

CS2311 Computer Programming

Memory Pointers
and
Dynamic memory/array allocation

Memory and Variable

- Variable is used to store data that will be accessed by a program on execution.
- Variables are internally stored in *some position* in the main memory, but we *usually* don't care where exactly they're stored
- 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

Variable and Memory

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

	0	1	2	3	4	5	6	7	8	9
3	100				200				a	
4										
5										
6										
7										
8										

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. *(Note that an **int** variable occupies 4 bytes)*
- Address of a variable depends on the computer. To find out the address, you may use the debugger (like Visual Studio). The address is usually expressed in hexadecimal *(base 16 with values 0-9 and A-F e.g. 0x00023AF0, 0x23456780)*

Pointer

- In C++, a pointer is a variable which designs to store the **address** of another variable. When a pointer storing the address of a variable, we said the pointer is *pointing* to the variable.
- Like normal variables, pointers are **typed**. As such, an integer pointer (**int ***) cannot be used to store the address of a **char**

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

Declaring pointers

- To declare a pointer variable, place a "*" sign before its name:

```
char * cPtr;    //a character pointer
int * nPtr;     //a integer pointer
float * fp;     //a floating point pointer
```

- There's some flexibility on the position of *, writing:

```
char * cPtr;    is OK
char*  cPtr;    is also OK
char  *cPtr;    is also OK
```

- But be careful if you join statements together:

```
char * a,b,c;
```

will define ONLY **a** as pointer, meanwhile **b** and **c** are normal characters

- To create 3 character pointers:

```
char *a, *b, *c;
```

Assigning positions to pointers

- Since pointers are typed, you **cannot** directly assign an address:

```
int *ptr;  
ptr = 0x23456780; //WRONG! Cannot take number
```

- You can only get an address by 1) referring to another variable:

```
int x;  
int *ptr;  
ptr = &x; //In C++ & here means "the address of"
```

- Or... 2) copying from another pointer:

```
int *ptr2;  
ptr2 = ptr; // both ptr, ptr2 stores addr of x
```

- Or... 3) doing arithmetic with other pointers:

```
ptr2 = ptr+10;
```

Reading content via pointers

- Operator `&` can get the address from a variable:

```
int x=100;  
int *ptr;  
ptr = &x;
```



- We can actually do the reverse:
get the value of a variable from an address !

- To do so, we need the `*` operator:

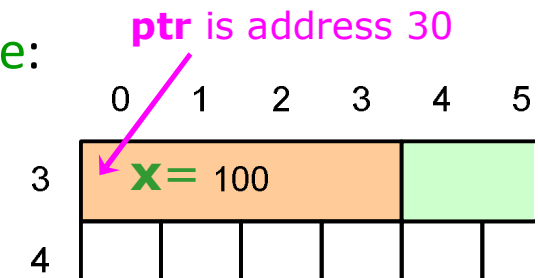
```
int y;  
y = *ptr;
```



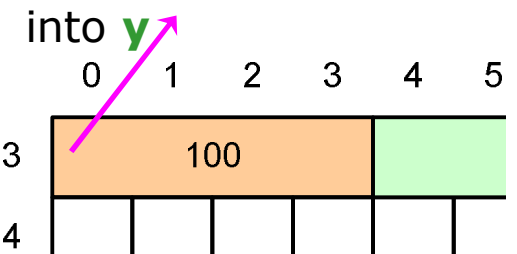
`ptr` is pointing to `x`, which is storing 100.

So when we assign the *content-being-pointed by* `ptr` to `y`,

`y` is also 100 (*i.e. same as x*).



Copy what we have in
address 30 (i.e. 100)

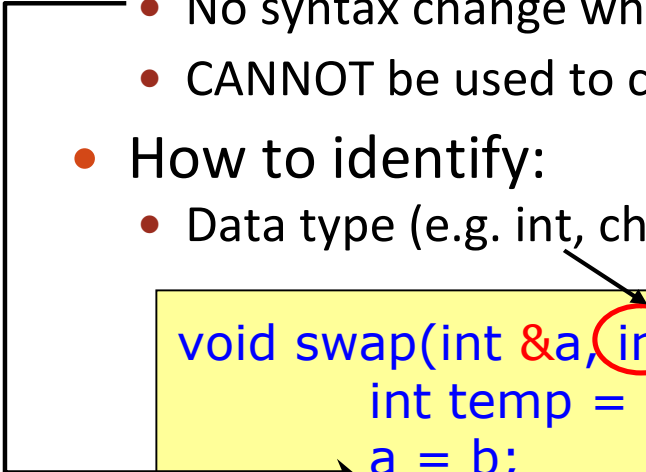


The confusing punctuations...

- Pointers are "famous" (*notorious!*) of being difficult as both the * and & operators are overloaded (*abused?*) for TWO TOTALLY DIFFERENT usages.
(*why don't they use different symbols ?!*)
- To save time, let's remember and *distinguish* the syntaxes and usages in the coming few pages before proceeding forward!

IMPORTANT: Usage 1 of &

- Usage 1 of the & operator is for **pass-by-reference** (Lecture 6)
- i.e. Allow a **function** to change the original parameter
- Special points:
 - Only used for function / parameter passing
 - NOT related to pointers and addresses
 - No syntax change when reading variables. (*still use **a**, don't write *a*)
 - CANNOT be used to create variable (*e.g. `int &a=5;` is WRONG!*)
- How to identify:
 - Data type (e.g. `int`, `char`) MUST appear before &



```
void swap(int &a, int &b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

IMPORTANT: Usage 2 of &

- Usage 2 of the & operator is to **get the address of a variable (New!)**
- The address is then stored in another pointer, or passed to functions which takes pointer arguments
- Special points:
 - Can be used *with or without* functions
 - NOT used for declaration
 - NOT creating new variable
- How to identify:
 - NO data type (e.g. int, char) before &

```
int x;  
int *ptr;  
ptr = &x;
```

It's WRONG to write `ptr = int&x;`

IMPORTANT: Usage 1 of *

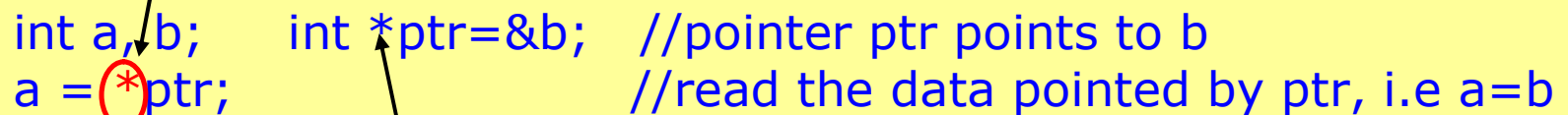
- Usage 1 of the * operator is to DECLARE /create pointer variable
- Special points:
 - Once the pointer is created, it is *usually* being referenced without *
(unless for Usage 2 mentioned in the next page)
e.g. Get Address: `ptr = &x;` (don't write `*ptr = &x`)
e.g. Copy Address: `ptr2 = ptr` (don't write `*ptr2 = ptr`)
e.g. Pass to function as param: `copyString(ptr1,ptr2);` (not `*ptr1`)
 - Applied per variable (`int *x, y;` ONLY `x` is pointer, `y` is integer)
 - Also used in function declaration (if the argument is pointer type)
In that case, * is also needed
(e.g. `void copyString ( char *dest, char *src )`)
- How to identify:
 - Data type (e.g. int, char) MUST appear before *

```
int*ptr;  
ptr = &x;           //or merged as int*ptr=&x;
```

IMPORTANT: Usage 2 of *

- Usage 2 of the * operator is **access the content pointed by pointer**
- The operator * here means **"the value pointed by"**
(e.g. `a=*ptr;` means `a = the value pointed by ptr`)
- This read/write operation is also known as "de-referencing"
- Special points:
 - NOT used for declaration and NOT creating new pointer / variable
 - Could be used together with parenthesis (e.g. `a = *(ptr);`)
 - The address after * can be a pointer or some calculated address
(e.g. `a = *(ptr+1);` will read the *next* integer *following* the address in `ptr`)
- How to identify:
 - NO Data type (e.g. `int`, `char`) before *

```
int a,b;    int *ptr=&b;    //pointer ptr points to b
a = (*ptr); //read the data pointed by ptr, i.e a=b
```



This is usage 1 (declaration), you can see `int` before *

Example

```
int x,y;                                //x and y are integer variables
void main() {
    int *p1,*p2;                        // Usage 1 of *: declare/create pointers
    x=10;
    y=12;
    p1=&x;                              // Usage 2 of &: p1 stores the address of x
    p2=&y;                              // Usage 2 of &: p2 stores the address of y

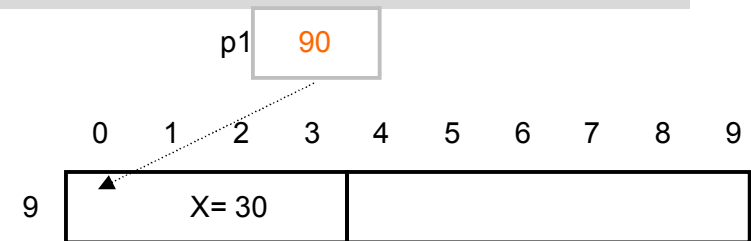
    *p1=5;                             // Usage 2 of *: change memory value
                                        // As p1 points to x, so x becomes 5

    *p2=*p1+10;                        // Usage 2 of *: change memory value
                                        // *p1 is "content pointed by p1" (i.e. x (=5) )
                                        // p2 points to y, so *p2= writes result into y
}
```

Common operations

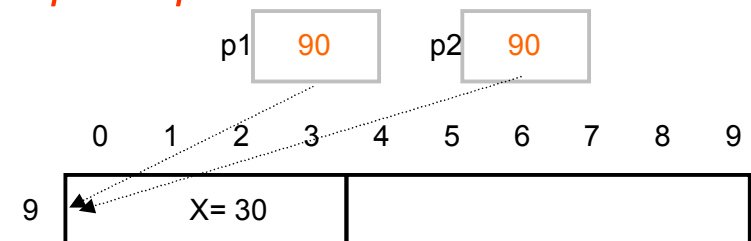
- Let pointer *p1* points to the variable *x*

- p1*=&*x*;



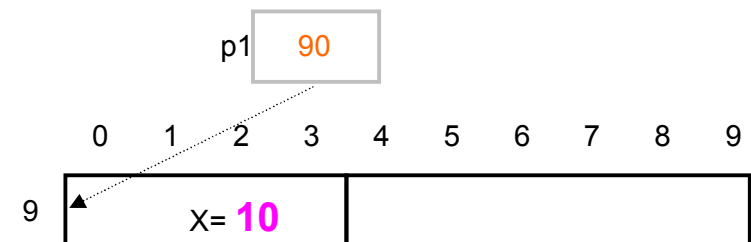
- Copy address (not memory content!) from *p1* to *p2*

- p2*=*p1*;



- Update the value (*memory content*) of variable pointed by a pointer

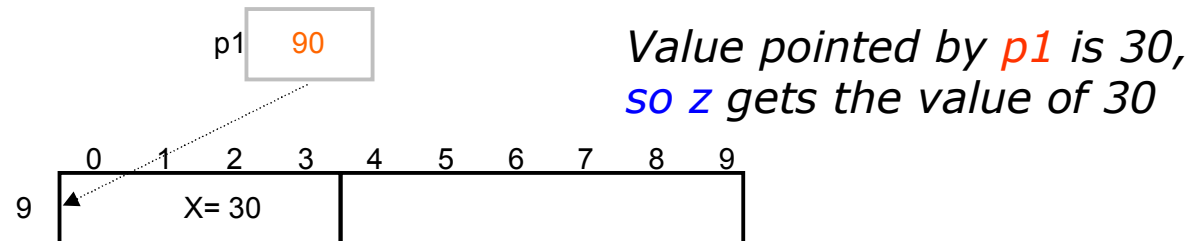
- **p1*=10;



Common operations

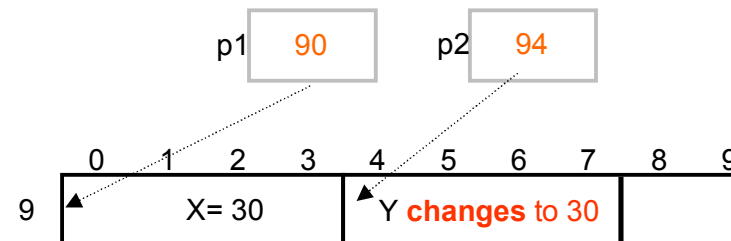
- Retrieve the value (*memory content*) of variable pointed by a pointer

- `int z=*p1;`



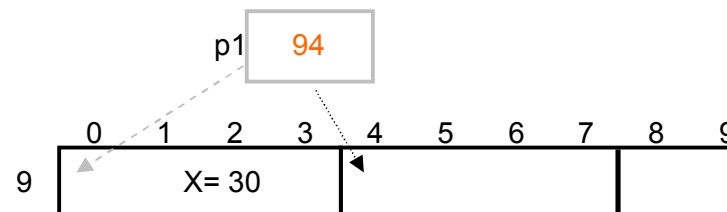
- Copy *memory pointed by p2* to *memory pointed by p1*

- `*p2=*p1;`



- Move a pointer to some neighboring position

- `p1++;` or `p1=p1+1;` will let *p1* points to the *next* integer



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:

40

Applications of pointer

- Fast Array/String Access
 - e.g. change the 5th character in a string
- Dynamic Memory Allocation
 - Request additional memory space for storing data.
 - The size of memory space is not necessarily **const**
(e.g. can depends on user input)
- Parameter passing – pass-by-pointer
 - Allows function to change the original variable
 - As an alternative to *pass-by-reference* (In C language, there's no *pass-by-reference*, so people used *pass-by-pointer*)

Parameter passing – *pass by pointer*

- Allows function to change the original variable
(the default – pass-by-value cannot modify the original)
- As an **alternative** to *pass-by-reference* *(In C language, there's no pass-by-reference, so people used pass-by-pointer)*
- Useful if function has >1 return value *(e.g. input from keyboard)*
- The caller pass the **address** *(not value!)* of a variable to a function. The *address* can be a pointer variable or being derived using the **&** operator *(Usage 2 of &)*
- The corresponding parameter is of “pointer type”, so in function declaration inside the parenthesis, we write **Func(int *param)** *(Usage 1 of *)* rather than **Func(int param)**

Revision:

Pass by value – function cannot modify the original

```
void f (char c1_in_f) {  
    c1_in_f='B';  
}  
  
void main() {  
    char c1_in_main='A';  
    f(c1_in_main);  
    cout << c1_in_main; //print 'A'  
}
```

In call by value, there is no way to modify c1_in_main inside f(), unless we use the return statement.

Revision:

Pass by reference – function can modify the original

```
void f (char &c1_in_f) { //Usage 1 of &: pass-by-ref
    c1_in_f='B';
}

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

③ Write & ONLY in the definition

② Just write c1_in_f, no &, no *

① Just write c1_in_main, no &

- c1_in_main and c1_in_f are actually the SAME variable
- Hence the update of c1_in_f affects c1_in_main

New concept:

Pass by pointer – function can modify the original

```
void f (char *c1_ptr) {  
    *c1_ptr='B';  
}  
  
void main() {  
    char c1_in_main='A';  
    f(&c1_in_main);  
    cout << c1_in_main;  
}          //print 'B'
```

Parameter is pointer type, so we need ***** (Usage 1 of *****). It points to the 'A' in main

Overwrite the 'A' (in main) with 'B'.
To write memory, we use ***** (Usage 2 of *****)

Not passing 'A', but rather we pass the address. (Usage 2 of **&**)

First, the address of **c1_in_main** is retrieved and passed to function

When we enter f(), the address in parameter is copied to **c1_ptr**

In other words, **c1_ptr** now points to the 'A' in main

Pass by pointer

```
void f (char *c1_ptr) {  
    *c1_ptr='B';  
}  
  
void main() {  
    char c1_in_main='A';  
    f(&c1_in_main);  
    cout << c1_in_main; //print 'B'  
}
```

↓ points to

In *normal* situation, the function by default has no access to the name **c1_in_main** because of the scope rules. But since we're modifying the content *INDIRECTLY* via *pointer* (*We did not use the name **c1_in_main** in function at all!*) We can easily modify the value of the original variable.

Note that we should not write **c1_ptr**='B'; But it should be ***c1_ptr**='B';
As we're trying the modify MEMORY but not the pointer (*address*) itself

Note the different meaning of *

The type of c1_ptr is char* (*Usage 1 – page 12*)

dereference a
pointer
(*Usage 2 – page 13*)

```
void f (char *c1_ptr) {  
    *c1_ptr='B';  
}  
  
void main() {  
    char c1_in_main='A';  
    f(&c1_in_main);  
    cout << c1_in_main; //print 'B'  
}
```

Example – Read two values from keyboard

Suppose we'd like to read the **width** and **height** from keyboard

Obviously **pass-by-value** does not work as the variables in main are not affected. Also **return** cannot return two values...

```
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;
}
```



Not Working!

Example – Read two values from keyboard

Solution 1: pass-by-reference (Lecture 6)

Use **&** for pass-by-ref
(Usage 1 of **&** - page 10)

```
void getDimension(float &width, float &height){  
    cout << "Please enter the width of rectangle:";  
    cin >> width;  
    cout << "Please enter the height of rectangle:";  
    cin >> height;  
}
```

Just write **width** and **height** as
the old version

```
void main(){  
    float w, h;  
    getDimension(w,h);  
    cout << "area=" << w*h << endl;  
}
```

Just write **w** and **h** as the old version

Example – Read two values from keyboard

Solution 2: pass-by-pointer

Use ***** as the parameters are address (Usage 1 of *****)

```
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;
}
```

Write ***width** and ***height** since we'd like to update Memory but not updating the Address (pointer) (Usage 2 of *****)

Write **&w** and **&h** as we're passing Address of variables (Usage 2 of **&** - page 11)

Pass-by-Value

```
int add(int m, int n)
{
    int c;
    c = m+n;
    m = 10; //not affects original
    return c;
}

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

Pass value as parameter.

Pass-by-Pointer

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

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

Pass address as parameter.

Pass by Pointer - Guideline

- Add the ***** sign to the parameters which use pass-by-pointer

- e.g.

```
void cal(int *x, int y){  
    //x is pass by pointer  
    //y is pass by value  
    .....  
    ..... *x;  
}
```

- Add the **&** sign to the *normal* variable to extract its address
(*don't need & if the variable is a pointer itself*)

- e.g.

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

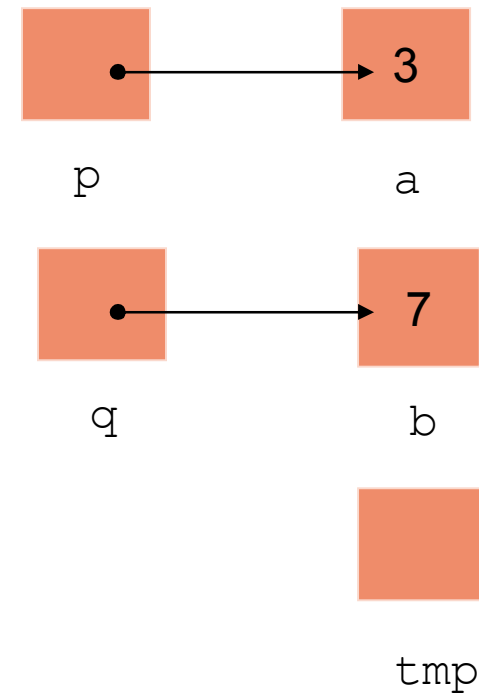
Example: swapping values

```
#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 */
}

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

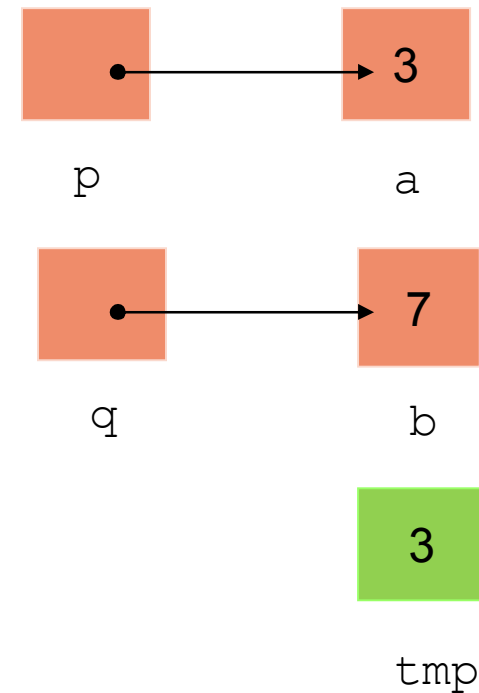


Example: swapping values

```
#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 */
}

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



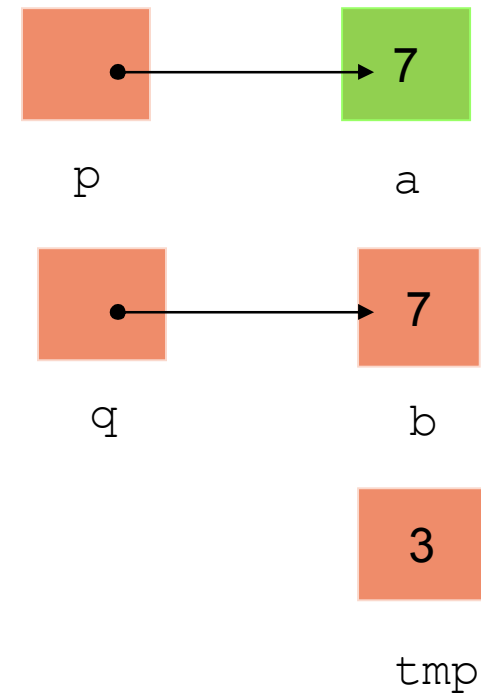
Example: swapping values

```
#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 */
}

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



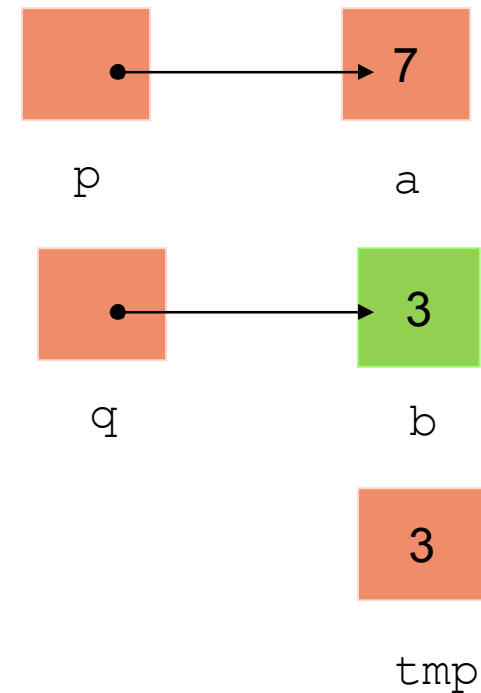
Example: swapping values

```
#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 */
}

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



Relationship between array and pointer

We can use array-like notation in pointers

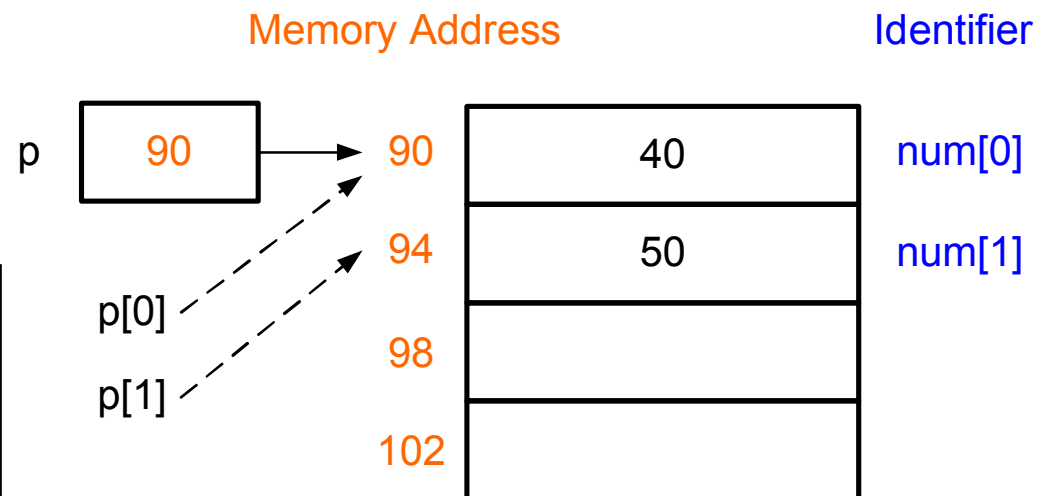
Instead of writing `*p`, we can also write as `p[0]`;

Instead of writing `*(p+1)`, which access the *next* item, we can just write `p[1]`;

```
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;
```



The name of the array (e.g. `num`) can be considered as an address/pointer
So writing `p=num`; will let the pointer points to the first byte of the array.

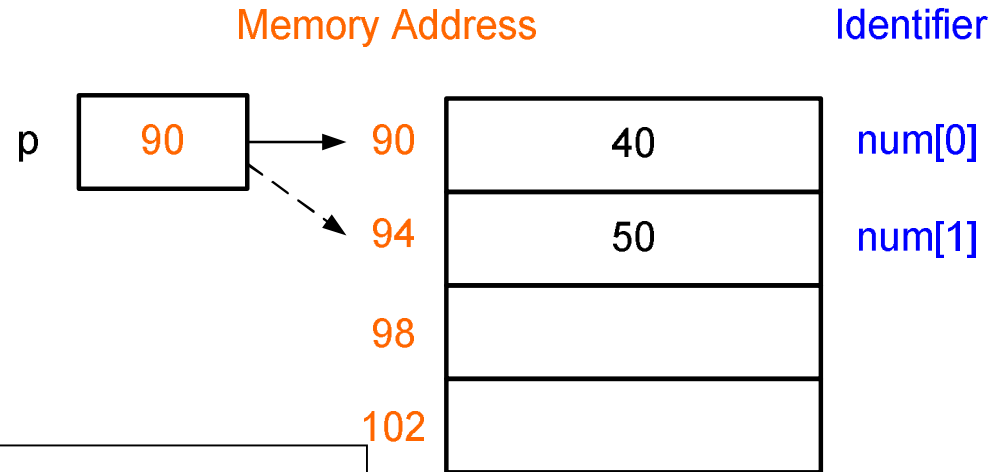
Note that `num` is not a real pointer and could not be redefined, so writing `num=&x`, or writing `num++` are illegal.

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 shifts the address **p** by `sizeof(int)` bytes (NOT 1 byte !)
Similarly **p=p+1**; also shifts `sizeof(int)` bytes

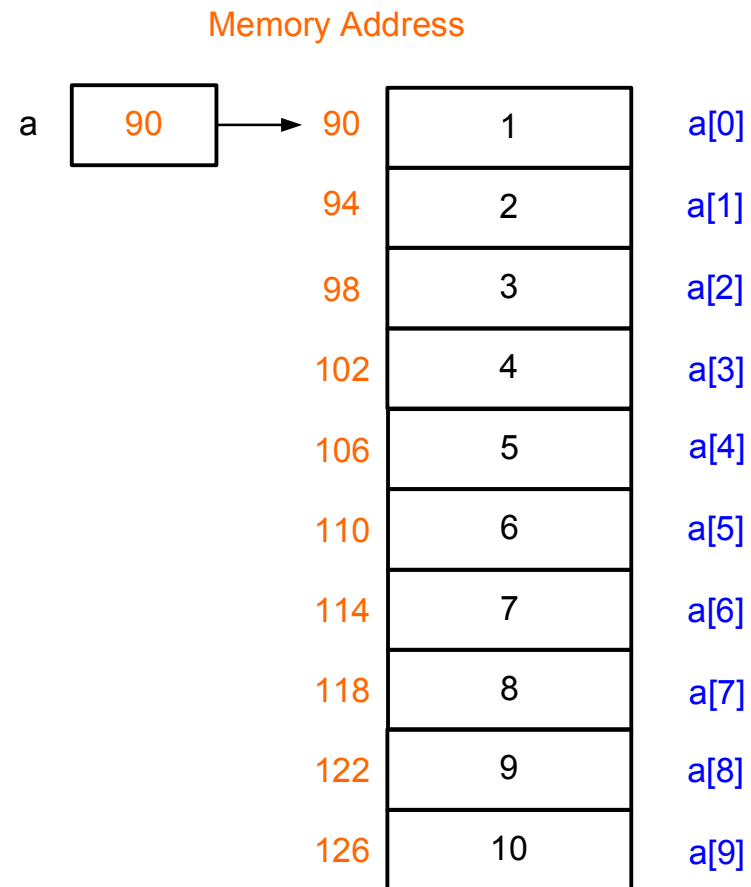
Relationship between array and pointer

Equivalent representation		Remark
<code>num</code>	<code>&num[0]</code>	<code>num</code> is the address of the 0th element of the array
<code>num+i</code>	<code>&(num[i])</code>	Address of the <i>i</i> th element of the array
<code>*ptr</code>	<code>ptr[0]</code>	The value of the 0th element of the array
<code>*(ptr+i)</code>	<code>ptr[i]</code>	The value of the <i>i</i> th element of the array
<code>(*ptr)+i</code>	<code>ptr[0]+i</code>	The value of the 0th element of the array plus <i>i</i>

Example : summing an array

```
const int 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. *(Because of performance consideration)*
- In other words, Array parameter passed to function is ALWAYS **pass-by-reference**. Any change in the function will be reflected to the caller.
- As a notational convenience, in function parameter declaration, the array bracket notation and the pointer notation are interchangeable

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

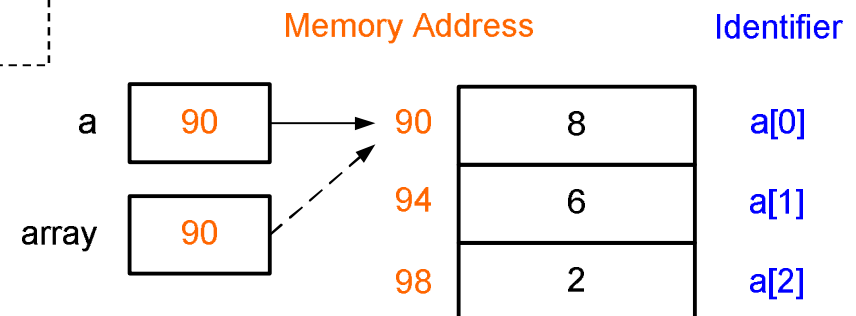
Example : Parameter Passing

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

```
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 address 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 automatically used. If we modify `array[i]` in `sum`, `a[i]` is also modified in `main`

Example – Copying strings (Array version)

- Assume the receiving string (array) has enough space
- Array version:

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

void CopyStr(char d[], char s[]) {
    int i;
    for (i=0 ; s[i]!='\0' ; i++)
        d[i] = s[i];
    d[i]='\0'; //IMPORTANT: End-of-String for Dest
}

void main() {
    char Source[] = "Hello World!";
    char Dest[30];
    CopyStr(Dest,Source);
    cout<<"The String is: "<<Dest<<endl;
}
```

Example – Copying strings (Pointer version)

- Assume the receiving string (array) has enough space
- Pointer version (For-loop):

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

void CopyStr(char *d, char *s) {
    for (____; *s!='\0' ; s++, d++)
        *d = *s;
    *d='\0'; //IMPORTANT: End-of-String for Dest
}

void main() {
    char Source[] = "Hello World!";
    char Dest[30];
    CopyStr(Dest, Source);
    cout<<"The String is: "<<Dest<<endl;
}
```

No initialization needed

2 adjustment operations.
s and d are local var in function, not affecting the caller

Example – Copying strings (Pointer version)

- Assume the receiving string (array) has enough space
- Pointer version (Shortest version with while loop):

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

void CopyStr(char *d, char *s) {
    while (*s) *(d++) = *(s++);
    *d = '\\0';    //IMPORTANT: End-of-String for Dest
}

void main() {
    char Source[] = "Hello World!";
    char Dest[30];
    CopyStr(Dest, Source);
    cout<<"The String is: "<<Dest<<endl;
}
```

As C++ treats 0 as false, so while (*s) is the same as while (*s!=0)

Have you tried this?

- Read in N numbers and report the total

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

void main() {
    int i,N,Total;
    cout<<"Input the count of numbers: ";
    cin>>N;

    int num[N];

    for (i=0;i<N;i++) cin>>num[i];

    Total=num[0];
    cout<<num[0];
    for (i=1;i<N;i++) {
        cout<<'+'<<num[i];
        Total+=num[i];
    }
    cout<<"="<<Total<<endl;
}
```

Not working in Visual Studio
as the size N is not constant !!

Dynamic Memory Allocation

- Instead of using *static* Array, we can also perform *dynamic* memory allocation (i.e. request memory *on-demand* during execution)
- Syntax: `pointer = new Type[size];`
- Example: `int *x= new int[50];`
will create a memory block (*i.e. array*) of 50 integers in memory and put the starting address in `x`. To access data, simply uses `x[i]` or `*(x+i)` as described earlier.
- The advantage is that the array size could be variable.
- IMPORTANT: When the data is no longer needed, write: `delete [] x;` to free up the memory (*Note the position of [], it's NOT delete x[]*)

The correct version

- Read in N numbers and report the total

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

void main() {
    int i,N,Total;
    cout<<"Input the count of numbers: ";
    cin>>N;

    int* num = new int[N];

    for (i=0;i<N;i++) cin>>num[i];

    Total=num[0];
    cout<<num[0];
    for (i=1;i<N;i++) {
        cout<<'+'<<num[i];
        Total+=num[i];
    }
    cout<<"="<<Total<<endl;
    delete [] num;
}
```

NULL pointer

- A special value (*actually it's 0*) which can be assigned to any type of pointer variable
- The name NULL is defined in several standard library headers, e.g. `<iostream>`
- When assigned to a pointer variable, it means the pointer points to nothing

- Example

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

Guidelines on using Pointer

- Initialize a pointer to **NULL** before you assign anything to it

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

- Check its value before use

```
if (cptr!=NULL){  
    .... x = cptr[i]; ...  
}
```

- Set it **NULL** again after freeing the memory

```
delete [] cPtr;  
cPtr=NULL;
```

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 points to.
- *** operator has two meaning
 - For declaration, e.g. `int *p1, char *pc;`
 - For dereference (read/write memory), e.g. `x=*p1, *pc='b';`
- *&* operator return the address of any variable
- Pass by pointer: pass the *address* of a variable to a function such that its content can be updated within the function.
- Sample use: function returns more than one value (e.g. swap)

Summary

- Pointer can be used to access array element.
- The name of an array (*without index*) is the address pointing to the first element of the array.
- Operator `new` allocates memory space to program and returns a pointer pointing to the beginning of the newly allocated space.
- Memory obtained by `new` must be `delete[]` after use.
- Extra care must be taken when handling pointer, as it may point to an invalid / unexpected location and make the program crashed.