a pointer to an array of three integers
cout << *(*(p+1)+2) << endl; // p[1][2]
cout << *(p[2]+1)   << endl; // p[2][1]
```

Pass 2D Array to Function

```
void foo(int x[][10]) { // the size of the second dimension MUST be given
    …                   // the size of the first dimension is optional
}
void main() {
    int y[20][10];
    foo(y);
}
```

Pass Array Pointer to Function

```
void foo(int (*x)[10]) {  //  pointer to an array of 10 integers
    …
}
void main() {
    int y[20][10];
    foo(y);
}
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer of Pointer

- Example:

```
int a = 4;
int *p = &a;
int **pp = &p; //  pp is a pointer to an int pointer
cout << *p << endl;
cout << **pp << endl;
```

Pointer of Pointer

- Example:

```
int a = 4;
int *p = &a;
int **pp = &p; //  pp is a pointer to an int pointer
cout << *p << endl;
cout << **pp << endl;
cout << hex << p << endl;
cout << hex << pp << endl;
cout << hex << *pp << endl;
```

Why Need Pointer of Pointer?

- Example: write a program to skip leading spaces in a string

- Does the right-side program work? Why?

```
void skipSpaces(char *p) {
        while (*p == ' ')
                p++;
        cout << p << endl;
}

int main() {
        char str[] = "    hello";
        char *p = str;
        skipSpaces(p);
        cout << p;
        return 0;
}
```

Why Need Pointer of Pointer?

- Example: write a program to skip leading spaces in a string

- We want the called function to modify the pointer, so …

```
void skipSpaces(char **p) {
        while (**p == ' ')
                (*p)++;
        cout << *p << endl;
}

int main() {
        char str[] = "    hello";
        char *p = str;
        skipSpaces(&p);
        cout << p;
        return 0;
}
```

Pointer's Pointer vs Pointer Reference

```
void skipSpaces(char **p) {
	while (**p == ' ')
		(*p)++;
	cout << *p << endl;
}
int main() {
	char str[] = "    hello";
	char *p = str;
	skipSpaces(&p);
	cout << p;
	return 0;
}
```

```
void skipSpaces(char* &p) {
	while (*p == ' ')
		p++;
	cout << p << endl;
}
int main() {
	char str[] = "    hello";
	char *p = str;
	skipSpaces(p);
	cout << p;
	return 0;
}
```

Quick Summary

- Array of pointer

```
int *a[2];
```

- Pointer of array

```
int a[4][2] = {{0,1}, {2,3}, {4,5}, {6,7}}; int (*p)[2] = a;
cout << p[2][1] << " " << *(*(p+2)+1) << " " << *(p[2]+1);
```

- Pointer of pointer

```
int a=4; int *p=&a; int **pp=&p; cout << **pp;
```

- Pointer reference

```
void func(char* &p);
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & pointer reference
- Dynamic memory allocation

Motivation

- In C/C++, the size of a statically allocated array has a limit

```
const unsigned int size = 0xffffffff;
int a[size];
```

- Sometime, we need to determine the array size at runtime

```
int size;
cin >> size;
int a[size];
```

Dynamic Memory Allocation

- Dynamic memory: memory that can be *allocated*, *resized*, and *freed* during program runtime.
- When do we need dynamic memory?
  1. when you need a very large array
  2. when we do not know how much amount of memory would be needed for the program beforehand.
  3. when you want to use your memory space more efficiently.
     - e.g., if you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory

Dynamic Memory Allocation

- Keywords: new & delete

```
// Declaration
int *p0 = new int(10); // init an integer 10 in memory, make p0 point to it
char *p1 = new char('a'); // init a char 'a' in memory, make p1 point to it

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete p0; // free the memory pointed by p0
delete p1; // free the memory pointed by p1

// Will be illegal after deletion
*p0 = 10;
```

Dynamic Memory Allocation

- Syntax on array: new [] and delete []

```
//  Declaration
int n; cin >> n;
int *p0 = new int[n]; //  allocate memory for an int array of n elements
char *p1 = new char[n]; //  allocate memory for a char array of n elements

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete[] p0; //  free the memory pointed by p0
delete[] p1; //  free the memory pointed by p1
```

The NULL pointer

- A special value that can be assigned to **any** type of pointer variable
  - e.g., int *a = NULL; double *b = NULL;
- A **symbolic constant** defined in standard library headers, e.g. <iostream>
- When assigned to a pointer variable, that variable points to nothing
- Initialization after declaration

```
int *ptr1 = NULL;
```

- Check null pointer before using the pointer:

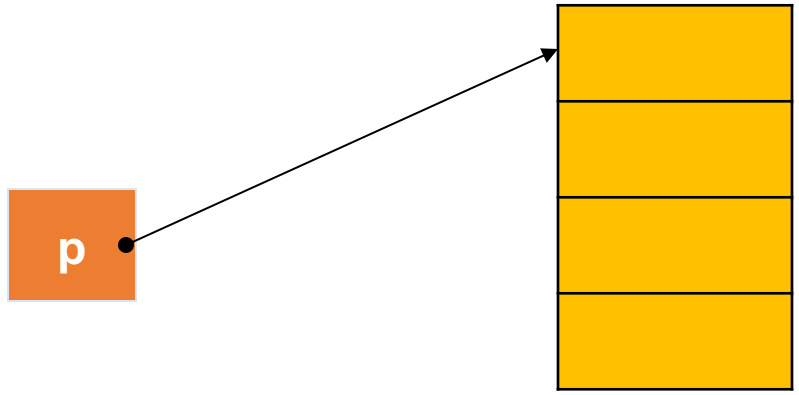
```
if (ptr)
if (!ptr)
```

Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

Dynamic Memory Walkthrough

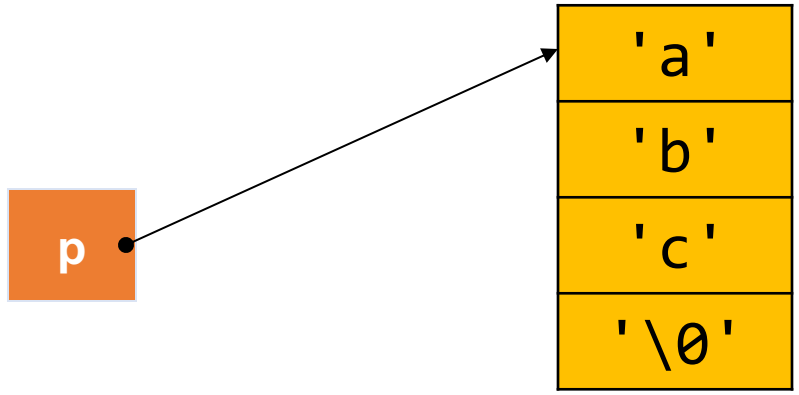
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 4 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

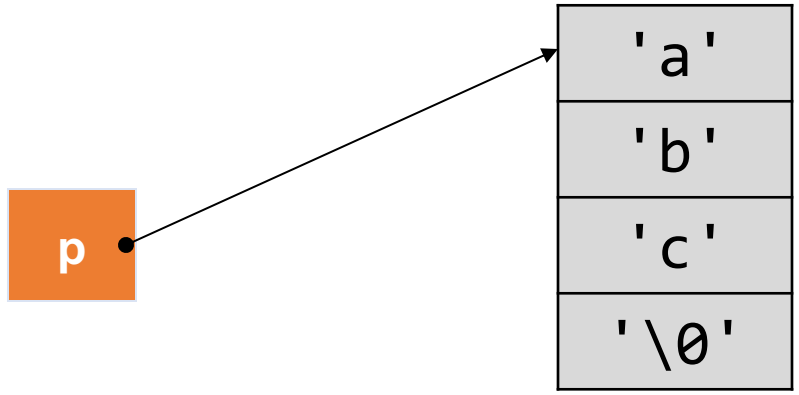
Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

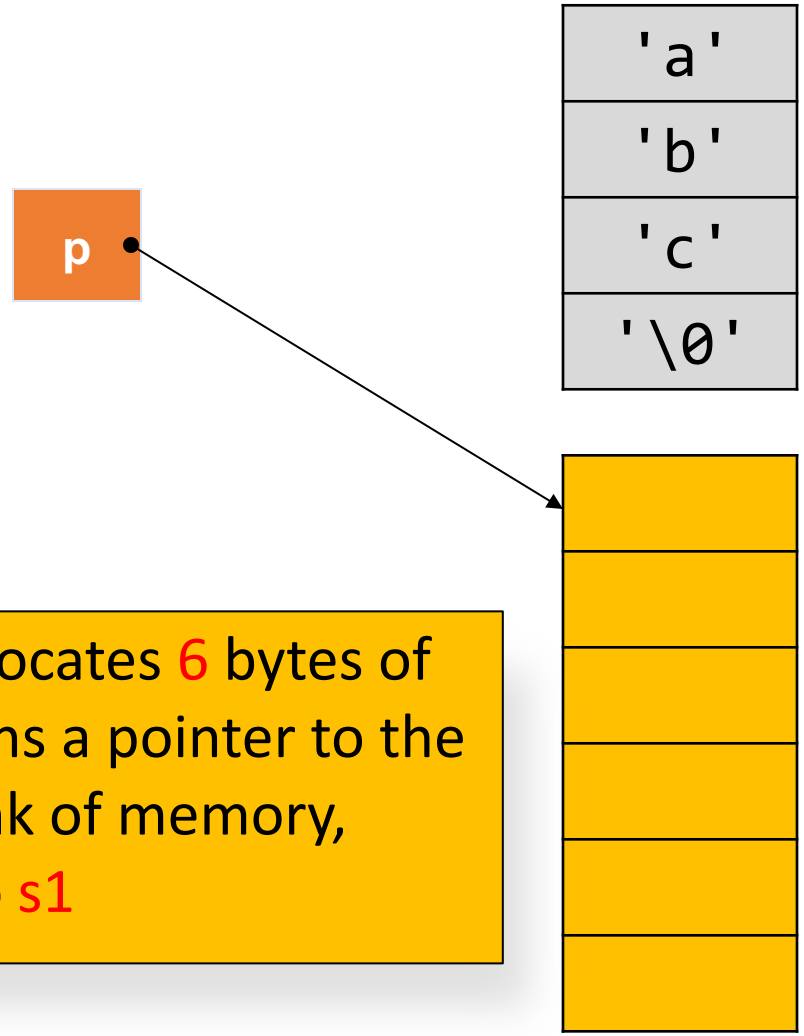


Grey memory means the block of memory is free and can be used to store other data.

p may or may not be pointing to the same address, and you can still print it, but that memory no longer belongs to p.

Dynamic Memory Walkthrough

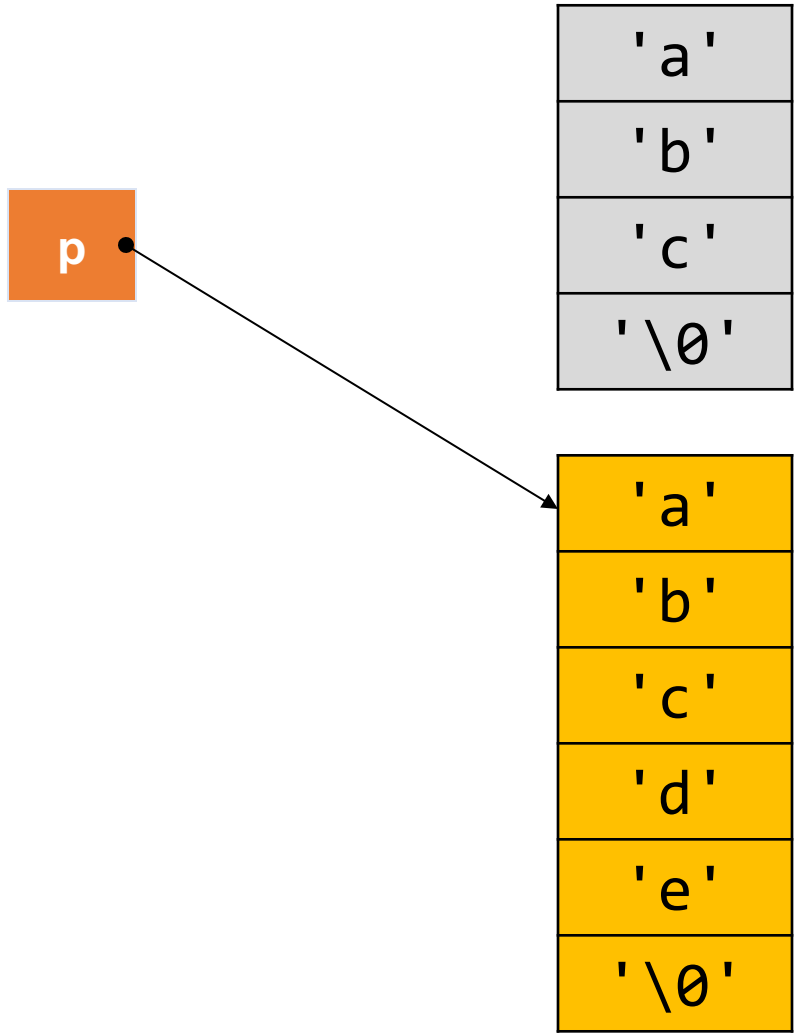
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 6 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

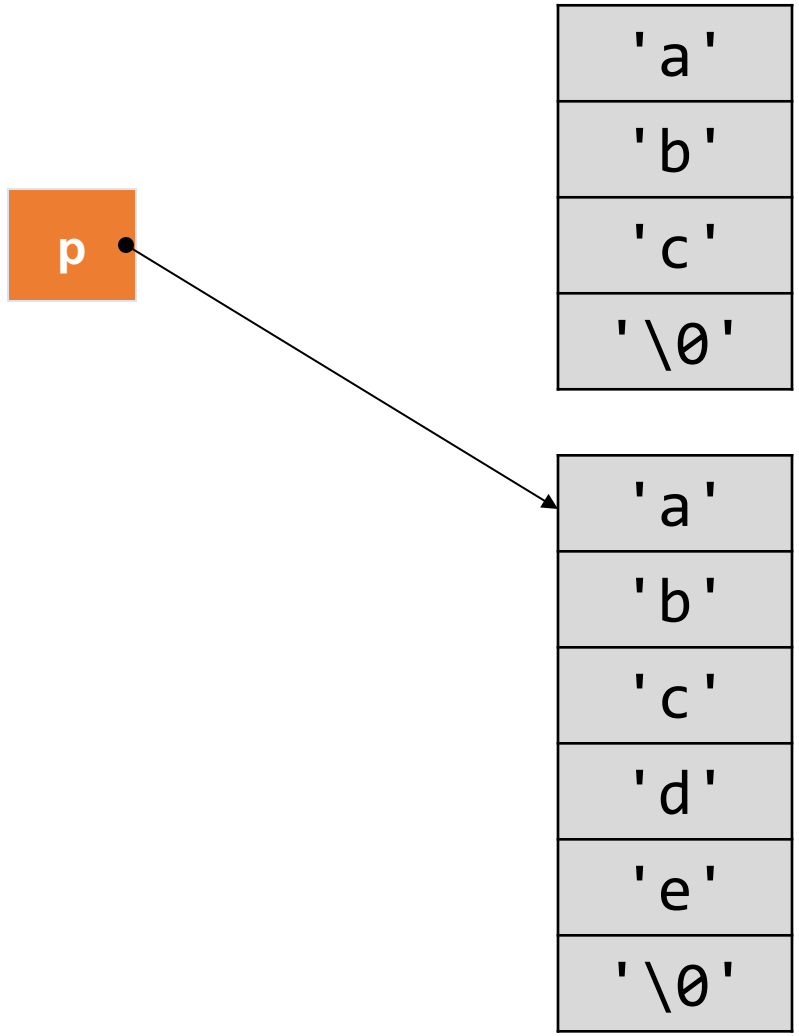
Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



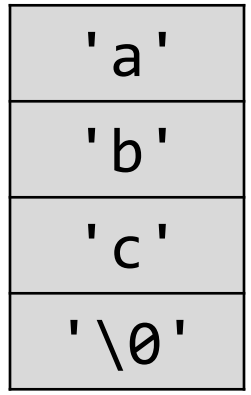
Dynamic Memory Walkthrough

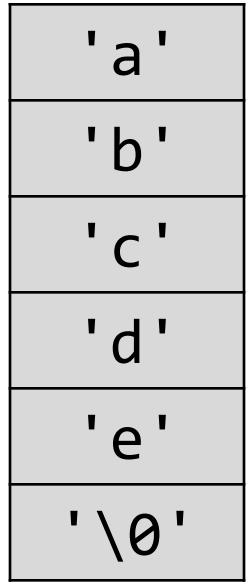
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL; // optional
```





Example

- score.txt contains the scores of 3 different courses for n students.
  - the first line of score.txt gives the value of n
  - reads all the scores, find all the students who have a failed score and output their scores for every course

- We can use dynamic memory allocation to solve the problem
  - As the number of the students is read from the input, we cannot define a normal 2D array (array size is not a constant).

score.txt :

```
4 3
85 89 64
93 82 94
55 92 59
59 88 70
```

```
ifstream fin("score.txt");
if (fin.fail())
  exit(1);
int n, m;
fin >> n >> m;
int **p = new int*[n];
for (int i = 0; i < n; i++) {
  p[i] = new int[m];
  for (int j = 0; j < m; j++)
    fin >> p[i][j];
}
fin.close();
```

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    if (p[i][j] < 60) {
      for (int k = 0; k < m; k++)
        cout << p[i][k] << ' ';
      cout << endl;
      break;
    }
  }
}
for (int i = 0; i < n; i++) {
  delete [] p[i];
}
delete[] p;
```