CS2313 Computer Programming

LT10 – Pointer

Outlines

- Memory and variable
- Pointer and its operation
- Call by reference
- Access array elements via pointer
- Manage string via pointer
- Dynamic memory allocation

Syntax Summary

- Punctuators
  - * (asterisk),&(ampersand)
- Constant
  - NULL

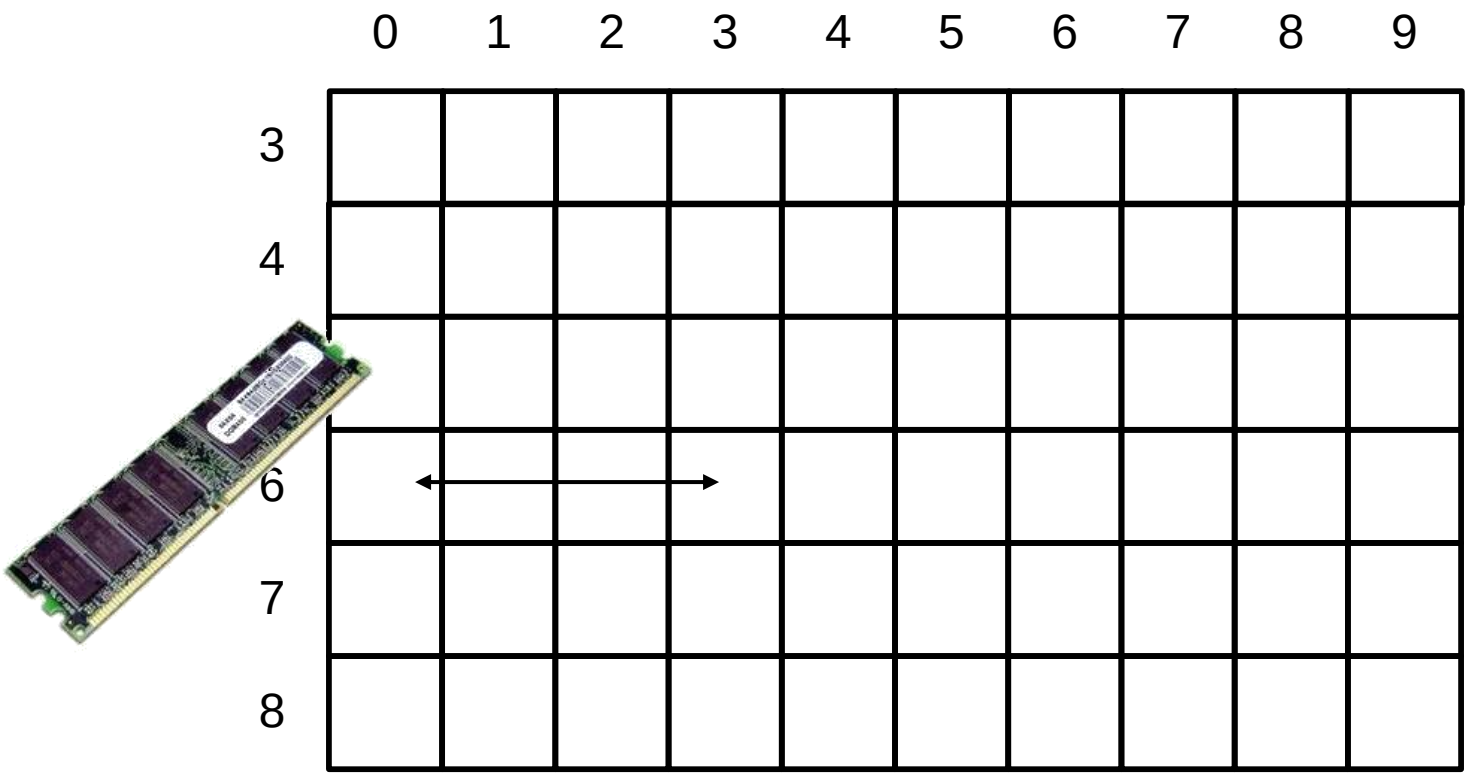
Memory and Variable

- Variable is used to store data that will be accessed by a program on execution.

- Normally, variables are stored in the main memory.

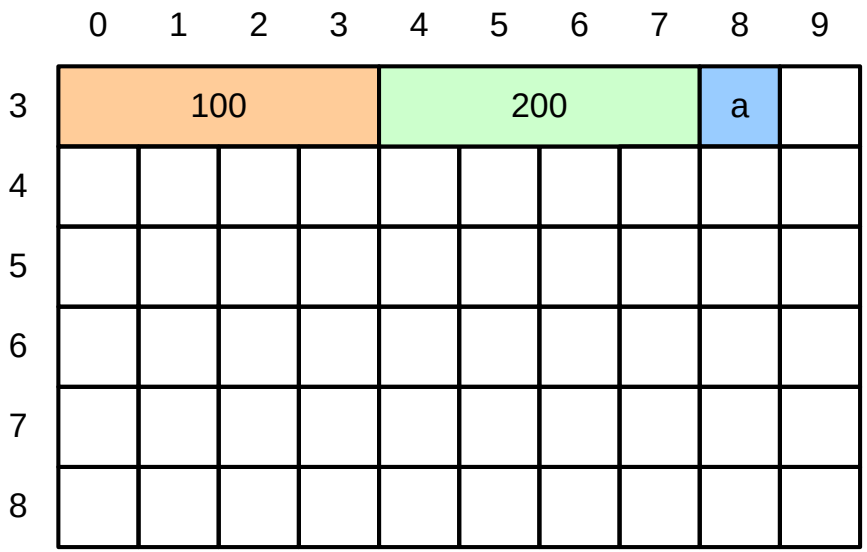
- A variable has four attributes:
  - Value - the content of the variable.
  - Identifier - the name of the variable.
  - Address - the memory location of the variable.
  - Scope - the accessibility of the variable.

Main Memory



Variable and Memory

```
void main(){
    int x;
    int y;
    char c;
    x=100;
    y=200;
    c='a';

}
```



Identifier	Value	Address
x	100	30
y	200	34
c	'a'	38

Variable and Memory

- *Most of the time*, the computer allocates adjacent memory locations for variables declared one after the other.
- A variable's address is the first byte occupied by the variable.
- Memory address of a variable depends on the computer, and is usually in hexadecimal (base 16 with values 0-9 and A-F).
  - e.g. 0x00023AF0, 0x00023AF0.

Pointer

- In C++, a pointer is a variable which is designed to store the address of another variable. When a pointer stores the address of a variable, we said the pointer is pointing to the variable.

- Pointer, like normal variable has a type, its type is determined by the type of variable it points to.

Variable type	int	float	double	char
Pointer type	int*	float*	double*	char*

* And & operator

- To declare a pointer variable, place a "*" sign before an identifier name:
  - `char *cPtr;  //a character pointer`
  - `int *nPtr;//a integer pointer`
  - `float *fp;//a floating point pointer`

- To retrieve the address of a variable, use the "&" operator:
  - `int x;`
  - `nPtr = &x; // &x will return the address of variable x;`

- To access the variable a pointer pointing to, use "*" operator (dereference):
  - `*nPtr = 10;`
  - `Y = *nPtr;`

Example

```
int x,y;       //x and y are integer variables

void main(){
  int *p1,*p2;   /*p1 and p2 are pointers of integer typed */

  x=10;
  y=12;

  p1=&x;       /* p1 stores the address of variable x */
  p2=&y;       /* p2 stores the address of variable y */

  *p1=5;       /* p1 value unchanged but x is updated to 5 */
  *p2=*p1+10; /*what are the values of p2 and y? */
}
```

Common Operations

- Set a pointer *p1* point to a variable *x*
  - `p1=&x;`
- Set a pointer *p1* point to the variable pointed by another pointer *p2*
  - `p1=p2`
- Update the value of variable pointed by a pointer
  - `*p=10;`
- Retrieve the value of variable pointed by a pointer
  - `int x=*p;`

More Examples

```
x=3;
y=5;
p1=&x;
p2=&y;
y=*p1-*p2;
p1=p2;
//*p1=-2;*p2=-2
//x=3,y=-2
y=*p1+*p2;
x=*p1+1;
//*p1=-4;*p2=-4
//x=-3;y=-4
```

Guideline

- * operator will give the value of pointing variable.

- & operator will give the address of a variable.

Applications of Pointer

- Call by reference.
- Fast array access
- Dynamic memory allocation
  - Require additional memory space for storing value.
  - Similar to variable declaration but the variable is stored outside the program.

Call-by-Reference

- Pass the address of a variable to a function.
- To update a variable provided by a caller since call-by-value cannot be used to update parameters to the function.
- Consider the following function which gets the inputs from keyboard.

```
void getDimension(float width, float height){
    cout << "Please enter the width of rectangle:";
    cin >> width;
    cout << "Please enter the height of rectangle:";
    cin >> height;
}
void main(){
    float  w, h;
    getDimension(w,h);
    cout << "area=" << w*h << endl;
}
```

Call-by-Value

```
void f (char c1_in_f){
     c1_in_f='B'; //c1_in_f=66
}

void main(){
     char c1_in_main='A'; // c1_in_main =65
     f(c1_in_main);
     cout << c1_in_main; //print 'A'
}
```

Call-by-Value

- When calling `f()`, the value (65 or 'A') is assigned to `c1_in_f`.
- Inside `f()`, the value of `c1_in_f` is changed to 66 or 'B'.
- `c1_in_main` remains unchanged (65 or 'A').
- In call-by-value, there is no way to modify `c1_in_main` inside `f()`, unless we use the `return` statement.

Call-by-Reference

```
void f (char *c1_ptr){
    *c1_ptr='B';
}

void main(){
    char c1_in_main='A'; // c1_in_main =65
    f(&c1_in_main);
    cout << c1_in_main; //print 'B'
}
```

When `f()` is called, the following operation is performed

`c1_ptr = &c1_in_main;`

First, the pointer value (address) `&c1_in_main` (which is 3000, in this example) is retrieved.

Call-by-Reference

```
void f (char *c1_ptr){
    *c1_ptr='B';
}

void main(){
    char c1_in_main='A'; // c1_in_main =65
    f(&c1_in_main);
    cout << c1_in_main; //print 'B'
}
```

points to

Variable	Variable type	Memory location	Content
c1_in_main	char	3000	65
c1_ptr	char pointer	4000	3000

Assign location 3000 to c1_ptr

Location of c1_in_main (location 3000) is assigned to c1_ptr1

c1_ptr = **&c1_in_main**;

Call-by-Reference

```
void f (char *c1_ptr){
    *c1_ptr='B';
}

void main(){
    char c1_in_main='A'; // c1_in_main =
    f(&c1_in_main);
    cout << c1_in_main; //print 'B'
}
```

update

`c1_ptr` points to location 3000 (that is the variable `c1_in_main`).
`*c1_ptr` refers to the variable pointed by `c1_ptr`, i.e. the variable stored at address 3000.

Variable	Variable type	Memory location	Content
c1_in_main	char	3000	66
c1_ptr	char pointer	4000	3000

```
c1_ptr='B'; //error
```

Reason: c1_ptr1 stores a location so it cannot stores a char (or the ASCII code of a char).

Note the Different Meaning of *

The type of `c1_ptr` is char* (pointer to char)

dereference a pointer

```
void f (char *c1_ptr){
    *c1_ptr='B';
}

void main(){
    char c1_in_main='A'; // c1_in_main =65
    f(&c1_in_main);
    cout << c1_in_main; //print 'B'
}
```

Exercise – What Are the Errors?

```
int x=3;
char c='a';
char *ptr;
ptr=&x;
ptr=c;
ptr=&c;
```

Answer

```
int x=3;
char c='a';
char *ptr;
ptr=&x; //error: ptr can only points to a char,
   but not int
ptr=c; //error: cannot assign a char to a pointer.
   A pointer can only store a location
ptr=&c; //correct
```

Exercise – What Is the Output?

```
int num=100;
int *ptr1;
ptr1=&num;
*ptr1=40;
cout << num;
```

Answer

```
int num=100;
int *ptr1;
ptr1=&num;
*ptr1=40;
cout << num; //print 40
```

Call-by-Value and Call-by-Reference

- In call-by-value, only a single value can be returned using a **return statement**.

- In call-by-reference, the parameter(s) can be a pointer which may reference or points to the variable(s) in the caller function.
  - More than one variables can be **updated**, achieving the effect of returning multiple values.

Call-by-Value

```
int add(int m, int n)
{
    int c;
    c = m+n;
    m = 10;
    return c;
}

void main()
{
    int a=3, b=5,c;
    c=add(a,b);
    cout << a<<" "<< c <<
    endl;
}
```

Pass value as parameter.

Call-by-Reference

```
int add(int *m, int *n)
{
    int c;
    c = *m + *n;
    *m = 10;
    return c;
}

void main()
{
    int a=3, b=5,c;
    c=add(&a,&b);
    cout << a<<" "<< c <<
    endl;
}
```

Pass address as parameter.

Call-by-Reference - Guideline

- Add the '*' sign to the function parameters that store the variable call by reference.
  - E.g.

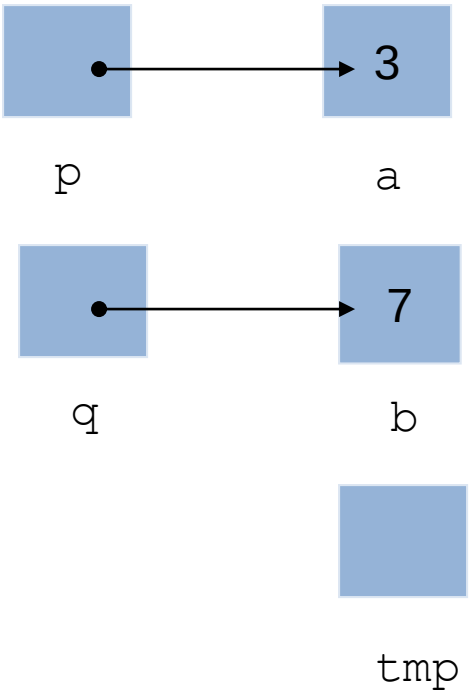
```
void cal(int *x, int y){
    //x is call by reference
    //y is call by value
    ……

}
```

- Add the '&' sign to the variable when it needs to be called by reference.
  - E.g.

```
cal(&result, factor);
```

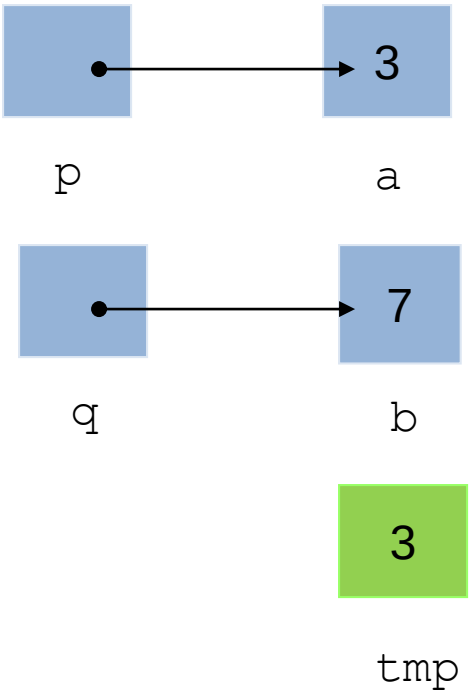
Example – Swapping Values

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

void swap(int *p, int *q) {
     int tmp;

     tmp = *p;        /* tmp = 3 */
     *p = *q;         /* *p = 7 */
     *q = tmp;        /* *q = 3 */
}

int main(void)
{
     int a = 3, b = 7;
     swap(&a, &b);
     cout << a <<" "<< b <<endl;
      /* 7  3 is printed */
     return 0;
}
```



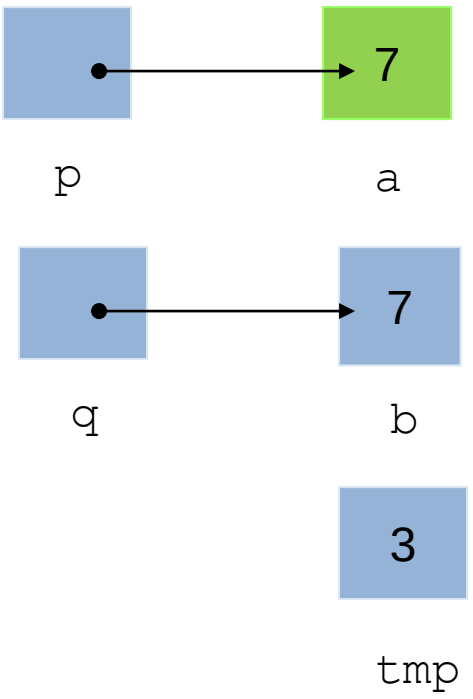
Example – Swapping Values

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

void swap(int *p, int *q) {
     int tmp;

     tmp = *p;        /* tmp = 3 */
     *p = *q;         /* *p = 7 */
     *q = tmp;        /* *q = 3 */
}

int main(void)
{
     int a = 3, b = 7;
     swap(&a, &b);
     cout << a <<" "<< b <<endl;
      /* 7  3 is printed */
     return 0;
}
```



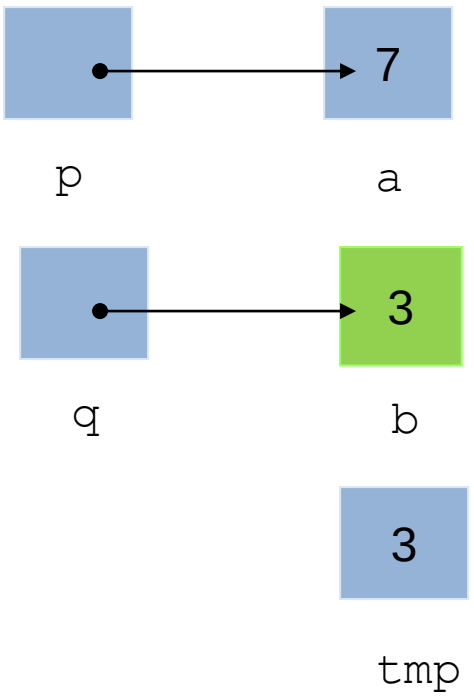
Example – Swapping Values

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

void swap(int *p, int *q) {
     int tmp;

     tmp = *p;        /* tmp = 3 */
     *p = *q;         /* *p = 7 */
     *q = tmp;        /* *q = 3 */
}

int main(void)
{
     int a = 3, b = 7;
     swap(&a, &b);
     cout << a <<" "<< b <<endl;
      /* 7  3 is printed */
     return 0;
}
```



Example – Swapping Values

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

void swap(int *p, int *q) {
     int tmp;

     tmp = *p;        /* tmp = 3 */
     *p = *q;         /* *p = 7 */
     *q = tmp;        /* *q = 3 */
}

int main(void)
{
     int a = 3, b = 7;
     swap(&a, &b);
     cout << a <<" "<< b <<endl;
      /* 7  3 is printed */
     return 0;
}
```



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;
```

Operations on Pointer Variables

- Copying the address:

```
p = q;
```

  - `p` and `q` point to the same variable.

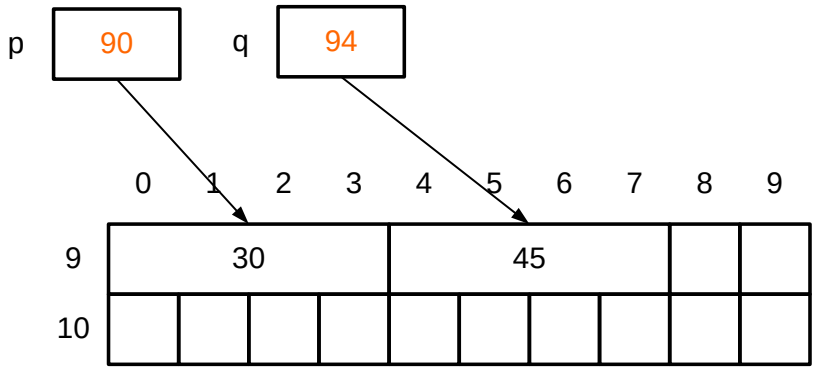
- Copying the content:
  - Copy the value of the variable which is pointed by the `q` to the variable which is pointed by `p`.

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

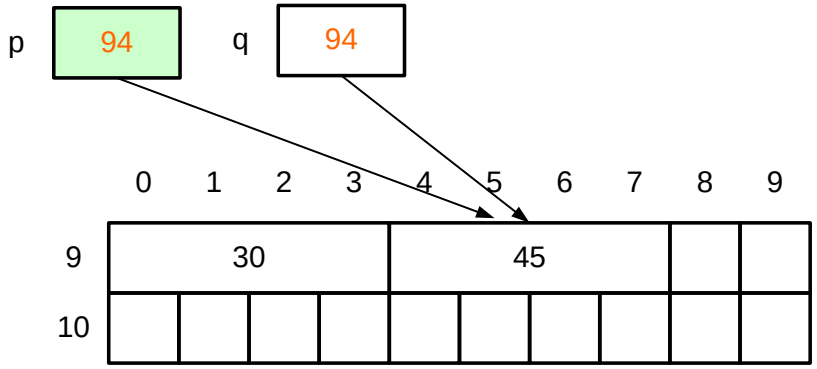
  - `p` and `q` may point to different variables.

Copy Address

Assignment: `p = q;`

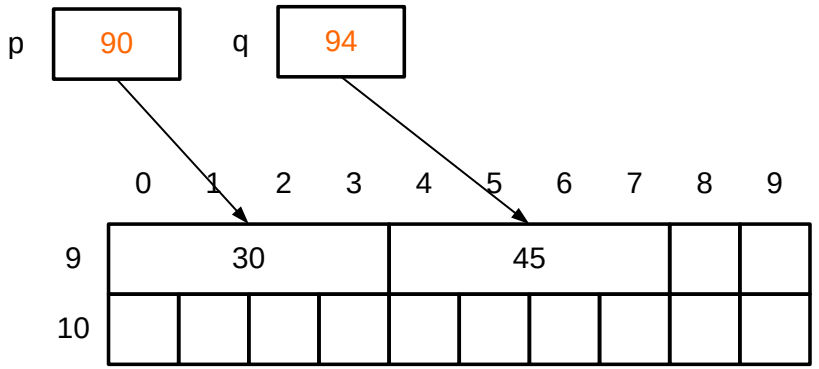


We copy the content (which is an address) of `q` to `p`.
After the assignment, `p` and `q` points to the same location in memory.
Therefore, if we change `*p`, `*q` will also be changed.

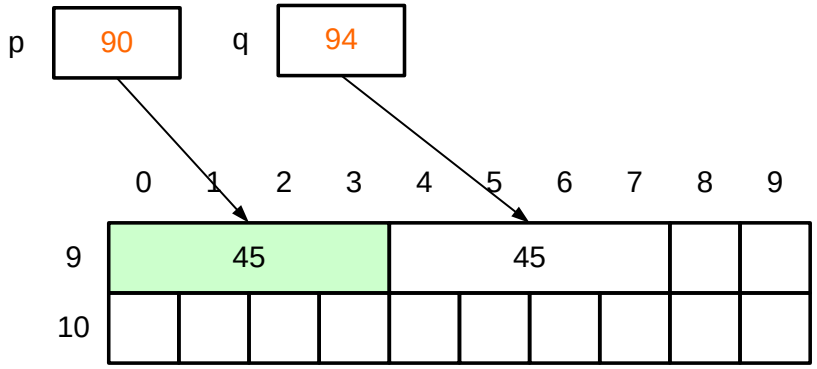


Copy Content

Different form: `*p = *q;`



We copy the value of variable pointed by `q` to the variable pointed by `p`. After the assignment, `p` and `q` 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 Array and Pointer

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;
```

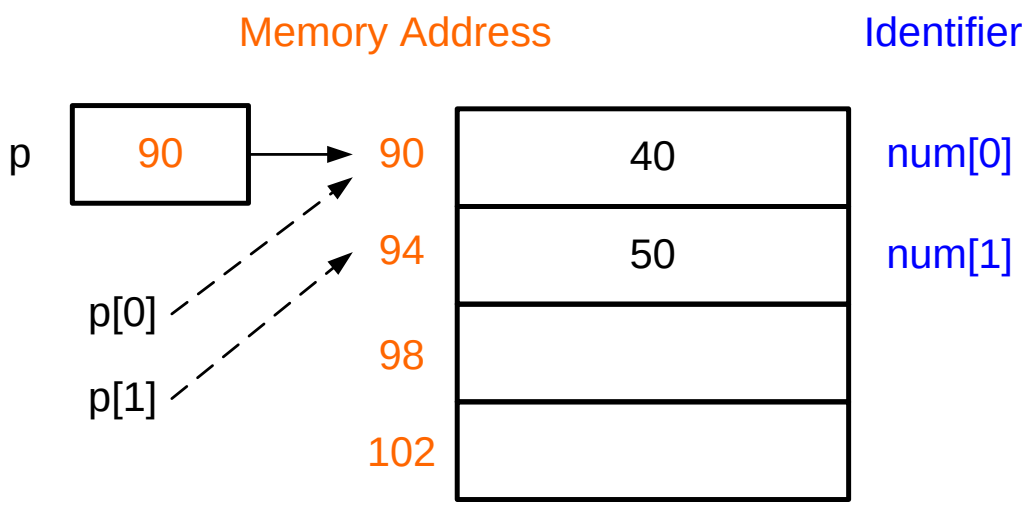
However, the value of num cannot be changed.

```
num=p; /*illegal*/
```

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

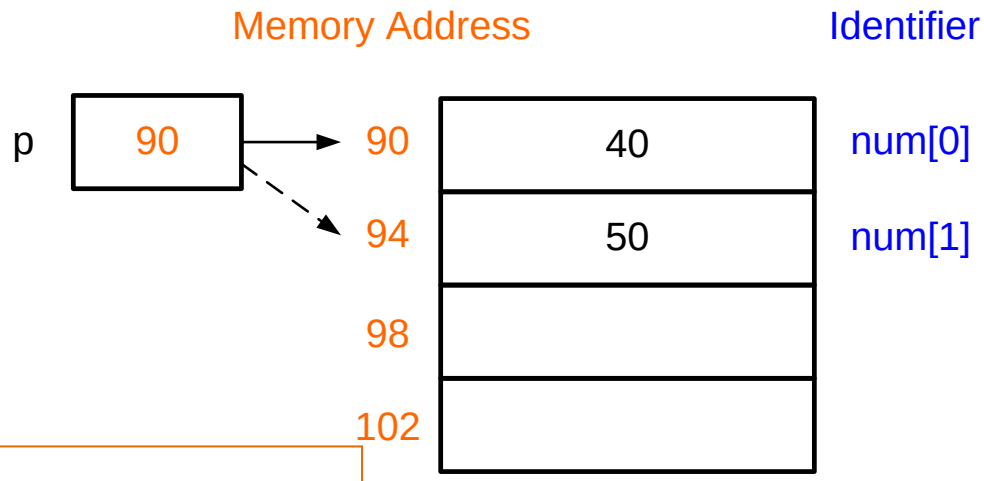
Equivalent to

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



Relationship between Array and Pointer

```
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;
```

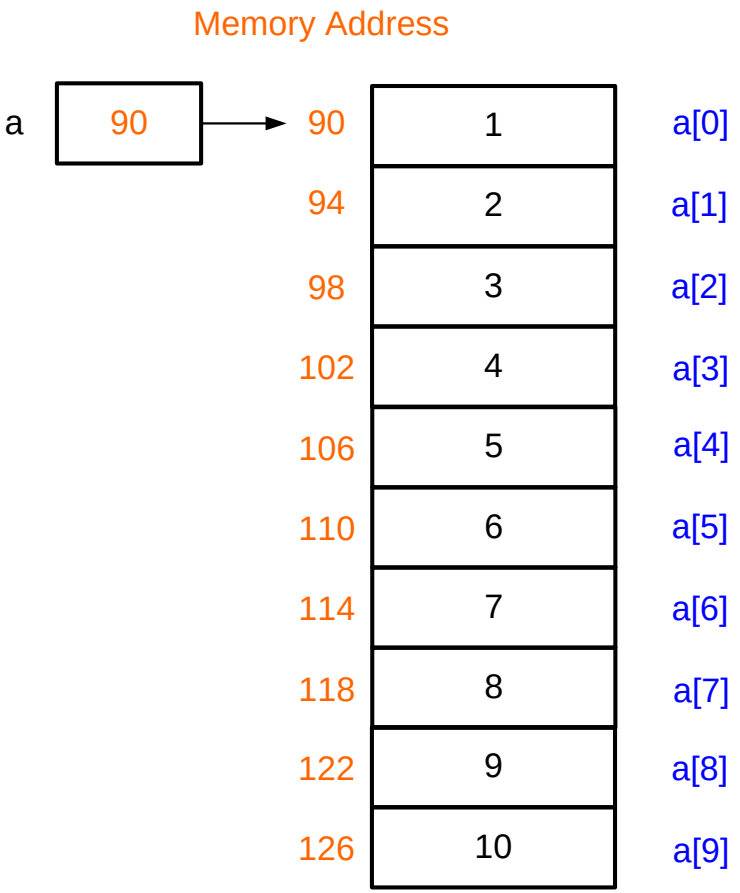
++p increments the content of p (an address) by sizeof(int) bytes

Array and Pointers

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

Pointer Arithmetic

```
#define N 10
void 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*/
}
```



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.

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

```
float sum(int *);
```
is the same as
```
float sum(int []);
```

- As part of the header of a function definition, the following declarations are equivalent

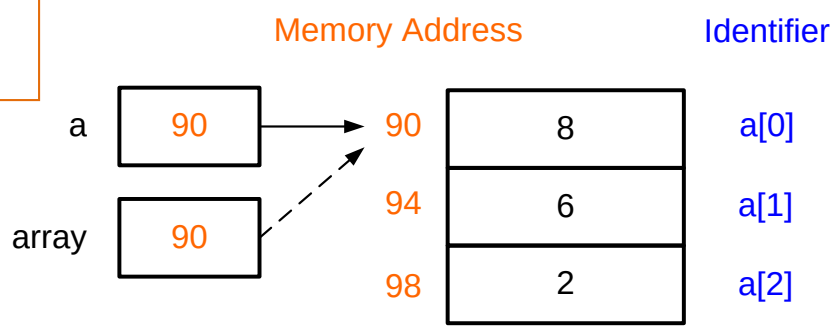
```
int array[]
int *array
```

Parameter Passing

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

```
float sum(int *array)
{
      int i;
      float 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]

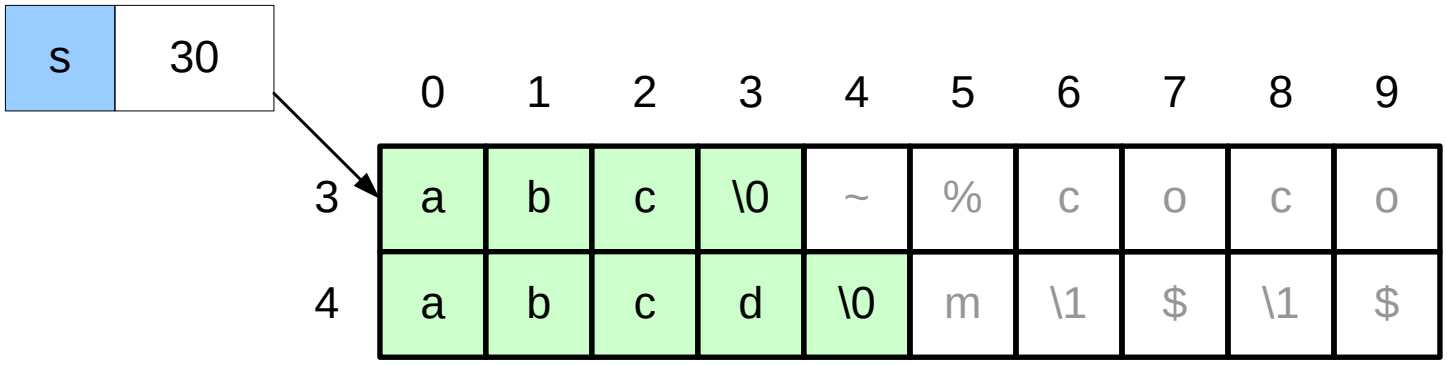


When `array` is passed as parameters, call-by-reference is used.
If we modify `array[i]` in `sum`, `a[i]` is also modified in main

Array, Pointer and String

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

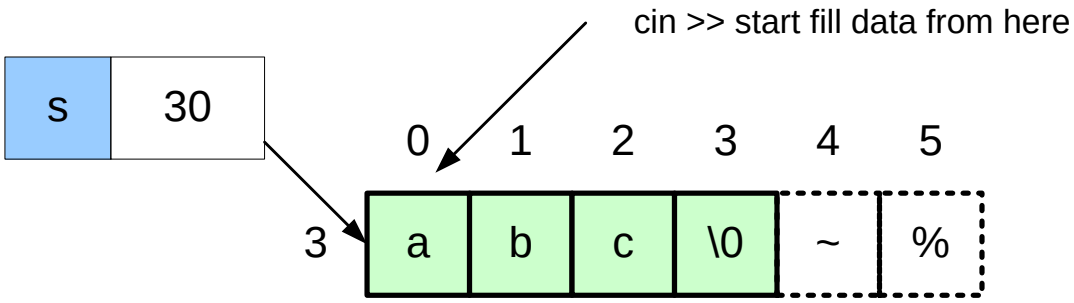
Illegal as `s` is a constant pointer and cannot be modified.



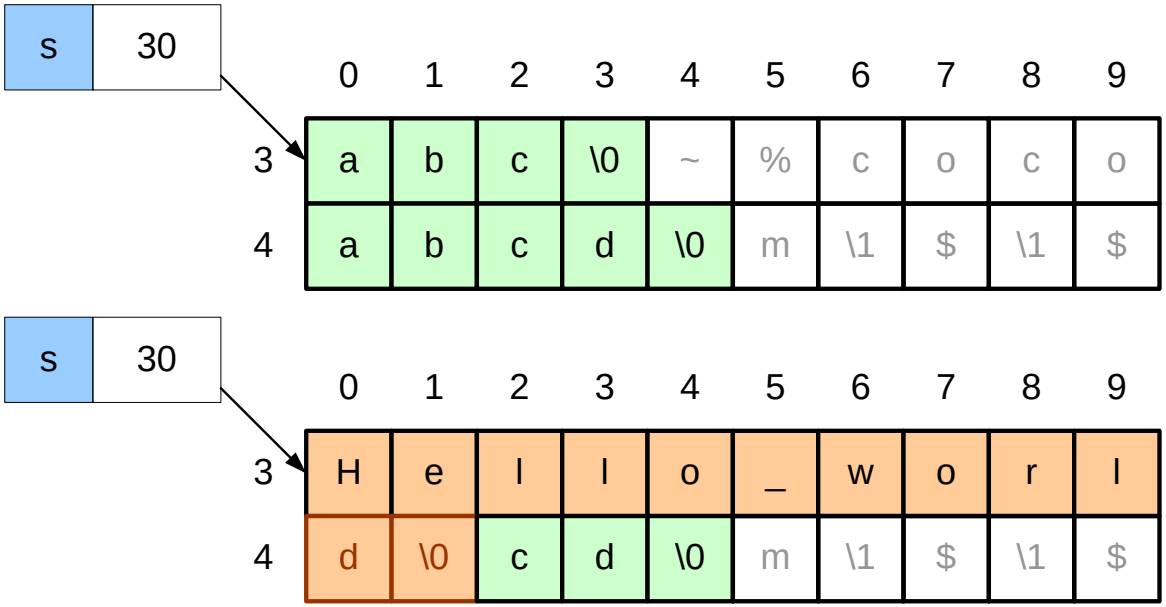
cin >> a string (I)

```
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;
void main () {
   char *s1;
   cin >> s1;
   cout << s1;
}
```

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 to by `s1`. This may generate errors as we may overwrite some important locations in memory.

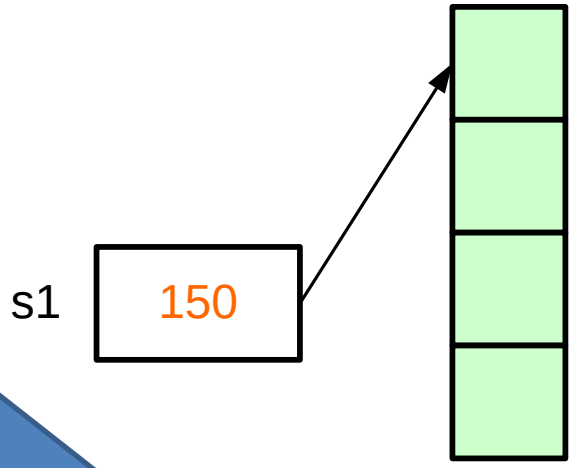
Dynamic Memory Allocation

- **new** to dynamically allocate memory for a variable or array (allow variable length).
  - `int len = 5;`
  - `int *ptr = new int[len];`
  - `int *ptr2 = new int;`
- Note you cannot use `int arry[len]` since len is an variable.
- **delete** to free memory space.
  - `delete [] ptr; // free memory space for whole array`
  - `delete ptr; // free only first element, should be used when only 1 element was new-ed, e.g. ptr2.`

new dynamically allocates a memory space of 5 int. new returns a pointer to the 1st int of the chunk of memory, which is assigned to ptr.

Dynamic Memory Allocation

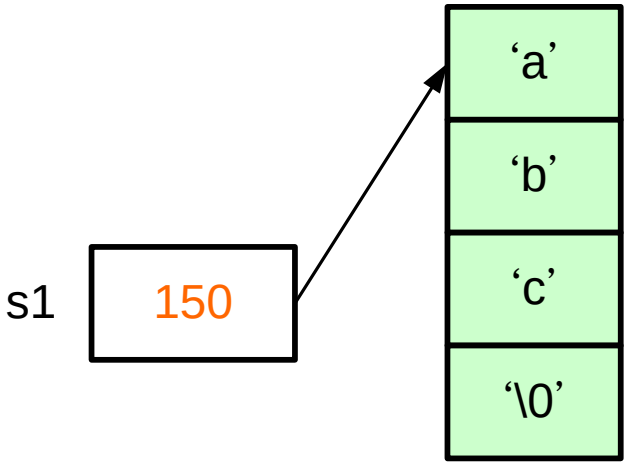
```
#include <iostream>
void main () {
    char *s1;
    s1=new char[4];
    cin >> s1; /*input "abc"*/
    cout << s1;
    delete [] s1;
    s1=new char[6];
    cin >> s1;
    cout << s1;
    delete [] s1;
    s1=NULL;
}
```



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

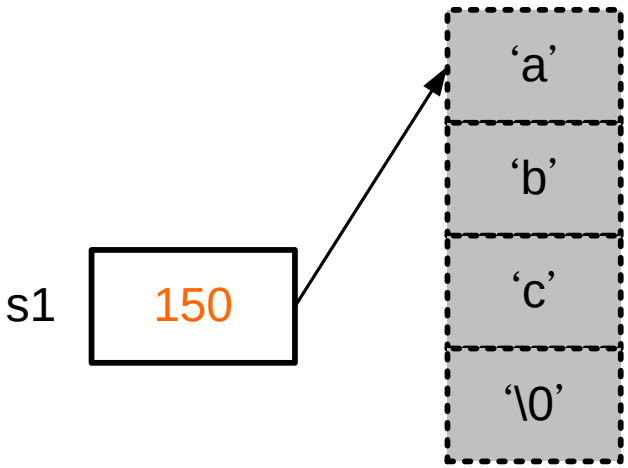
Dynamic Memory Allocation

```
#include <iostream>
void 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;

}
```



Dynamic Memory Allocation

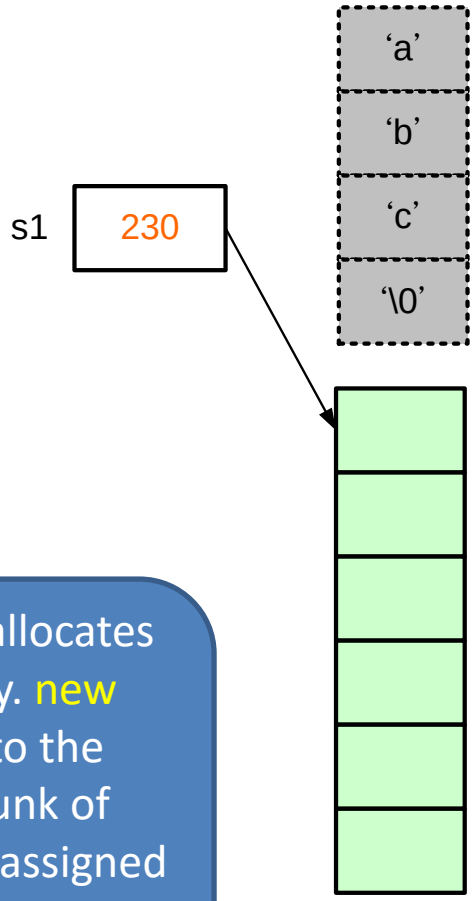
```
#include <iostream>
void main () {
   char *s1;
   s1=new char[4];
   cin >> s1; /*input "abc"*/
   cout << s1;
   delete [] s1;
   s1=new char[6];
   cin >> s1;
   cout << s1;
   delete [] s1;
   s1=NULL;

}
```



Memory is free and can be used to store other data

Dynamic Memory Allocation

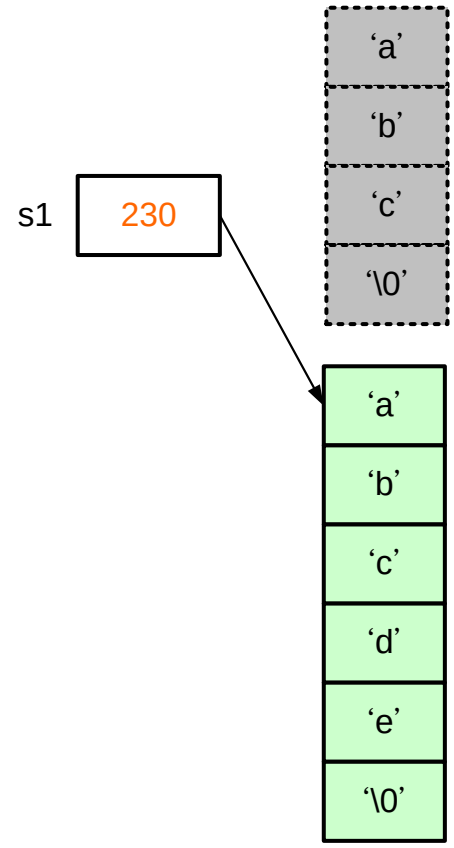
```
#include <iostream>
void main () {
    char *s1;
    s1=new char[4];
    cin >> s1; /*input "abc"*/
    cout << s1;
    delete [] s1;
    s1=new char[6];
    cin >> s1;
    cout << s1;
    delete [] s1;
    s1=NULL;

}
```



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 Allocation

```
#include <iostream>
void 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;

}
```



Guidelines on Using Pointer

- Initial a pointer to NULL after declaration
  - char *cPtr=NULL;
- Check its value before use
  - if (cptr!=NULL){
  - ….
  - }
- Set it NULL again after free
  - delete cPtr;
  - cPtr=NULL;
- Is it possible to verify a pointer is pointing to a valid / expected address??

Reference Type in C++

- Reference type in C++
  - datatype& identifier
  - `int a = 5;`
  - `int& b=a;`
- Reference defines an alias for an given variable
  - `int& c; // wrong, c has to be an alias for a existing variable.`
- Reference is used in the same way as the original variable.
  - `b = 10;`
  - `cout << a; // print 10`
  - `cout << b; // print 10`
- You cannot change the reference to another variable after initialization.
  - `int a=5, d=10;`
  - `int& b = a;`
  - `b = d; // this is assignment, not changing reference for b.`

Pointer and Reference Type

- Both can be used for call-by-reference
  - If function parameters are pointers or references
  - the variables pointed by / referenced by the parameters can be changed inside the function.
  - E.g. the variables passed from the caller can be changed in both.
    - `void f(int *pA, int *pB);`
    - `void f(int &a,  int &b);`
- You need to use dereferencing to access the variable pointed by a pointer.
  - `int *ptr = &a;`
  - `cout << *ptr;`
- You use reference variable exactly the same way as the original variable.
  - `int& b = a;`
  - `cout << b;`

Summary

- Pointer is a special variable used to store the memory address (location) of another variable.
- Pointer is typed, and its type is determined by the variable it is pointing to.
- * operator has two meaning
  - For declaration, e.g int *p1, char *pc;
  - For dereference, e.g. x=*p1, *pc='b';
- & operator return the address of any variable

Summary

- Call by reference: pass the address of a variable to a function such that its content can be updated within the function.
- For a function returns one value only, we can use a return statement.
- For a function returns more than one value, one must use call by reference.

Summary

- Pointer can be used to access array element.
- Array variable without subscribe is a pointer pointing to the first element of the array.
- String is stored as an array of character.
- cstring must be terminated by an ‘\0’ character, therefore a string with 5 characters will take up 6 characters space.
- Operator new allocates memory space to program 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 pointer, as it may point to an invalid / unexpected location and make the program crashed.