CS2311 Computer Programming

LT10: Pointers :

Arrays, Strings

& Dynamic Memory Allocation

Part II

Outline

- Access array elements via pointers
- Manage strings via pointers
- Dynamic memory allocation

The NULL pointer

- A special value that can be assigned to any type of pointer variable
- A symbolic constant defined in several standard library headers, e.g. **<iostream>**
- When assigned to a pointer variable, that variable points to nothing
- Example

```
int *ptr1 = NULL;
int *ptr2 = 0;
```


Operations on pointers

- Copying the address

```
p = q;
```

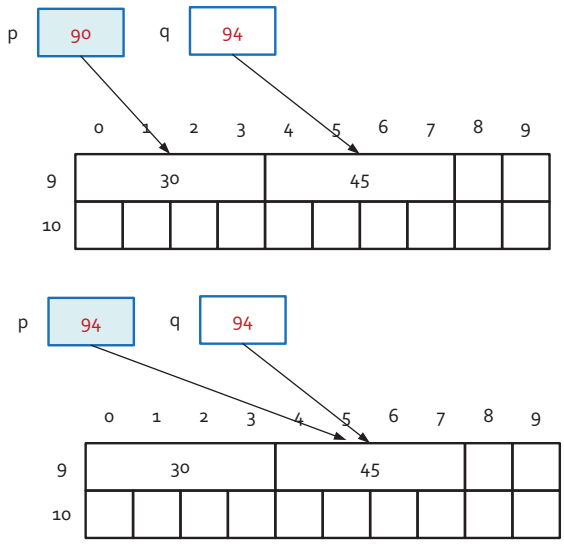
  - **p** and **q** point to the same variable
- Copying the content
  - Copy the value of the variable which is pointed by the **p** to the variable which is pointed by **q**

```
*p = *q;
```

  - **p** and **q** may point to different variables.

Copy the address

Assignment: p = q;



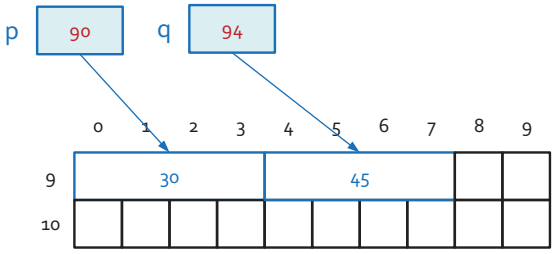
We copy the content (which is an address) of q to p.

After the assignment, p and q point to the same location in memory.

Therefore, if we change *p, *q will also be changed

Copy the content

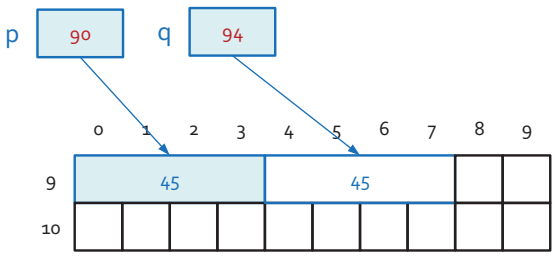
*p = *q;



We copy the value of the variable pointed by q to the variable pointed by p.

After the assignment, p and q still point to different locations in memory.

if we change *p, *q will not be changed as p and q points to different location in memory.

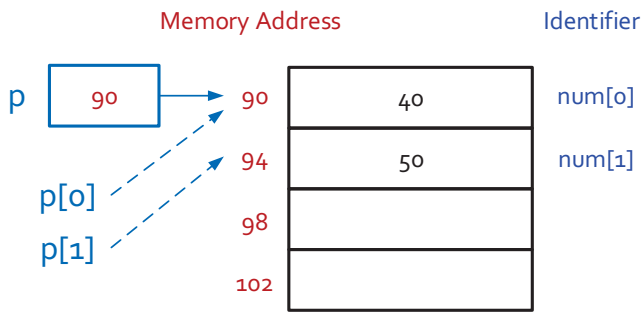


Relationship between arrays and pointers

```
int num[2] = {40,50};
num[0] = 400;
num[1] = 500;
```

Equivalent to

```
int num[2] = {40,50};
int *p;
p = num;
p[0] = 400; p[1] = 500;
```



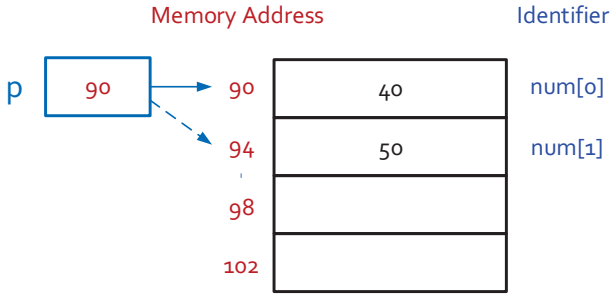
We can use array-like notation in pointers
num is a constant pointer to the first byte of the array;
The value of p can be changed.
p = num;
However, the value of **num** cannot be changed.
num = p; /*illegal*/

Relationship between arrays and pointers

```
int num[2] = {40,50};
int *p;
p = num;
p[0] = 400;
p[1] = 500;
```

Equivalent to

```
int num[2] = {40,50};
int *p;
p = num;  /* p points to 90 */
*p = 400;
++p;      /*p points to 94 */
*p = 500;
```



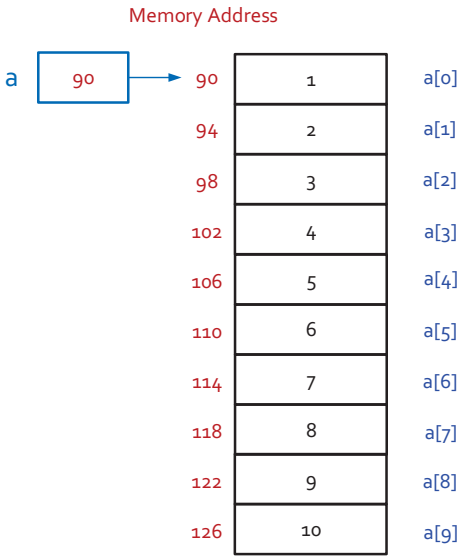
++p increments the content of p (an address) by sizeof(int) bytes

Arrays and pointers

Equivalent representation		Remark
num	&num[0]	num is the address of the 0^{th} 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

Example 2: Summing an array

```
#define N 10
int main(){
    int a[N] = {1,2,3,4,5,6,7,8,9,10};
    int i, sum = 0;
    for (i = 0; i < N; ++i)
        sum += *(a + i);
    cout << sum; /*55 is printed*/
    return 0;
}
```



a+1 is the address of a[1]
a+2 is the address of a[2]
…
a+i is the address of a[i]

So, *(a+i) means a[i]

Passing arrays to functions

- When an array is being passed, its base address is passed; the array elements themselves are not copied
  - → This is call-by-reference

- As a notational convenience, the compiler allows array bracket notation to be used in declaring pointers as parameters, e.g. (next page)

  double sum(int *array); is the same as

  double sum(int array[]);

Example 3: Parameter Passing

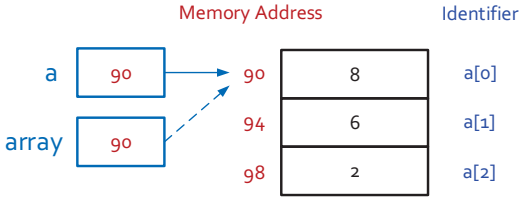
```
/* Compute the mean value */
#include <iostream>
using namespace std;
#define  N  5
double sum(int *);
int main() {
    int a[N] = {8,6,2,7,1};
    double mean;
    mean = sum(a)/N;
    cout << "mean = " << mean << endl;
    return 0;
}
```

```
double sum(int *array) {
    int i;
    double total = 0.0;

    for (i=0; i<N; i++)
        total += array[i];

    return total;
}
```

When **sum(a)** is called, the content of **a** (address of **a[0]**) is assigned to the pointer array. Therefore the pointer array points to **a[0]**.

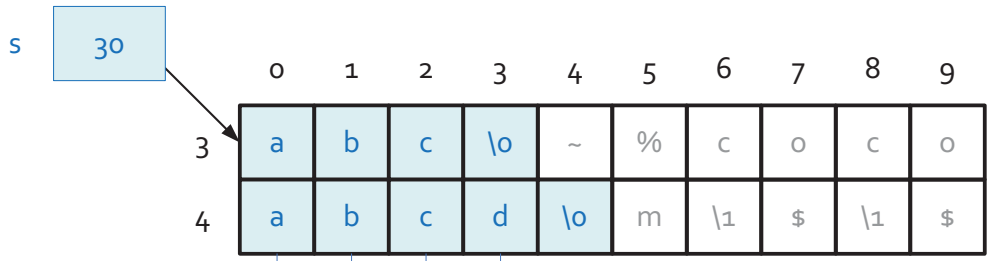


When **array** is passed as parameters, call-by-reference is used. If we modify array[i] in **sum**, a[i] is also modified in **main**

Arrays, pointers and strings

```
char s[] = "abc";
s = "abcd"; // illegal
```

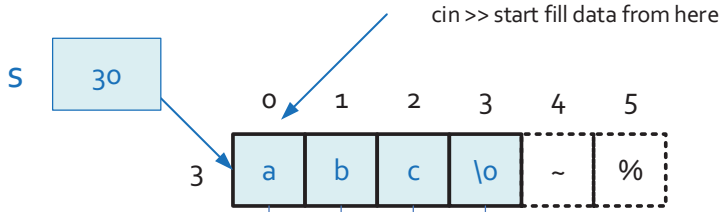
Illegal as **s** is a constant pointer and cannot be modified



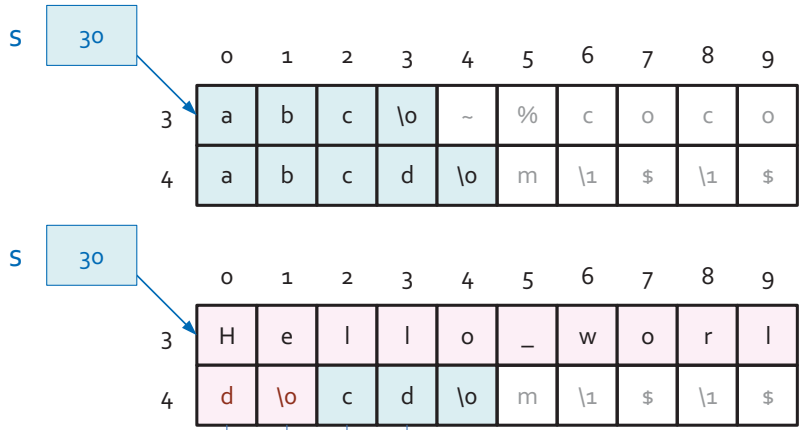
cin >> a string (I)

```
char s[] = "abc";
cin >> s;
```

input: **Hello_World**



cin >> a string (I)



Size of **s** is **4**. Array out-of-bound!
cin >> does not perform bound-checking
Better to use:
cin.getline(s,4); /*read at most 3 characters*/
Remember to leave space for the final **'\0'** character

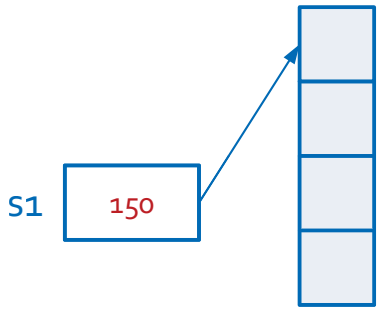
cin >> a string (II)

```
#include <iostream>
using namespace std;
int main () {
    char *s1;
    cin >> s1;
    cout << s1;
    return 0;
}
```

Problem: when we declare the pointer **s1**, we do not know where **s1** points to.
In this example,
we try to read a string and store it in the location pointed by **s1**.
This may generate errors as we may overwrite some important locations in memory

Dynamic Memory Allocation

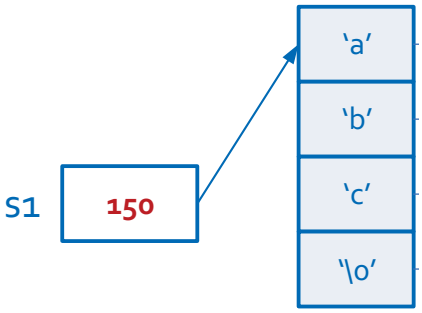
```
#include <iostream>
int main () {
    char *s1;
    s1 = new char[4];
    cin >> s1; /*input "abc"*/
    cout << s1;
    delete s1;
    s1 = new char(6);
    cin >> s1;
    cout << s1;
    delete s1;
    s1 = NULL;
    return 0;
}
```



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

Example 4: Dynamic memory allocation

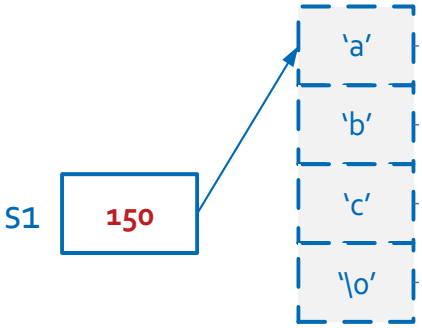
```
#include <iostream>
int main () {
    char *s1;
    s1 = new char[4];
    cin >> s1;   /*input "abc"*/
    cout << s1;
    delete s1;
    s1 = new char(6) ; // same as char[6]
    cin >> s1;
    cout << s1;
    delete s1;
    s1 = NULL;
    return 0;
}
```



Example 4: Dynamic memory allocation

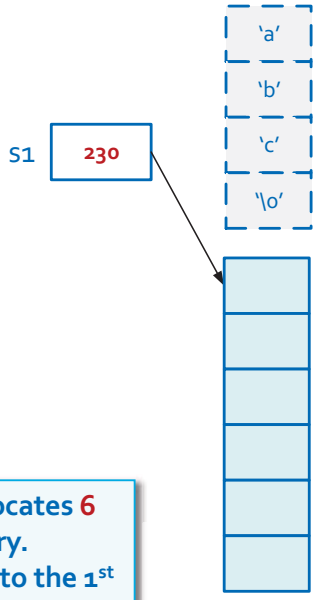
```
#include <iostream>
int main () {
    char *s1;
    s1 = new char[4];
    cin >> s1;   /*input "abc"*/
    cout << s1;
    delete s1;
    s1 = new char(6) ; // same as char[6]
    cin >> s1;
    cout << s1;
    delete s1;
    s1 = NULL;
    return 0;
}
```

Memory is free and can be used to store other data

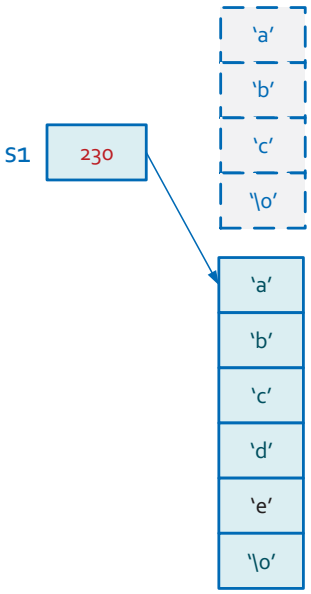



```
#include <iostream>
int main () {
    char *s1;
    s1 = new char[4];
    cin >> s1;   /*input "abc"*/
    cout << s1;
    delete s1;
    s1 = new char(6) ; // same as char[6]
    cin >> s1;
    cout << s1;
    delete s1;
    s1 = NULL;
    return 0;
}
```

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



```
#include <iostream>
int main () {
    char *s1;
    s1 = new char[4];
    cin >> s1;   /*input "abc"*/
    cout << s1;
    delete s1;
    s1 = new char(6) ; // same as char[6]
    cin >> s1;
    cout << s1;
    delete s1;
    s1 = NULL;
    return 0;
}
```



cin.ignore() before cin.getline()

```
int main() {
    char *s2 = NULL;
    int size;

    cout << "Tell me the size of the string: ";
    cin >> size;
    s2 = new char[size+1];

    cout << "Now enter a string with at most " << size << " characters: ";
    cin.ignore();  // it absorbs the newline char
    cin.getline(s2, size+1);

    cout << "You entered:\n" << s2 << endl;
    delete s2;

    return 0;
}
```

cin.ignore()

- For **std::cin** statements, you use **ignore()** before you do a **getline()** call
- When a user inputs something with **std::cin**, they hit enter and a **'\n'** char gets into the **cin** buffer. Then if you use **getline()**, it gets the newline char instead of the string you want
- But **cin** itself doesn't have this issue…
- More info:
  - https://stackoverflow.com/questions/25475384/when-and-why-do-i-need-to-use-cin-ignore-in-c
  - http://www.cplusplus.com/reference/istream/istream/ignore/

Guidelines on using pointers

- Initial a pointer to NULL after declaration

```
char *cPtr=NULL;
```

- Check its value before use

```
if (cptr!=NULL) {
        …
}
```

- Free the memory allocated by the "new" operator using delete

```
cPtr = new char[6]; …
delete cPtr;
```

- Set it NULL again after free

```
delete cPtr;
cPtr=NULL;
```

Summary

- Pointers can be used to access array element.
- Array name is a pointer pointing to the first element of the array.
- A string is stored as an array of characters.
- Strings must be terminated by an '\0' character, therefore a string with 5 characters will take up 6 characters space.
- Operator new allocates memory space and returns a pointer pointing to the newly allocated space.
- Memory obtained by new must be deleted after use.
- Extra care must be taken when handling pointers, as it may point to an invalid / unexpected location and make the program crashed.