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

```
char* chptr = NULL;
int* iptr = NULL;
double* dptr = NULL;
```

- 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;  // assume p & q are pointers to a data type
```

  - **p** and **q** points to the *same* variable

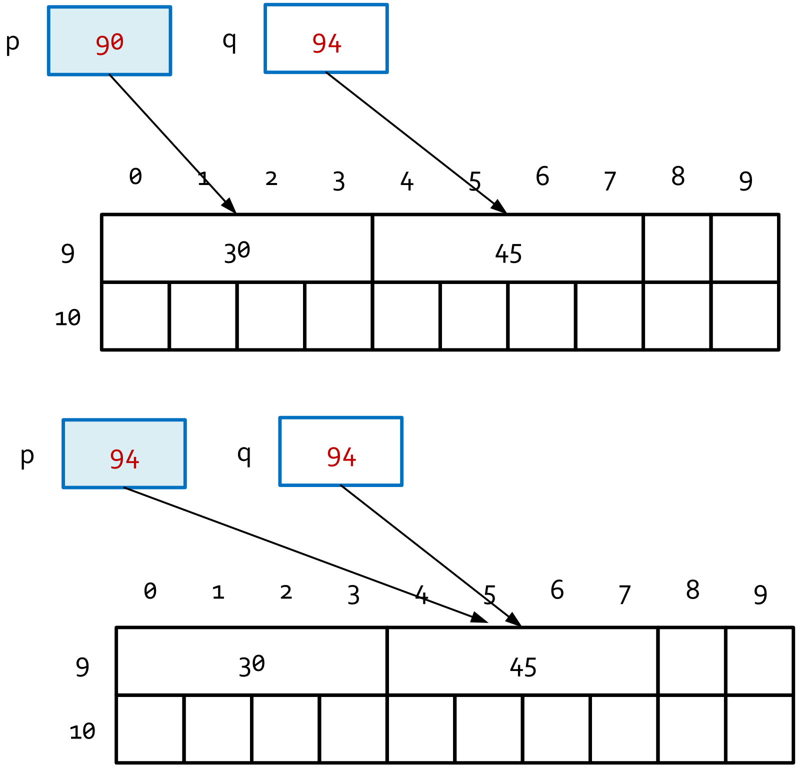
Copy the Address

Assignment: **p = q;**

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

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

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



Operations on Pointers

- Copying the address

  ```
  p = q;  // assume p & q are pointers to a data type
  ```

  - **p** and **q** points 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.

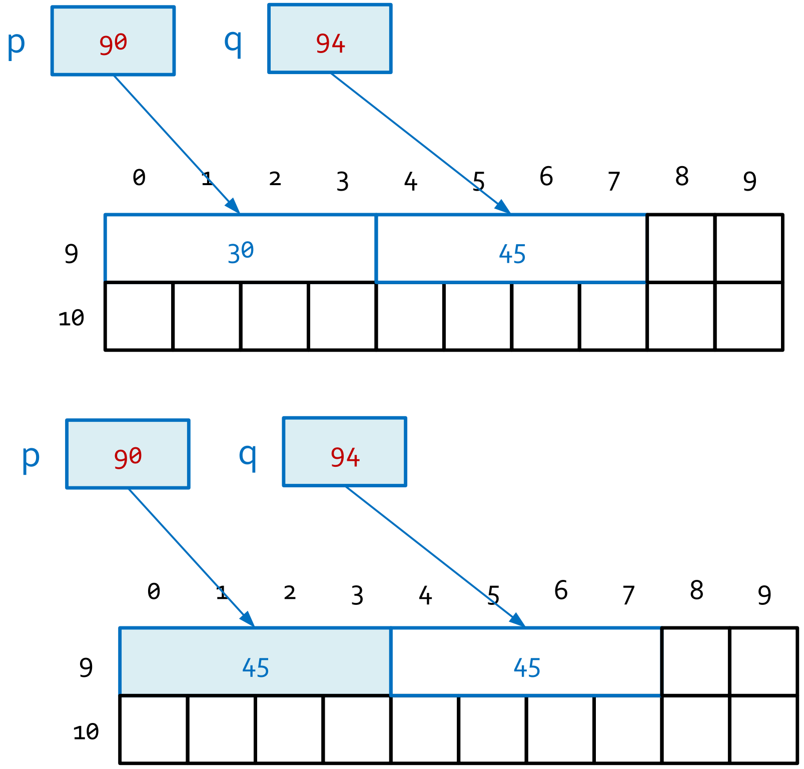
Copy the Content

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

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

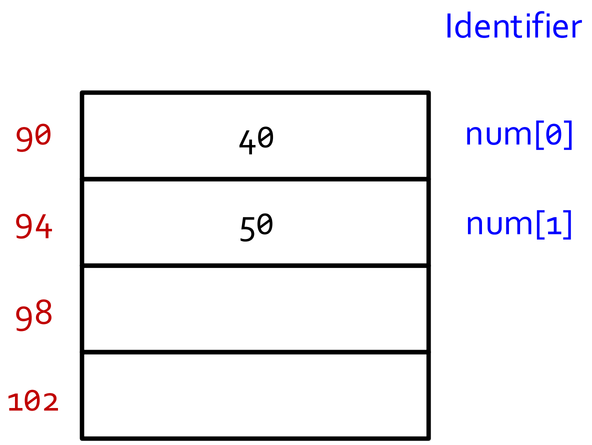
2. After the assignment, p and q still point to different locations in memory.

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



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.

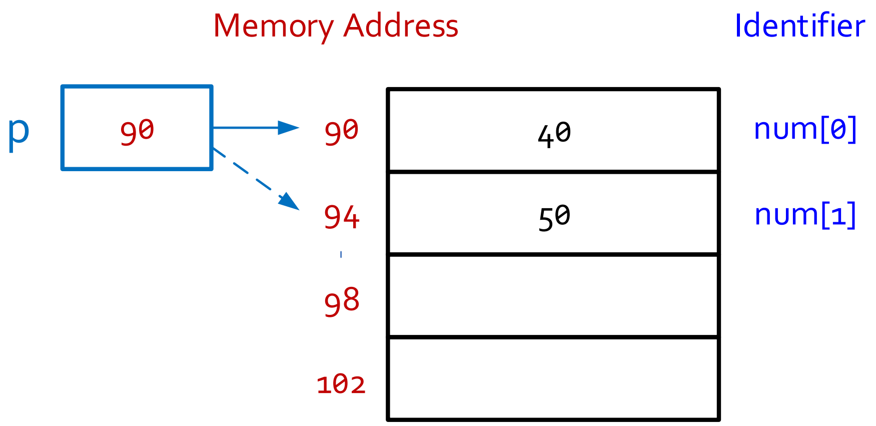
```
        p = num;
```

However, the value of **num** cannot be changed.

```
        num ✗= p; // illegal
```

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 ith 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 ith 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

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

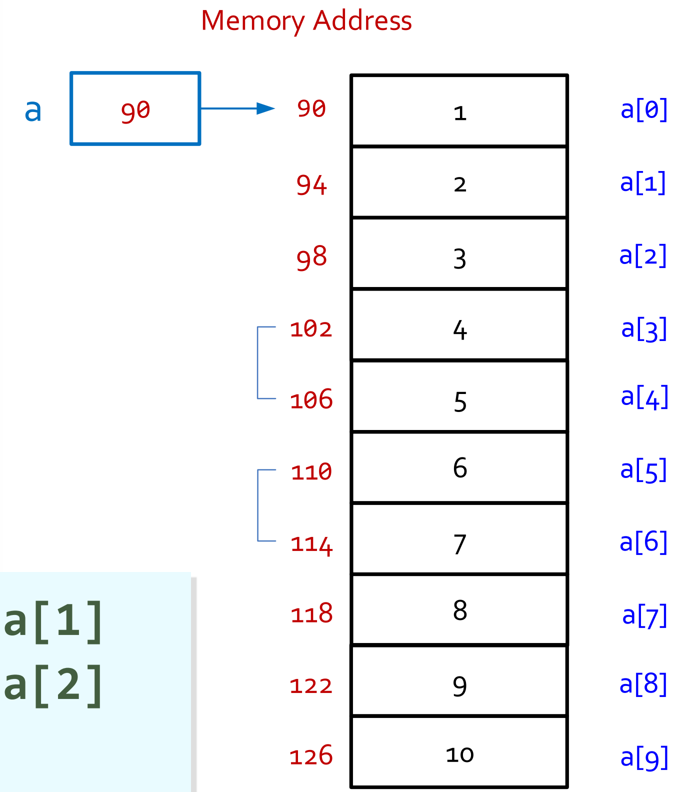
`a+1` is the address of `a[1]`

`a+2` is the address of `a[2]`

…

`a+i` is the address of `a[i]`

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



Passing an Array to a Function

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

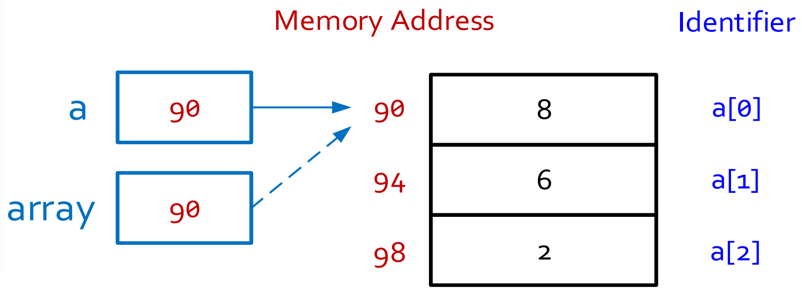
- As a notational convenience, the compiler allows *array bracket notation (indexing)* to be used in declaring pointers as parameters
  - example:

```
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;
const int 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) {
   double total = 0.0;
   for (int i=0; i<N; i++)
       total += array[i];
    return total;
}
```



		Memory Address		Identifier
a	90	90	8	a[0]
array	90	94	6	a[1]
		98	2	a[2]

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 an ***array*** is passed as parameters, call-by-reference is used.

If we modify **array[i]** in **sum**, **a[i]** is also modified in **main**.

Access Elements in 2D Array with Pointers

- We can use a point to access a 2D array

- For a 2D array **`int a[4][3]`**, **`a[i]`** (*i=0,1,2,3*) is the address of the *first element* in the *i-th row*

- For *each row*, it is equivalent to one ***1D array***

- We can declare a pointer **`int *p = a[i]`** to access every element on the *i-th row*

Access Elements in 2D Array with Pointers

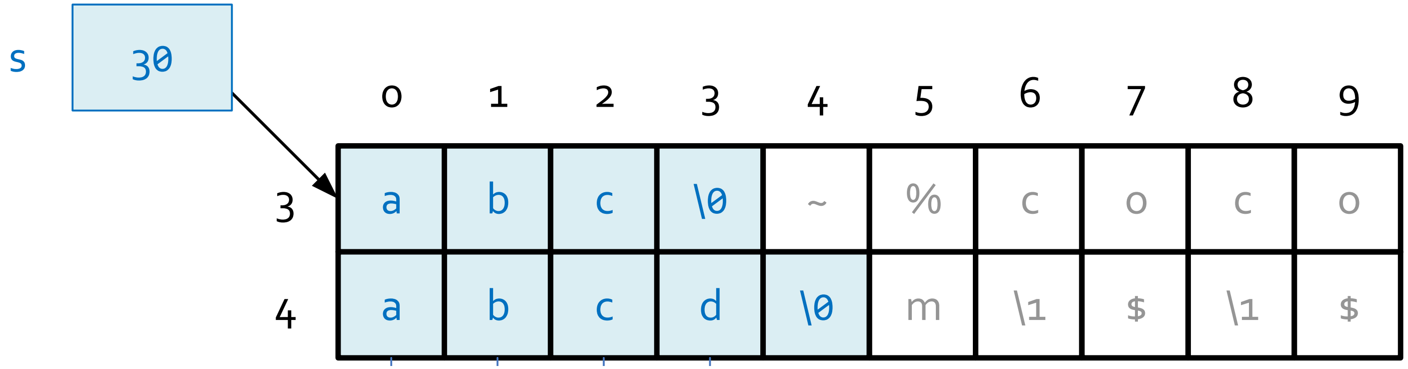
- Like the 1D array, we use the '*' sign to access the elements in one 2D array

```
int a[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
int *p = a[0];
for(int i=0; i<12; i++) {
   cout << *p << endl;
   p++;
}
```

Arrays, Pointers and Strings

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

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

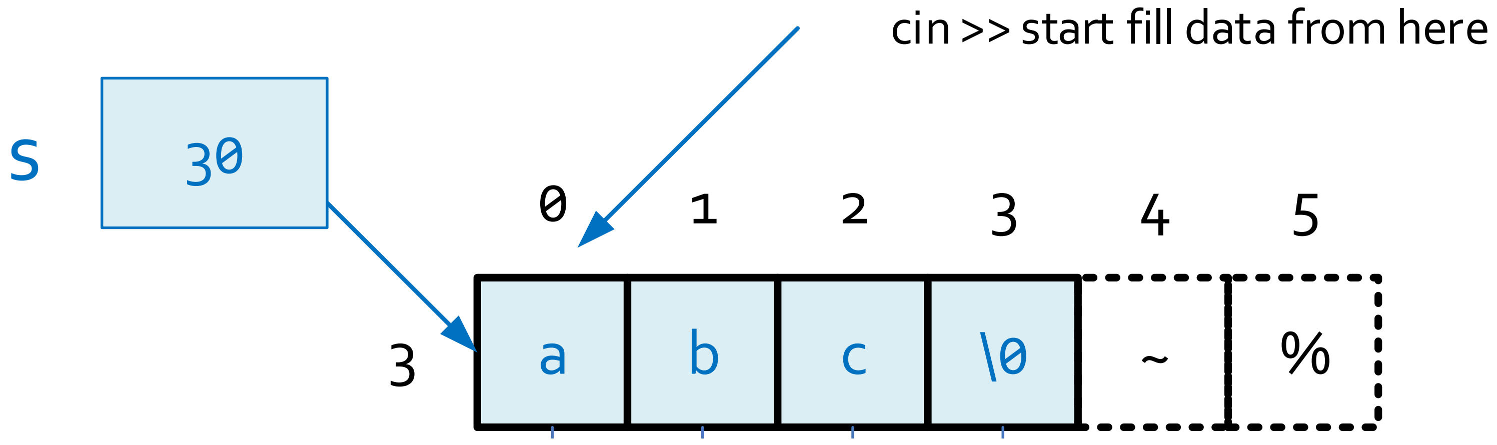


	0	1	2	3	4	5	6	7	8	9
3	a	b	c	\0	~	%	c	o	c	o
4	a	b	c	d	\0	m	\1	$	\1	$

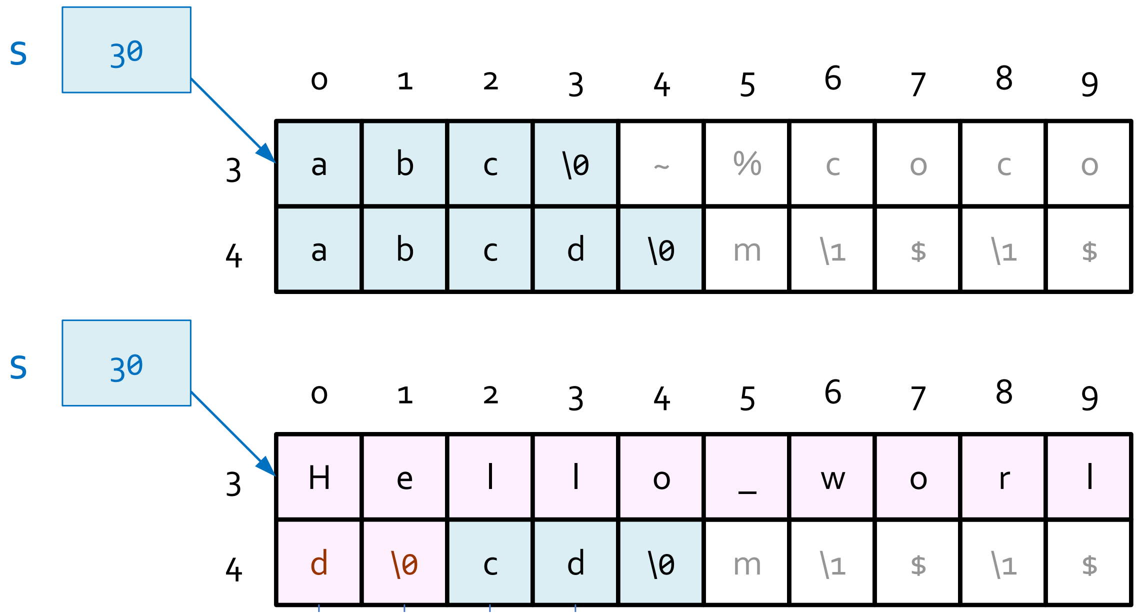
cin >> a String 1

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

input: Hello_World



cin >> a String 1



s 30

	0	1	2	3	4	5	6	7	8	9
3	a	b	c	\0	~	%	c	o	c	o
4	a	b	c	d	\0	m	\1	$	\1	$

s 30

	0	1	2	3	4	5	6	7	8	9
3	H	e	l	l	o	_	w	o	r	l
4	d	\0	c	d	\0	m	\1	$	\1	$

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.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/

cin.ignore() Before cin.getline()

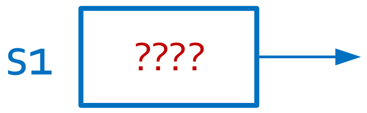
```
int main() {
   char *s2 = NULL;
   int size;
   cout << "Tell me the size of the string: ";
   cin >> size;

   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:" << endl << s2 << endl;
   delete s2;
   return 0;
}
```

cin >> a String 2

```
#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

- Keywords: **new** & **delete**

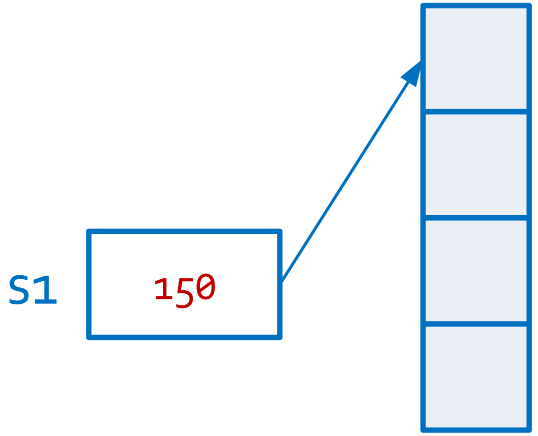
```
int *p = new int;
int *p = new int(10);
char *p = new char('a');
delete p;
*p = 10; // illegal
```

- Keywords: **new []** & **delete []**

```
int *p = new int [20];
char *q = new char[20];
delete [] p;
delete [] q;
```

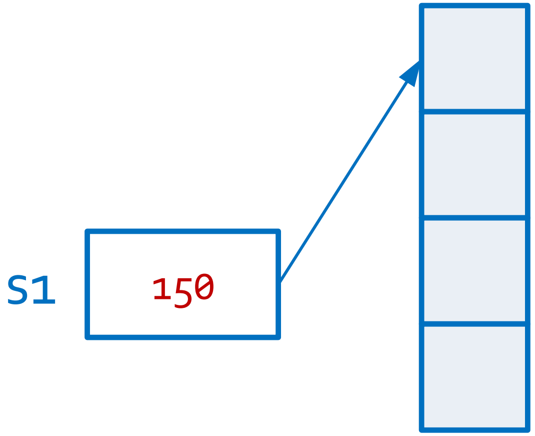
Dynamic Memory Allocation

```
#include <iostream>
int main () {
  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;
  return 0;
}
```



Dynamic Memory Allocation

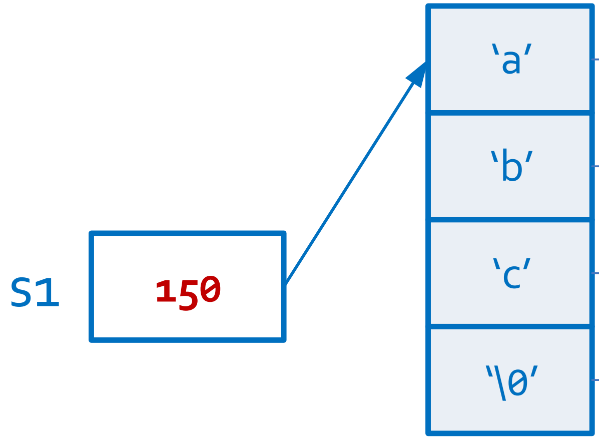
```
#include <iostream>
int main () {
  char *s1 = NULL;
  s1 = new char[4];
  cin >> s1; // input "abc"
  cout << s1;
  delete [] s1;
  s1 = new char[6];
  cin >> s1;
  cout << s1;
  delete [] s1;
  s1 = NULL;
  return 0;
}
```



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

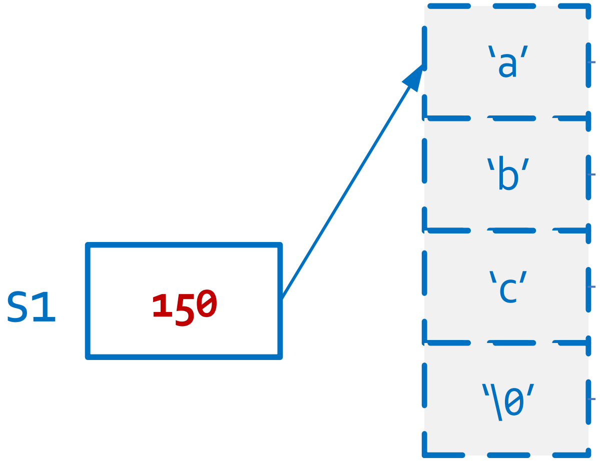
Example 4: Dynamic memory allocation

```
#include <iostream>
int main () {
   char *s1 = 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;
   return 0;
}
```



Example 4: Dynamic Memory Allocation

```
#include <iostream>
int main () {
   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;
   return 0;
}
```

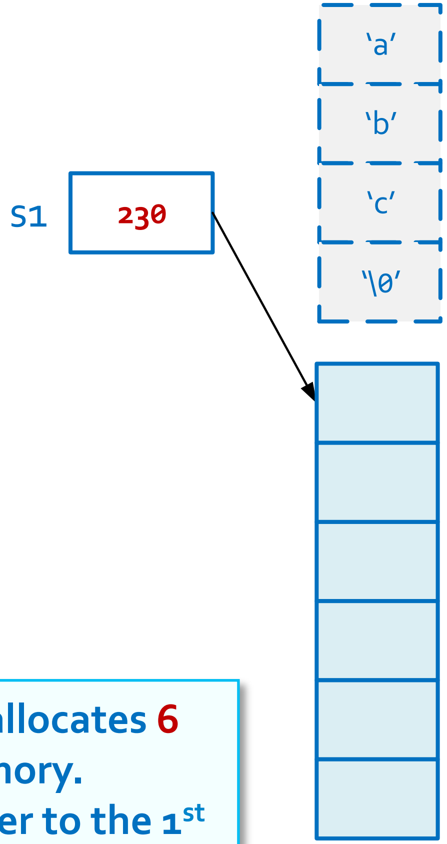


Memory is free and can be used to store other data

Example 4: Dynamic Memory Allocation

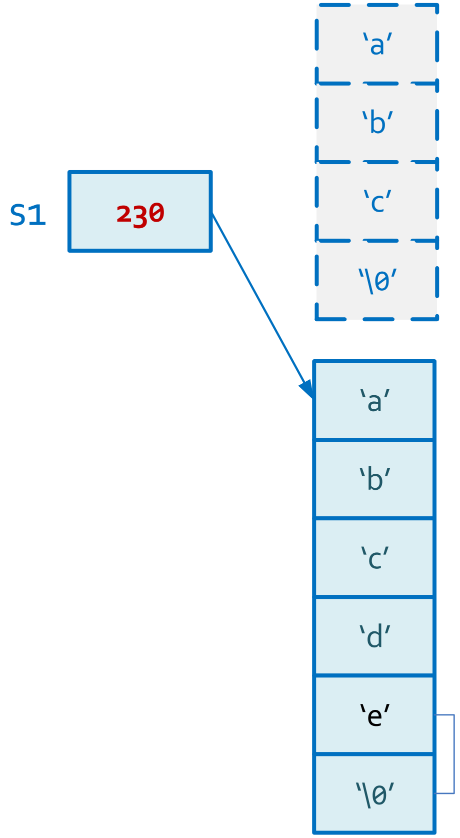
```
#include <iostream>
int main () {
  char *s1 = NULL;
  s1 = new char[4];
  cin >> s1; // input "abc"
  cout << s1;
  delete [] s1;
  s1 = new char[6];
  cin >> s1;
  cout << s1;
  delete [] s1;
  s1 = NULL;
  return 0;
}
```

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



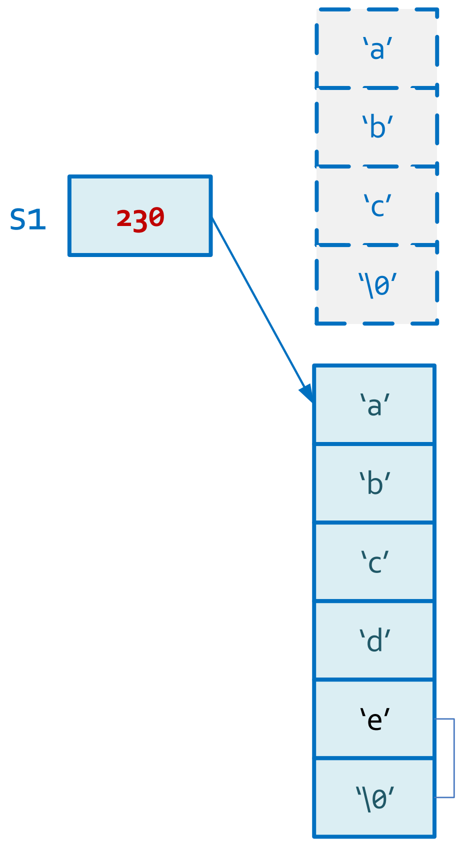
Example 4: Dynamic Memory Allocation

```
#include <iostream>
int main () {
  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;
  return 0;
}
```



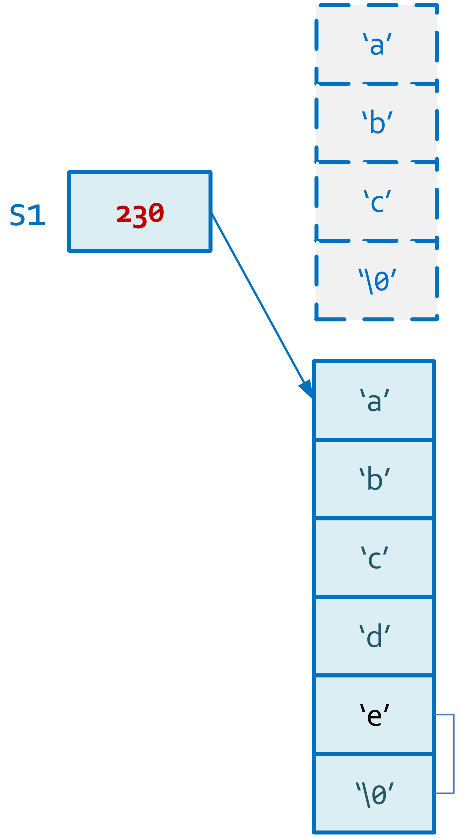
Example 4: Dynamic Memory Allocation

```
#include <iostream>
int main () {
  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;
  return 0;
}
```



Example 4: Dynamic Memory Allocation

```
#include <iostream>
int main () {
   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;
   return 0;
}
```



Example Use of Dynamic Memory Allocation

- The input file **scores.txt** contains the scores of 3 different courses for ***n*** students.
  - The first line of **scores.txt** gives the value of ***n***
  - Reads all the scores, find all the students who have a failed score (**score < 60**) and output their scores for every course
- As the number of the students is read from the input, we cannot define a normal 2D array *(array size is not a constant)* Hence, we can use *dynamic memory allocation* to solve the problem

scores.txt

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

Function check_score()

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

void check_score() {

   ifstream in("scores.txt");
   if (in.fail()) {
      exit(1);
   }
   int n;
   in >> n;
   int** p = new int* [n];
   for (int i = 0; i < n; i++) {
      p[i] = new int [3];
      for (int j = 0; j < 3; j++)
         in >> p[i][j];
   }
   in.close();
   // check grade

}
```

```
for (int i = 0; i < n; i++) {
   bool fail = false;
   for (int j = 0; j < 3 && !fail; j++)
      if (fail = (p[i][j] < 60)) {
         for (int k=0; k<3; k++)
            cout << p[i][k] << ' ';
         cout << endl;
      }
}
for (int i = 0; i < n; i++)
   delete [] p[i];
delete [] p;
```

scores.txt

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

Guidelines on using Pointers

- **Initialise** 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 **`delete`**d after use.
- **Extra care** must be taken when handling pointers, as it may point to an invalid / unexpected location and make the program crashed.