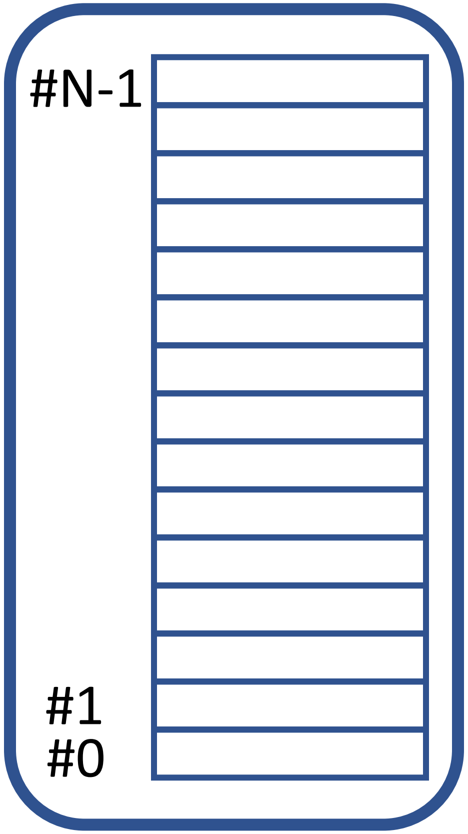
Pointers

Mohammed Bahja
School of Computer Science
University of Birmingham

Programmer’s View: Memory as an addressable storage

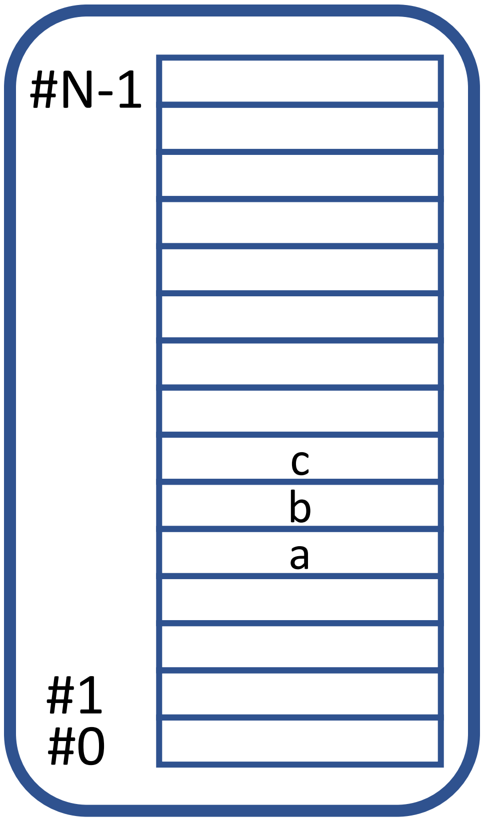


Memory

- Memory consists of small ‘locations’
- Each location can store a small information

From C program to Memory

```c
int a=4, b=5, c;
c = a*b;
```

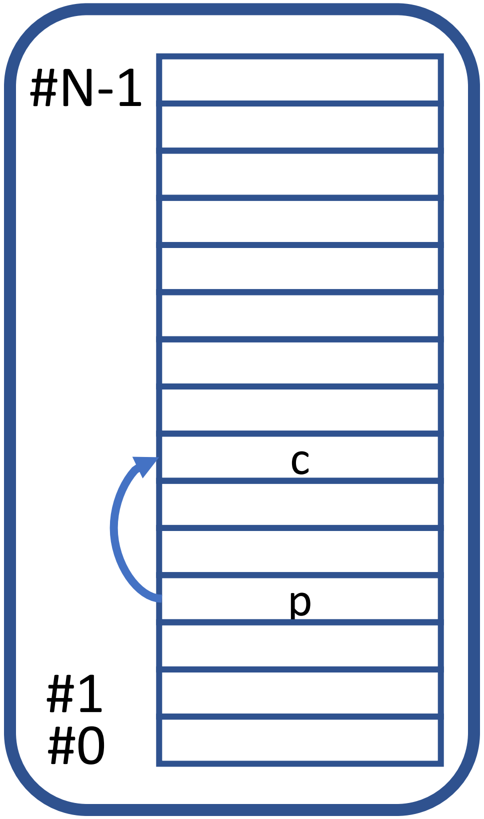


- Compiler allocates memory for the variables.
- Each variable has a unique address.

Pointer

Definition: A pointer is a variable that contains the **address** of a variable.

If ‘p’ is a pointer to a variable ‘c’, then the situation will be like this.

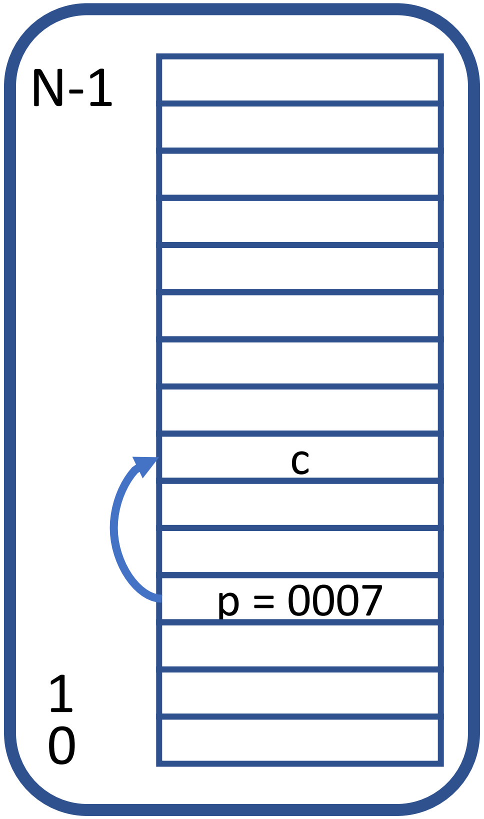


A pointer variable is also stored in the memory.

Pointer

Definition: A pointer is a variable that contains the **address** of a variable.

If ‘p’ is a pointer to a variable ‘c’, then the situation will be like this.



Example:
If ‘c’ is present in the memory location with address, say 0007, then the value of p will be 0007.

A pointer variable is also stored in the memory.

Unary operator &

The unary operator ‘&’ is the ‘**address-of**’ operator.
It gives the address of an object.
So,

```
p = &c;
```

assigns the address of c to p, and p is said to "point to" c.

Unary operator *

The unary operator '*' is called **indirection** or **dereferencing** operator. It is applied to a pointer to accesses the object the pointer points to.

Example:

If p is a pointer to an integer object, say c=5, then

```
*p
```

will give the value of c, i.e. 5

So,

```
p = &c;
c = *p;
```

Declaration of a Pointer variable

- A pointer variable is declared as follows.

```
T *p;
```

where `T` is a placeholder for an appropriate data-type.

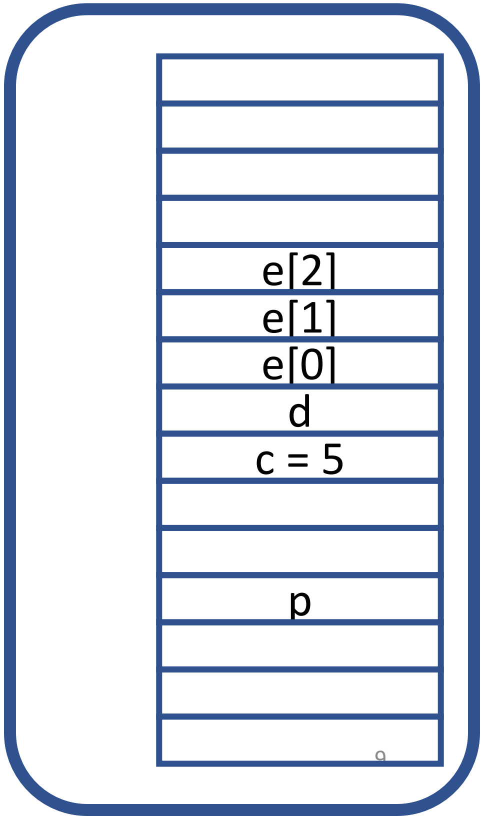
- p can point to a variable of type `T`.

```
Example:

int *p;    // p points to int variable
```

This means, the expression '`*p`' is an `int`

Example: Use of pointers

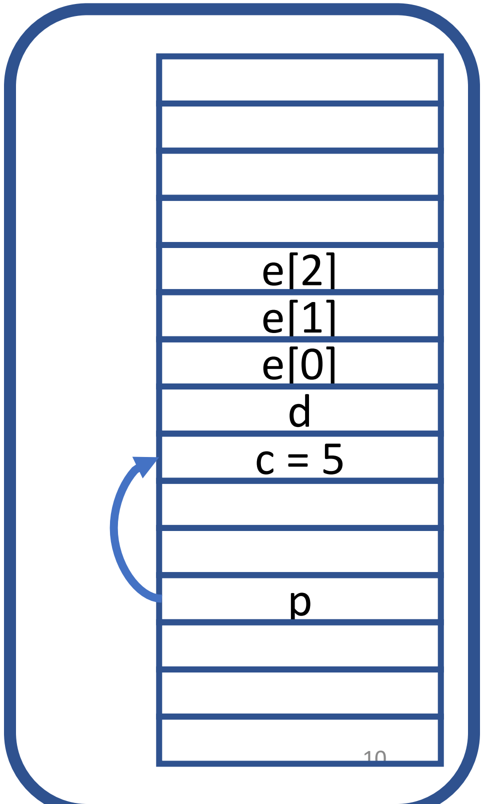
```
int c = 5, d, e[3];
int *p;        // Declared pointer p of type int

p = &c;        // p now points to c
d = *p;        // d is now 5
p = &e[0];     // p now points to e[0]
```



Example: Use of pointers

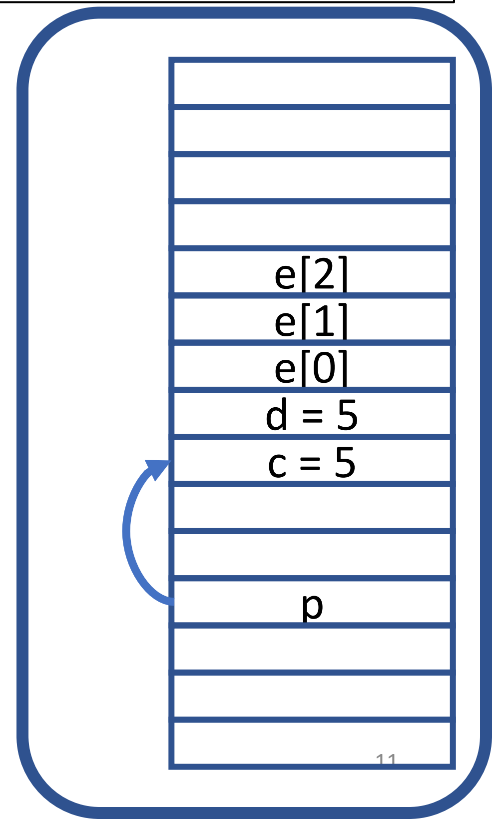
```
int c = 5, d, e[3];
int *p;        // Declared pointer p of type int

p = &c;        // p now points to c
d = *p;        // d is now 5
p = &e[0];     // p now points to e[0]
```

p contains the address of the memory location where c is residing.



Example: Use of pointers

```
int c = 5, d, e[3];
int *p;        // Declared pointer p of type int

p = &c;        // p now points to c
d = *p;        // d is now 5
p = &e[0];     // p now points to e[0]
```

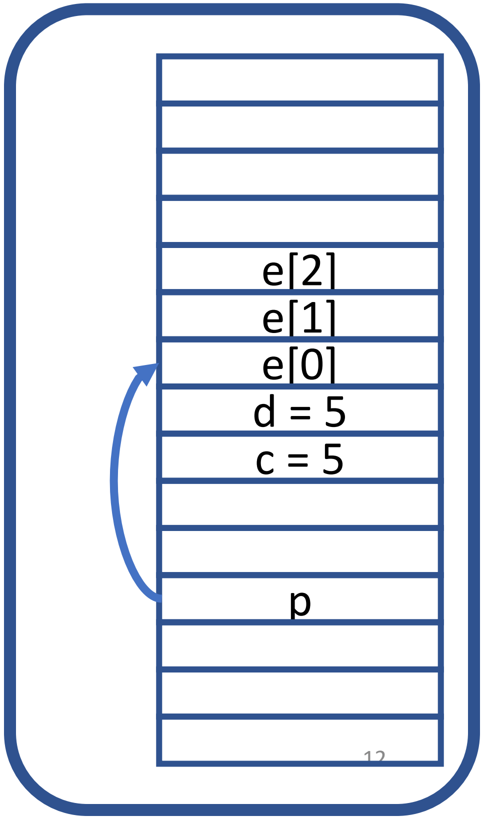


Dereferencing operator * gives the object pointed by p.
So, d gets the value of c.

Example: Use of pointers

```
int c = 5, d, e[3];
int *p;         // Declared pointer p of type int

p = &c;         // p now points to c
d = *p;         // d is now 5
p = &e[0];      // p now points to e[0]
```

p now points the first element of array e[ ].
So, p contains the address of the memory location where e[0] is residing.



Pointer to an item of different type

- A pointer should point to an object of the same type.

- There are implications if a pointer points to a different type.

```
int i;
char c;
int *p;    // p can point to int
p = &i;    // Correct
p = &c;    // Can result in wrong calculations
```

Example: Implications of pointing to different type

- `int` pointer 'thinks' the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer 'thinks' the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer 'thinks' the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example: `int i = 0xAABBCCDD;`

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example: `int i = 0xAABBCCDD;`

AA	BB	CC	DD

first byte　　　　　　　　last byte

Storage of i on big-endian computer

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example: `int i = 0xAABBCCDD;`

DD	CC	BB	AA

first byte last byte

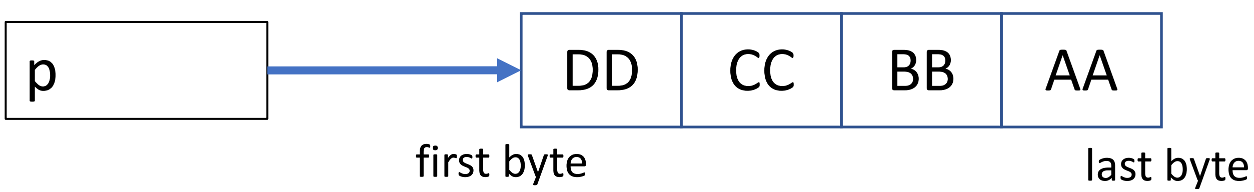
Storage of i on little-endian computer

Example: Implications of pointing to different type

- `int` pointer 'thinks' the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer 'thinks' the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer 'thinks' the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]

```
type *p = &i;
```

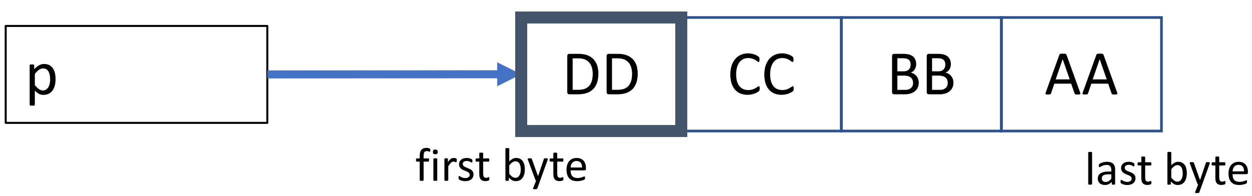


Fact: Pointer always gets the address of the first byte.

Example: Implications of pointing to different type

- `int` pointer 'thinks' the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer 'thinks' the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer 'thinks' the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example:
```
int i = 0xAABBCCDD;
char *p = &i;
```
[Consider little-endian]



Since p is `char` pointer, *p has value `0xDD`

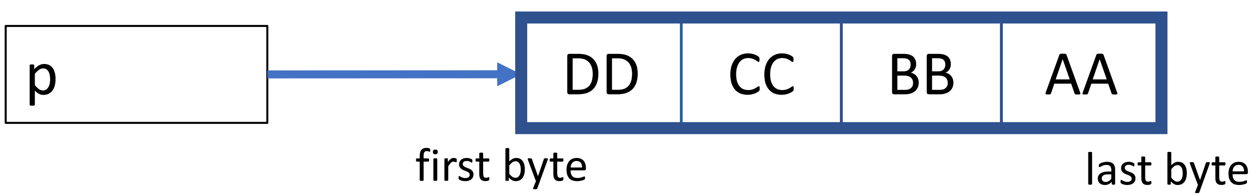
Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example:

```
int i = 0xAABBCCDD;
int *p = &i;
```

[Consider little-endian]



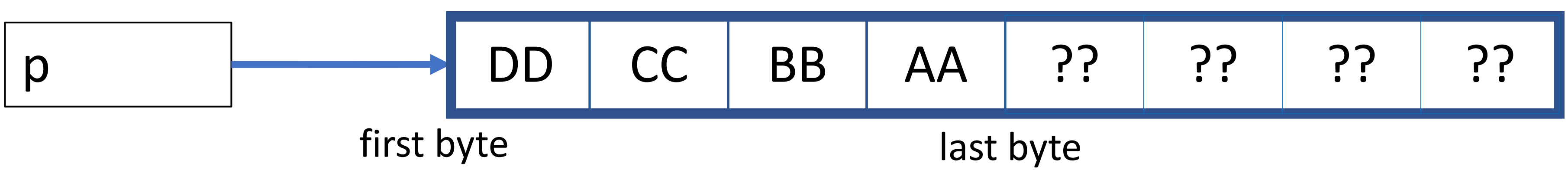
Since p is `int` pointer, *p has value `0xAABBCCDD`

Example: Implications of pointing to different type

- `int` pointer 'thinks' the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer 'thinks' the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer 'thinks' the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types …]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]

```
long long *p = &i;
```



Since p is `long long` pointer, *p has value `0xAABBCCDD????????`

Never assign an absolute address to a pointer!

```
int *p = 100; // illegal assignment to pointer
```

Use of pointer in expressions (1)

- Pointer variables can appear in expressions
- If ‘p’ points to the object ‘x’, then *p can occur in any context where x could.

```c
int x;
...
x = x + 10;
```

is equivalent to

```c
int x;
int *p = &x;
...
*p = *p + 10;
```

- Unary operators ‘*’ and ‘&’ have higher precedence than arithmetic operators.

```c
*p = *p + 10;
```

is

```c
(*p) = (*p) + 10;
```

Use of pointer in expressions (2)

`*p = *p + 1;` is `++*p;`

But, `*p = *p + 1;` is not `*p++;`

Explanation:

Prefix ++ and -- operators have same precedence as unary * and &

Postfix ++ and -- operators have higher precedence than unary * and &

So, parenthesis is needed.

`*p = *p + 1;` is `(*p)++;;`

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + − ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ −	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	= += −= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;
int *p1 = &x;
int *p2;
int *p3 = p1;
p2 = p1;
```

x

Use of pointer in expressions (3)

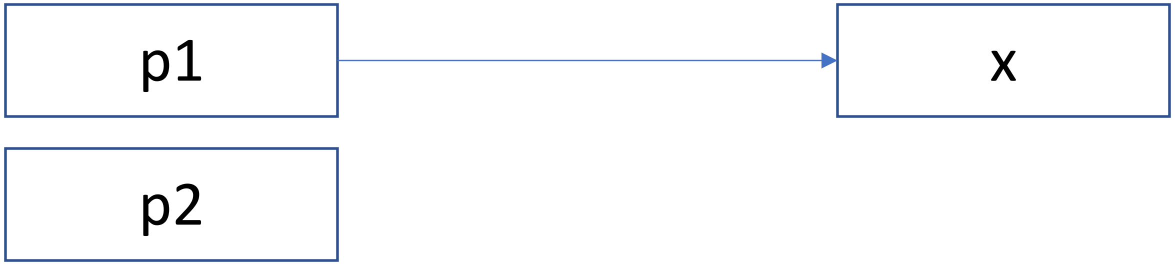
- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;
int *p1 = &x;
int *p2;
int *p3 = p1;
p2 = p1;
```

Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

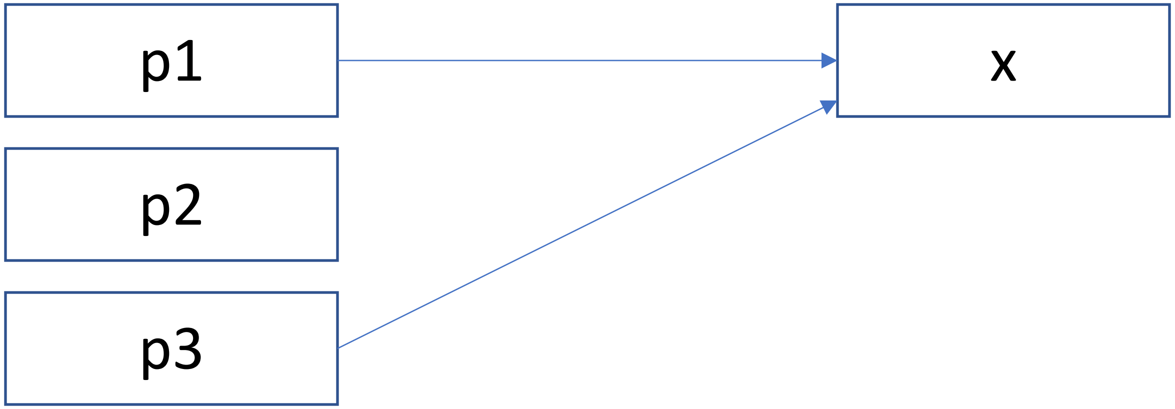
```
int x;
int *p1 = &x;
int *p2;
int *p3 = p1;
p2 = p1;
```



Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

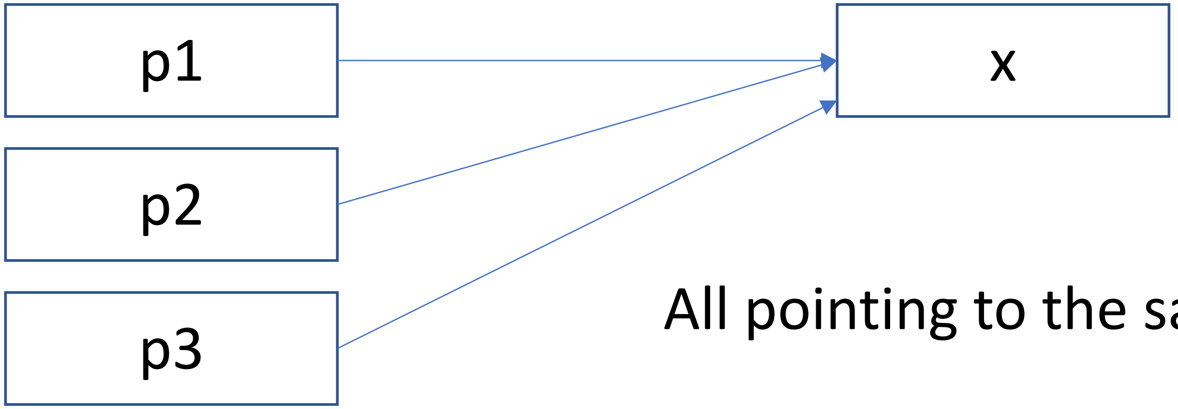
```
int x;
int *p1 = &x;
int *p2;
int *p3 = p1;
p2 = p1;
```



Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;
int *p1 = &x;
int *p2;
int *p3 = p1;
p2 = p1;
```



All pointing to the same object

Scale factor in pointer expression

- If p is a pointer then p++ or p+1
  - points to the next object of the same type. So,
  - value of p increments by 4 when p is an `int` pointer
  - value of p increments by 4 when p is a `float` pointer
  - value of p increments by 1 when p is a `char` pointer
- Scale factor is the number of bytes used by a data-type
- To know scale factor for a data-type, use the sizeof() function
- Syntax is: **sizeof(data_type)**

Size of different data types

```c
int main(){
        printf("Size of int in bytes %d\n", sizeof(int));
        printf("Size of char in bytes %d\n", sizeof(char));
        printf("Size of float in bytes %d\n", sizeof(float));

        printf("Size of int* in bytes %d\n", sizeof(int*));
        printf("Size of char* in bytes %d\n", sizeof(char*));
        printf("Size of float* in bytes %d\n", sizeof(float*));
}
```

Program prints:
Size of int in bytes 4
Size of char in bytes 1
Size of float in bytes 4
Size of int* in bytes 8
Size of char* in bytes 8
Size of float* in bytes 8

See, pointer variables take always 8 bytes, independent of the type it points to. Why?

Pointers and Arrays

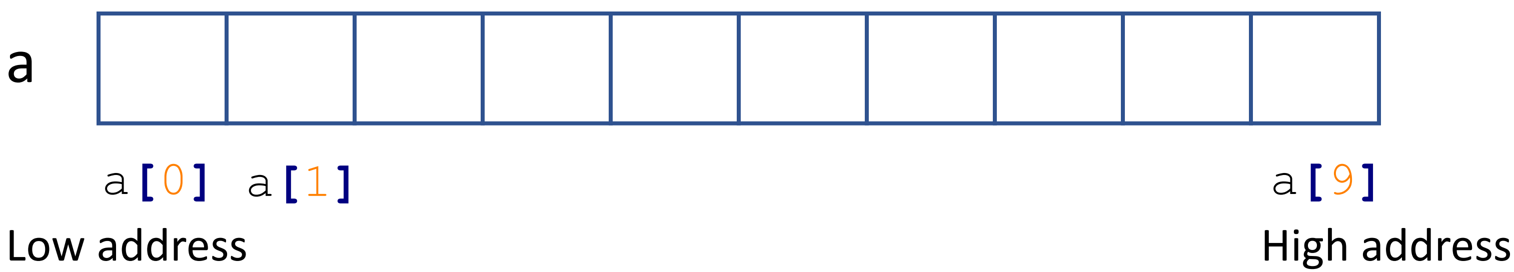
Storage of Array elements

- Array is a **sequential** collection of elements of the same type.

Example:

```
int a[10];
```

is a block of 10 consecutive objects named `a[0]`,`a[1]`,…,`a[9]`.

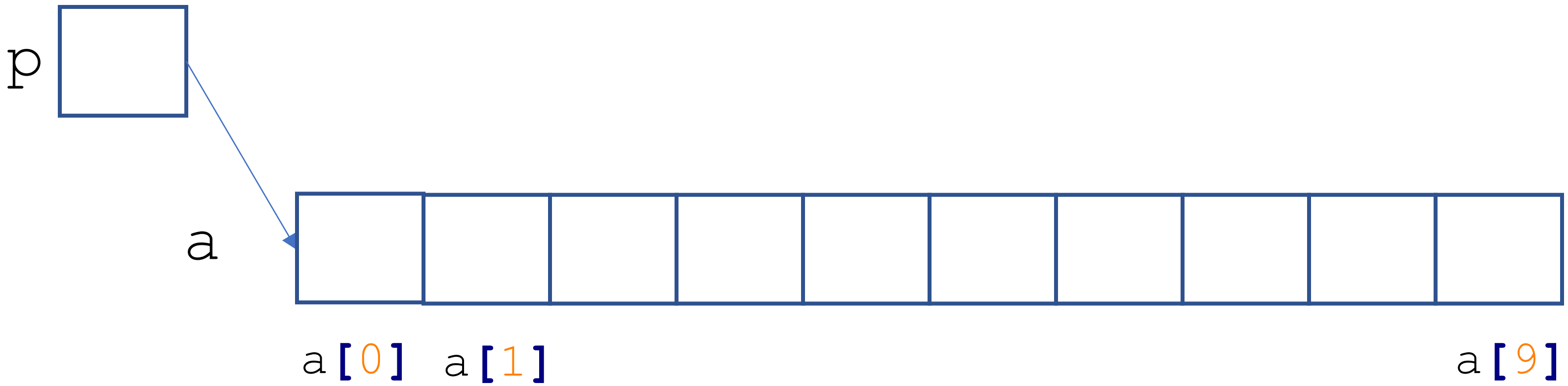


- The notation a[i] refers to the i-th element of the array.

Storage of Array elements

```
int a[10];
int *p = &a[0];
```

p is a pointer to the first element of array a.



Now

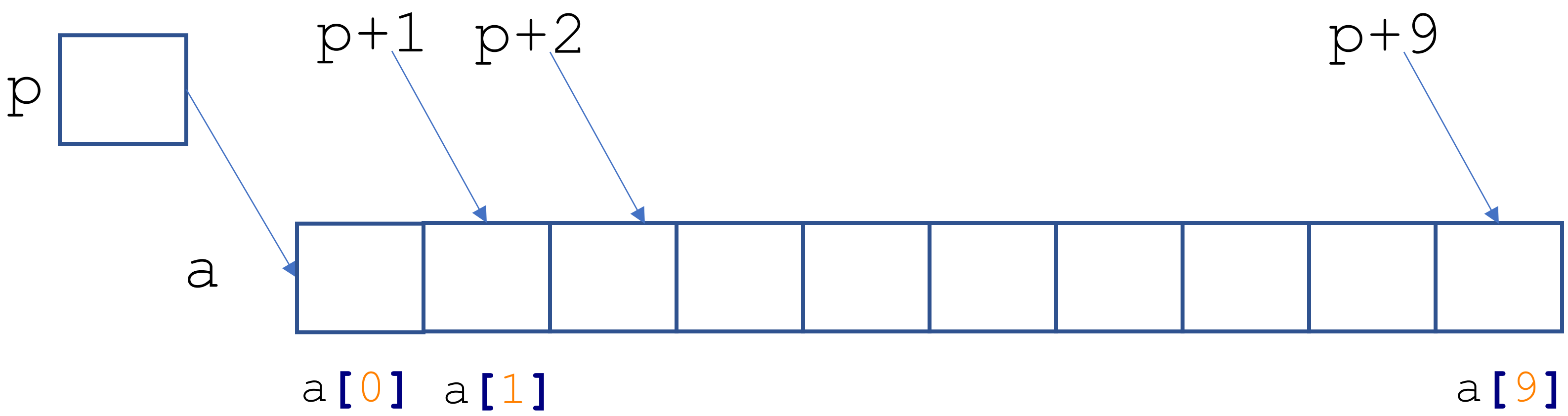
```
int b = *p;
```

will copy value of a[0] into b.

Accessing Array elements using Pointer

If p points to the first element a[0], then

- p+1 points to a[1]
- p+i points to a[i]



So, *(p+i) refers to the content of a[i]

Example: Accessing Array elements using Pointer

Compute the sum of the integer array

```c
int a[] = {2,4,5,7,0,1,9,4,8,3,11};
```

```c
int sum=0, i;
for(i=0; i<5; i++)
  sum = sum + a[i];
```

Using array indexing

```c
int *p = &a[0];
int sum=0, i;
for(i=0; i<5; i++)
  sum = sum + *(p+i);
```

Using pointer

Array 'name' is an address

Array-name is a synonym for the location of the initial element.

So,

```
int *p = &a[0];
```

can also be written as

```
int *p = a;
```

→ You can think of array-name 'a' as pointer to the array a[ ].

```
int *p = &a[0];
int sum=0, i;
for(i=0; i<5; i++)
  sum = sum + *(p+i);
```

Sum calculation using pointer.

```
int sum=0, i;
for(i=0; i<5; i++)
  sum = sum + *(a+i);
```

Sum calculation: treating array name as pointer.

Array 'name' is an address: pitfalls

An array-name is a constant expression, not a variable.

So,

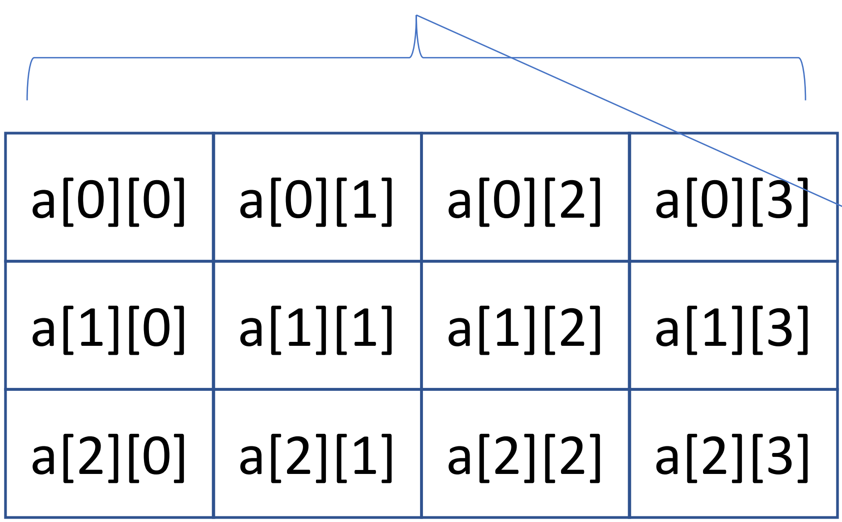
```
int a[10];
int *p = &a[0];
p++;   // legal since p is a variable
p=p+1; // legal

a++;   // illegal since a is not a variable
a=a+1; // illegal
```

Pointers to 2D Arrays

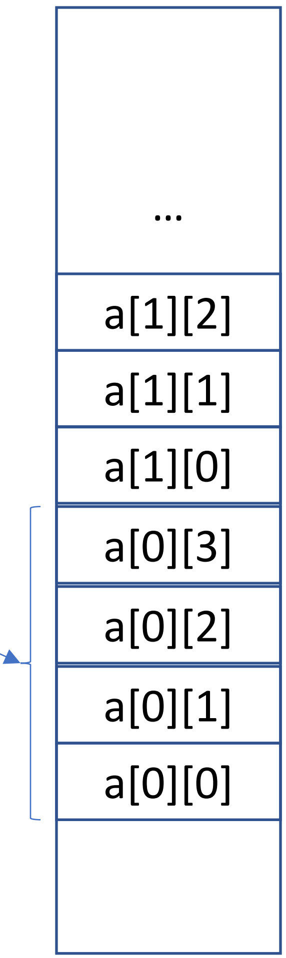
Recap: Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on



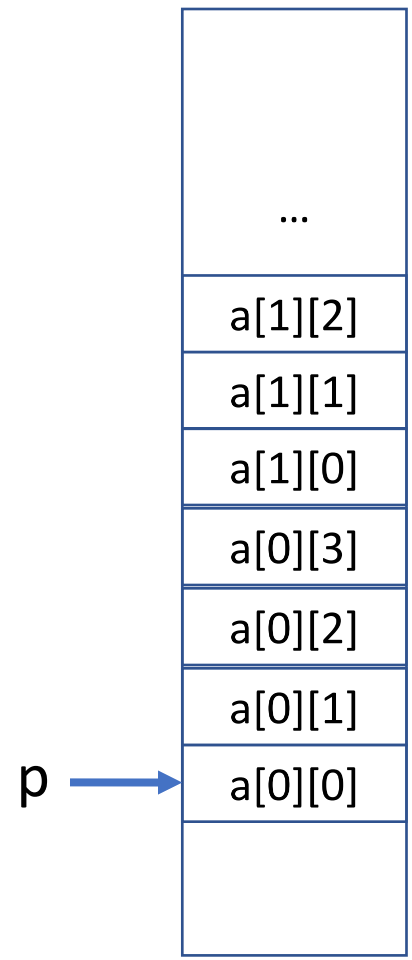
Logical view of array a[3][4]



Memory layout

Accessing 2D array elements using pointer

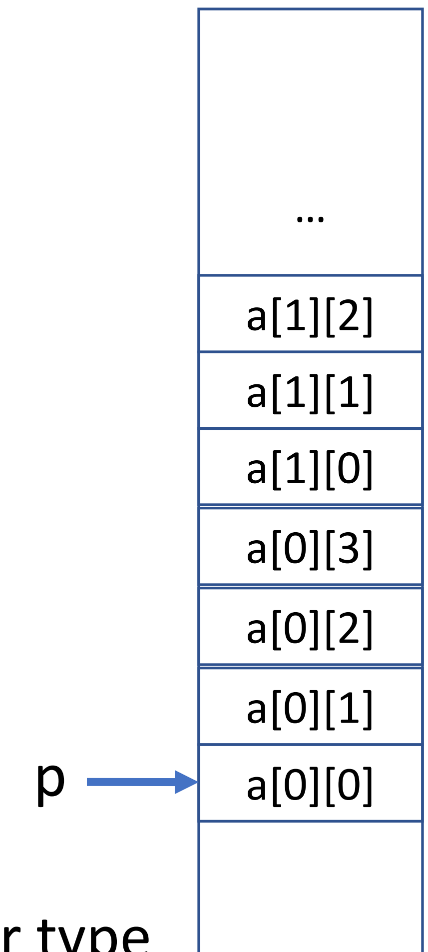
```
#define ROW 3
#define COL 4
int main(){
        int a[ROW][COL] = {{1,2,3,4},
                           {5,6,7,8},
                           {9,10,11,12}};

        int *p = &a[0][0];
        int i;
        for(i=0; i<ROW*COL; i++)
                printf("%d\n", *(p+i));

        return 0;
}
```



Prints the entire 2D array in row-major order

Accessing 2D array elements using pointer

```
#define ROW 3
#define COL 4
int main(){
        int a[ROW][COL] = {{1,2,3,4},
                           {5,6,7,8},
                           {9,10,11,12}};

        //int *p = &a[0][0];
        int *p = a;
        int i;
        for(i=0; i<ROW*COL; i++)
                printf("%d\n", *(p+i));

        return 0;
}
```



Note: Compiler warns about incompatible pointer type (explained in the next slide)

Remember: Array names of 1D and 2D arrays

- For an 1D array a[ ]
  - the array name a is a constant expression whose value is the address of a[0]
  - a+i is the address of a[i]

- For a 2D array a[ ][ ]
  - The array name a is a constant expression whose value is the address of the 0th **row**
  - a+i is the address of the ith **row**

```c
int a[ROW][COL] = {{1,2,3,4},
                   {5,6,7,8},
                   {9,10,11,12}};

//int *p = &a[0][0];
int *p = a;
```

In this example

- p should point to an int
- However, array name 'a' is the address of 0th **row** (not an int).

Hence, C compiler warns about incompatible

Pointer and string of characters

- A string of characters is a 1D array of characters
- ASCII code (1 byte) of each character element is stored in consecutive memory locations
- String is terminated by the null character '\0' (ASCII value 0).
- The null string (length zero) is the null character only

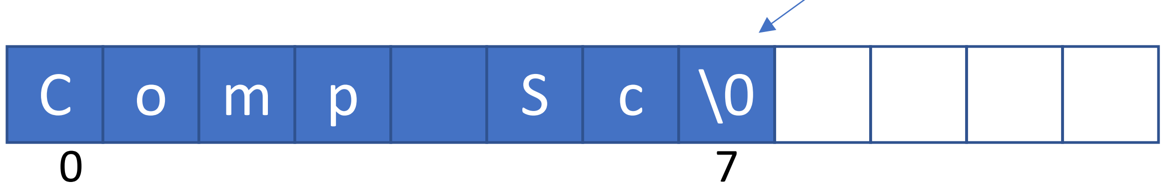
```
char name[ ] = "Comp Sc";
```

C	o	m	p		S	c	\0				
0							7				

String access using pointer

```
int main(){
        char name[ ] = "Comp Sc";
        char *ptr = &name[0];

        // print char-by-char
        while(*ptr != '\0'){
                printf("%c", *ptr);
                ptr++;
        }
        return 0;
}
```

\0 is used to indicate termination of a string



Length is 7, but the array is [0 … 7]

String access using pointer: direct initialization

```
int main(){
        //char name[] = "Comp Sc";
        //char *ptr = &name[0];
        char *ptr = "Comp Sc";

        // print char-by-char
        while(*ptr != '\0'){
                printf("%c", *ptr);
                ptr++;
        }
        return 0;
}
```

With such a declaration, the string gets stored in the ‘read-only’ section of the program code.

String access using pointer: direct initialization

```
int main(){
        //char name[] = "Comp Sc";
        //char *ptr = &name[0];
        char *ptr = "Comp Sc";
        *ptr = 'c' ;

        // print char-by-char
        while(*ptr != '\0'){
                printf("%c", *ptr);
                ptr++;
        }
        return 0;
}
```

With such a declaration, the string gets stored in the 'read-only' section of the program code.

Any attempt to modify read-only data will cause 'segmentation' fault and the program will crash.