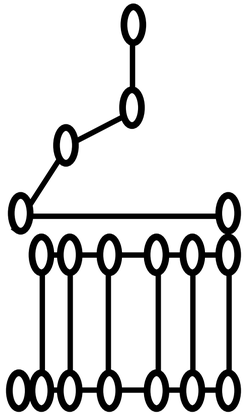
Lecture: More on arrays and pointers

ENGR 2730 Computers in Engineering



CQ: What is the output of main()?

```
int arraySum(int arr[], int arr_size);

int main()
{
    int a[3] = {2, 2, 2};
    int n = 3;
    int sum;

    sum = arraySum(a, n);

    cout << sum << " " << a[0] << " " << a[1] << " " << a[2] << endl;

    return 0;
}

int arraySum(int arr[], int arr_size)
{

    for (int i=1; i < arr_size; i++)
    {
        arr[i] = arr[i] + arr[i-1];
    }

    return arr[arr_size - 1];
}
```

A: 4 2 2 2

B: 6 2 2 2

C: 4 2 4 2

D: 6 2 4 6

CQ: What is the output of main()?

```
int arraySum(int arr[], int arr_size);

int main()
{
    int a[3] = {2, 2, 2};
    int n = 3;
    int sum;

    sum = arraySum(a, n);

    cout << sum << " " << a[0] << " " << a[1] << " " << a[2] << endl;

    return 0;
}

int arraySum(int * arr, int arr_size)
{

    for (int i=1; i < arr_size; i++)
    {
        arr[i] = arr[i] + arr[i-1];
    }

    return arr[arr_size - 1];
}
```

A: 4 2 2 2

B: 6 2 2 2

C: 4 2 4 2

D: 6 2 4 6

Changed “int arr[]” to “int * arr”.
Works the same.

Summary of our first similarity between arrays and pointers

```
int arraySum(int arr[], int arr_size);
```

(you may use either declaration)

```
int arraySum(int *arr, int arr_size);
```

Since a pointer is passed rather than the whole array, modifying the array within the function also causes the original array to be modified!

But wait!

Why can you use the syntax ‘arr[i]’ if arr is a pointer? We have only seen using * (the indirection/dereferencing operator) with pointers...

```
int arraySum(int *arr, int arr_size)
{

    for (int i=1; i < arr_size; i++)
    {
        arr[i] = arr[i] + arr[i-1];
    }

    return arr[arr_size - 1];
}
```

Similarities between pointers and arrays

- In expressions, an array name is treated as a constant pointer to the first element of the array.

```
int a[] = {1, 2, 3};
int *p = a;
```

- A subscript is equivalent to an offset from a pointer.

`a[2]` ⟷ `*(a + 2)`

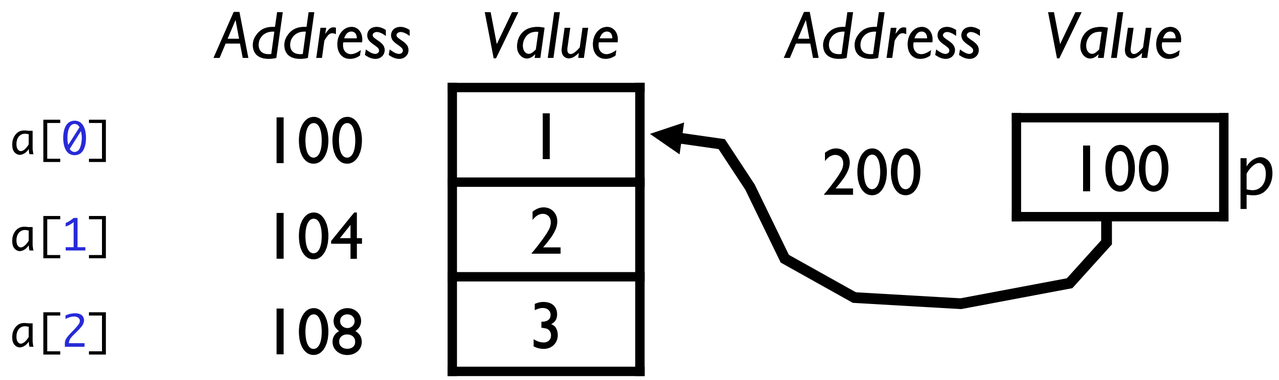
Similarities between arrays and pointers in expressions

- In expressions, an array name is a const pointer to the first element of the array.

```
int a[] = {1, 2, 3};
int * p = a;
```

- A subscript is equivalent to an offset from a pointer.

`a[2]` ⟷ `*(a + 2)`



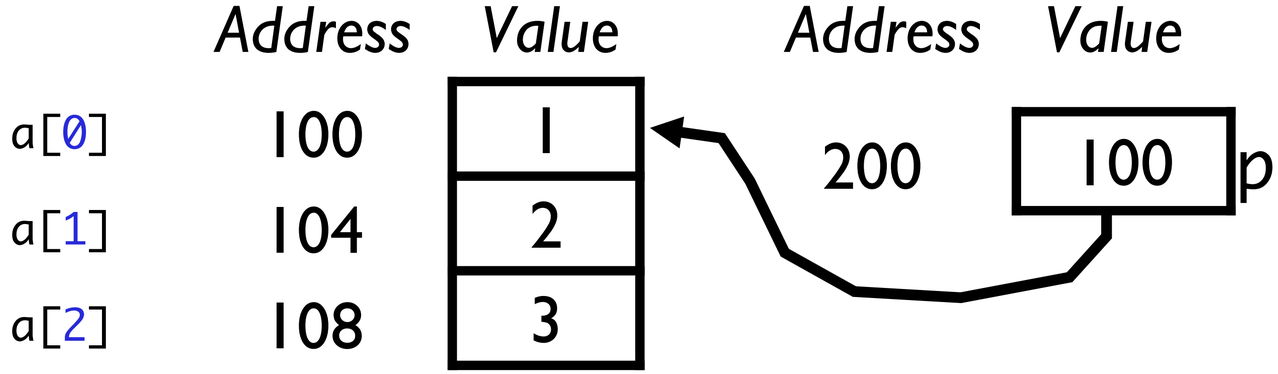
Similarities between arrays and pointers in expressions

- In expressions, an array name is (usually) treated as a pointer to the first element of the array.

```
int a[] = {1, 2, 3};
int *p = a;
```

- A subscript is equivalent to an offset from a pointer.

`a[2]` ⟷ `*(a + 2)`



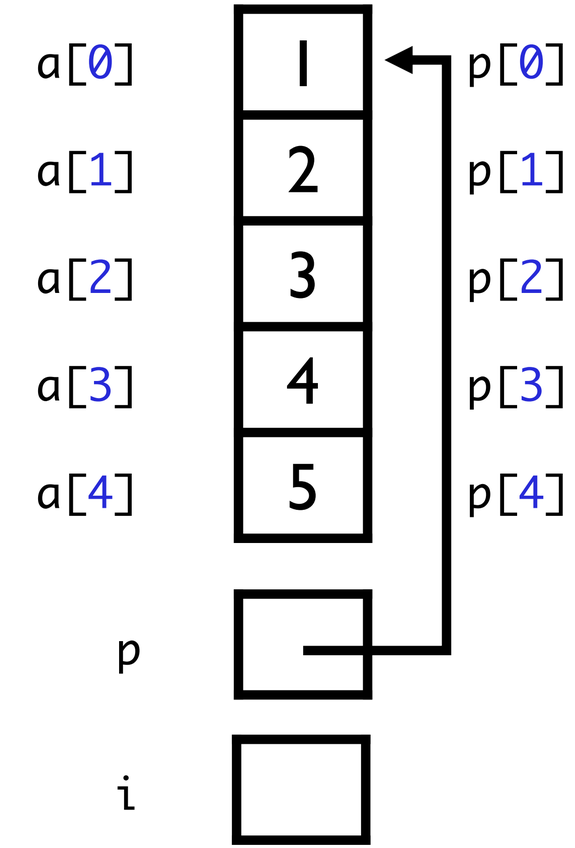
`a[2]` ⟶ `*(a + 2)`

`p[2]` ⟶ `*(p + 2)`

An important practical consequence

If you define an array, you can also use a pointer to modify/access elements of the array using the subscript notation.

```
int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    int *p = a;


    /* use pointer to modify elements of array */
    for (int i = 0; i < 5; i++)
    {
        p[i] = p[i] + 1;
    }
    /* print array */
    for (int i = 0; i < 5; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl ;
}
```

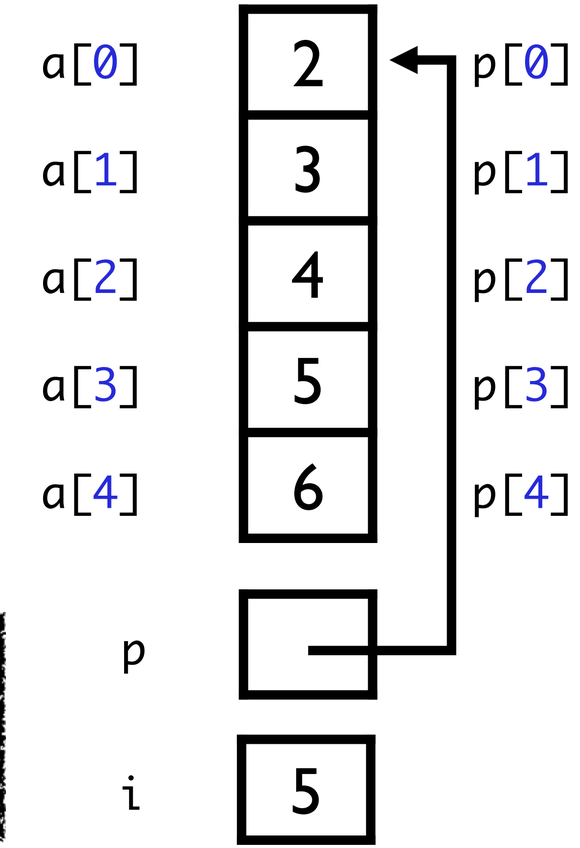


An important practical consequence

If you define an array, you can also use a pointer to modify/access elements of the array using the subscript notation.

```
int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    int *p = a;


    /* use pointer to modify elements of array */
    for (int i = 0; i < 5; i++)
    {
        p[i] = p[i] + 1;
    }
    /* print array */
    for (int i = 0; i < 5; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl ;
}
```

Output:
2 3 4 5 6

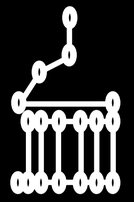


CQ 1: Which memory diagram below best reflects the following code?

```
int main()
{
    int a[3] = {1, 2, 3};
    int b = 5;
    int *p1 = nullptr;
    int *p2 = nullptr;

    p1 = a;
    p2 = &b;

    p1[1] = *p2;
    b = 2;
    p2[0] = 7;

    return 0;
}
```

choices to appear

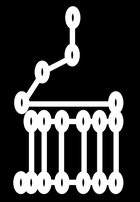
	Address	Value
a[0]	100	
a[1]	104	
a[2]	108	
b	112	
p1	116	
p2	120	



CQ 1: Which memory diagram below best reflects the following C++ code?

```
int main()
{
    int a[3] = {1, 2, 3};
    int b = 5;
    int *p1 = nullptr;
    int *p2 = nullptr;

    p1 = a;
    p2 = &b;

    p1[1] = *p2;
    b = 2;
    p2[0] = 7;

    return 0;
}
```

	Address	A. Value	B. Value
a[0]	100	1	1
a[1]	104	5	2
a[2]	108	3	3
b	112	7	2
p1	116	100	100
p2	120	112	7



CQ 2: What is printed as a result of calling the following function?

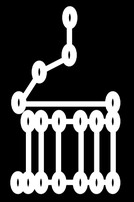
```
int main()
{

    int c[5] = {0};
    int *p = &(c[1]);

    for (int i = 0; i < 3; i++)
    {
        p[i] = 1;
    }

    for (int i = 0; i < 5; i++)
    {
        cout << c[i] << " " ;
    }
    cout << endl ;

    return 0;
}
```

A. 1 1 1 0 0

B. 0 1 1 1 0

C. 0 1 1 1 1

D. 1 1 1 1 0



CQ 3: Will the following function compile/run?

```
int main()
{
3       int a[5] = {1, 2, 3};


        for (int i = 0; i < 5; i++)
        {
8           cout << i[a] << " " ;
        }
        cout << endl ;

}
```

A. Yes

B. No, line 3 has a problem.

C. No, line 8 has a problem.

Warning: Do **NOT** write code like this!
This example is only to help emphasize that
i[a] becomes *(i + a) just like
a[i] would become *(a + i).

Using **const** with pointers

Motivation

Can we specify that the values of an array passed to a function should not be modified (so that we will receive a **compiler error** if we try to modify it)?

```
int main() {
    int a[5] = {2, 2, 2, 2, 2};

    printArray(a, 5);
    tryToModifyArray(a, 5);
    printArray(a, 5);

    return 0;
}

void tryToModifyArray(int *arr, int n) {
    int i;
    for (i=0; i < n; i++) {
        arr[i] = arr[i] - 1;
    }
}
```

⟵ suppose we do not want to allow this

Motivation

Can we specify that the values of an array passed to a function should not be modified (so that we will receive a **compiler error** if we try to modify it)?

```
int main(){
    int a[5] = {2, 2, 2, 2, 2};

    printArray(a, 5);
    tryToModifyArray(a, 5);
    printArray(a, 5);

    return 0;
}

void tryToModifyArray(const int *arr, int n){
    int i;
    for (i=0; i < n; i++){
        arr[i] = arr[i] - 1;
    }
}
```

We can use:
`const int *arr`

arr is a pointer to a constant integer

← suppose we do not want to allow this

compiler error: “assignment of read-only location”

Example use of const int *

```
int main() {
    int a[5] = {1, 1, 1, 1, 1};
    int n = 5;
    int sum;

    sum = computeArraySum(a, n);
    printArray(a, n);
    cout << "sum = " << sum << endl;

    return 0;
}

int computeArraySum(const int * arr, int n){
    int sum = 0;
    int i;

    for (i = 0; i < n; i++){
        sum += arr[i];
    }

    return sum;
}
```

computeArraySum() does not need to modify any of the elements of arr so prevent the function from changing the array values by using const as shown.

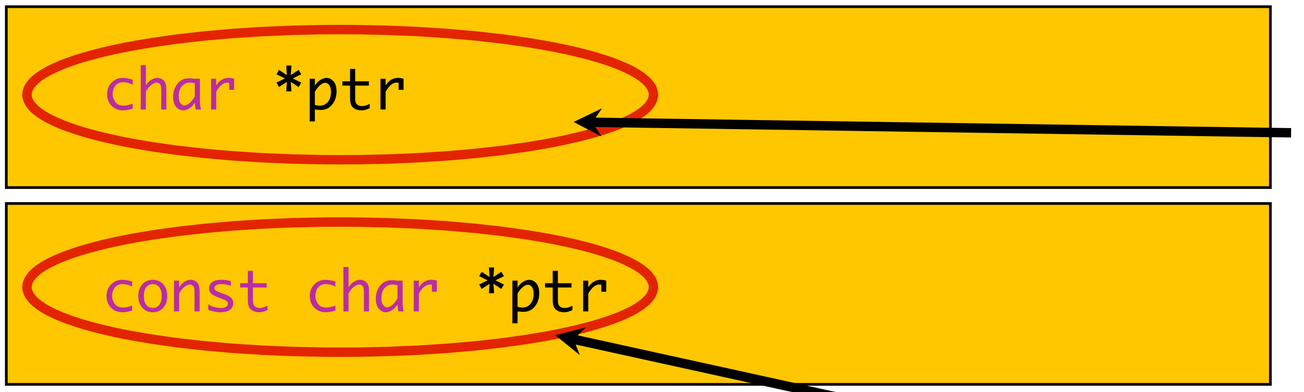
Some **const** variations with pointers

```
char * ptr;
//ptr is a pointer to a character

const char * ptr;
//ptr is a pointer to a character that is constant

char * const ptr;
//ptr is a constant pointer to a character

const char * const ptr;
//ptr is a constant pointer to a character that is constant
```

(read from right to left)

Some **const** variations with pointers

Two important cases to understand as parameters of functions



allows
you to modify the value of the pointer and value of the character

allows
you to modify the value of the pointer but NOT the value of the character

```
char * const ptr
```

```
const char * const ptr
```

(read from right to left)

An example of modifying the value of a pointer within a function

```
/* a C string example */
int main()
{
    char string[] = "a sample string";
    cout << "The original string: "<< string << endl;
    convertToUpperCase(string);
    cout << "The modified string: " << string << endl;

    return 0;
}

/* convert string to uppercase letters */
void convertToUpperCase(char *sPtr)
{
    while (*sPtr != '\0')  /* while current character is not '\0' */
    {
        if (islower(*sPtr))  /* if character is lowercase */
        {
            *sPtr = toupper(*sPtr); /* convert to uppercase */
        }
        ++sPtr; /* move sPtr to the next character */
    }
}
```

The **sizeof** operator

The **sizeof** operator can be used to determine the size (in bytes) of a data type

```
char c;
short s;
int i;
long l;
float f;
double d;
long double ld;
int array[20];
int *ptr = array;
```

```
cout << "     sizeof c = "<< (int) sizeof c
     << " sizeof(char) = "<< (int) sizeof(char) << endl

     << " sizeof s = "<< (int) sizeof s
     << " sizeof(short) = " << (int) sizeof(short)<< endl

     << "sizeof i = "<< (int) sizeof i
     <<"sizeof(int) = " << (int) sizeof(int) << endl

     << "sizeof l = "<< (int) sizeof l
     << "sizeof(long) = " << (int) sizeof(long) << endl

     << "sizeof f = "<< (int) sizeof f
     << "sizeof(float) = " << (int) sizeof(float)<< endl

     << "sizeof d = "<< (int) sizeof d
     << "sizeof(double) = " << (int) sizeof(double)<< endl

     <<" sizeof ld = "<< (int) sizeof ld
     << "sizeof(long double) = " << (int) sizeof(long double) << endl

     <<"sizeof array = "<< (int) sizeof array << endl
     <<"sizeof ptr = "<< (int) sizeof ptr << endl ;
```

The **sizeof** operator can be used to determine the size (in bytes) of a data type

```
char c;
short s;
int i;
long l;
float f;
double d;
long double ld;
int array[20];
int *ptr = array;
```

Example output:

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 16  sizeof(long double) = 16
sizeof array = 80
sizeof ptr = 4
```

```
= "<< (int) sizeof c
<<" sizeof ld = "<< (int) sizeof ld
<< "sizeof(long double) = " << (int) sizeof(long double) << endl

<<"sizeof array = "<< (int) sizeof array << endl
<<"sizeof ptr = "<< (int) sizeof ptr << endl ;
```

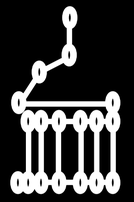
The **sizeof** operator can be used to determine the size (in bytes) of a data type

```
char c;
short s;
int i;
long l;
float f;
double d;
long double ld;
int array[20];
int *ptr = array
```

```
cout << "     sizeof c = "<< (int) sizeof c
     << " sizeof(char) = "<< (int) sizeof(char) << endl

     << " sizeof s = "<< (int) sizeof s
     << " sizeof(short) = " << (int) sizeof(short)<< endl

     << "sizeof i = "<< (int) sizeof i
     << "sizeof(int) = " << (int) sizeof(int) << endl
```

Portability Tip 8.2

The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, always use `sizeof` *to determine the number of bytes used to store the data types.*



CQ: What is printed as a result of calling sizeOfClickerQuestion()? (Assume data types have sizes as indicated on sizeofExample slide.)

```
int main()
{
    int array[8];

    cout <<  (int) sizeof array << " " << obtainSize(array) << endl;

    return 0;
}

int obtainSize(int a[])
{
    return sizeof a;
}
```

A. 32 32

B. 32 4

C. 4 4

Be careful! Remember the pointer to the first element is passed to a function rather than the entire array.