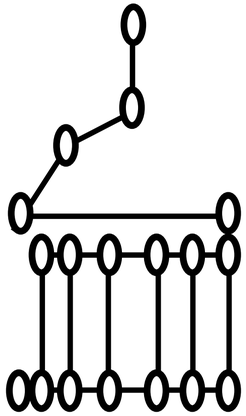
Lecture: Pass by simulated reference

ENGR 2730: Computers in Engineering



Pass-by-reference using pointers

Using pointers in C++

- Pointers should be initialized to nullptr (new in C++11) or to a memory either when they're declared or in an assignment.
- A pointer with the value nullptr "points to nothing" and is known as a null pointer.

Error-Prevention Tip 8.1

Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.

Good Programming Practice 8.1

Although it's not a requirement, we like to include the letters `Ptr` *in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.*

Simulated Pass-by-reference with Pointers

- There are three ways in C++ to pass arguments to a function
  - Pass-by-value
  - Pass-by-reference with reference arguments (*type* &)
  - Pass-by-reference with pointer argument (*type* *)

- Pointers can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of copying the objects

CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

```
#include <iostream>

void addFive(int *val);

int main()
{
    int x = 0;
    int *ptr = &x;

    

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```

A. addFive(x);

B. addFive(&x);

CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

```
#include <iostream>

void addFive(int *val);

int main()
{
    int x = 0;
    int *ptr = &x;

    

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```

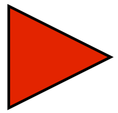
A. addFive(ptr);

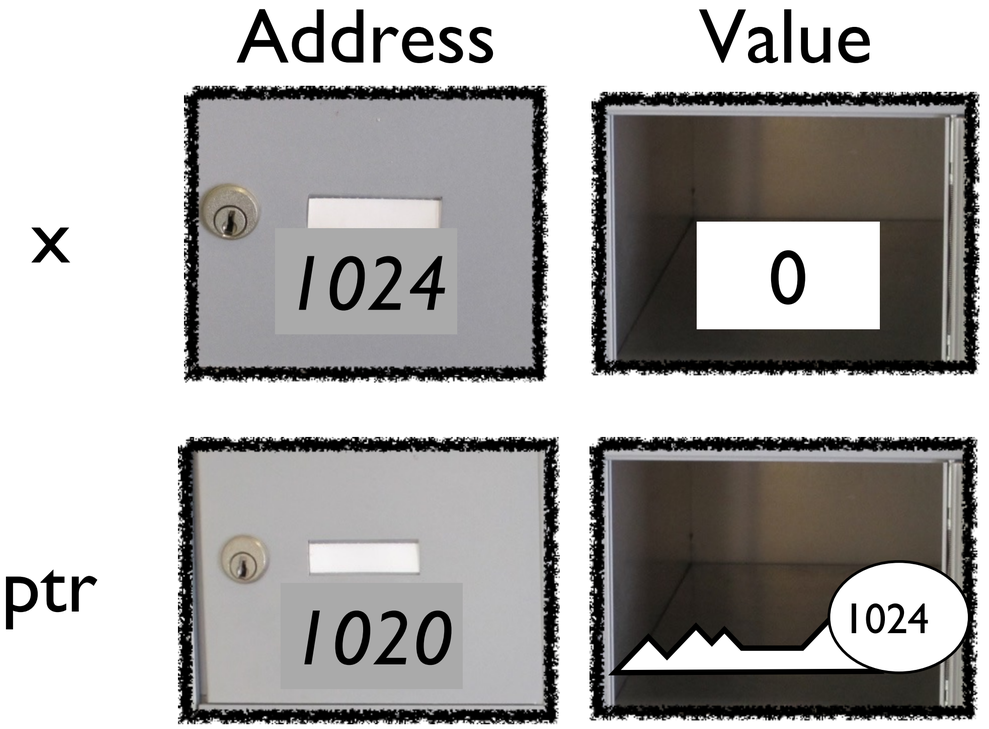
B. addFive(&ptr);

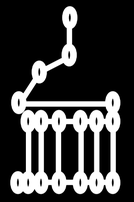
C. none of the above

CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

```
#include <iostream>

void addFive(int *val);

int main()
{
    int x = 0;
    int *ptr = &x;

    addFive(ptr);

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```

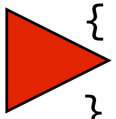


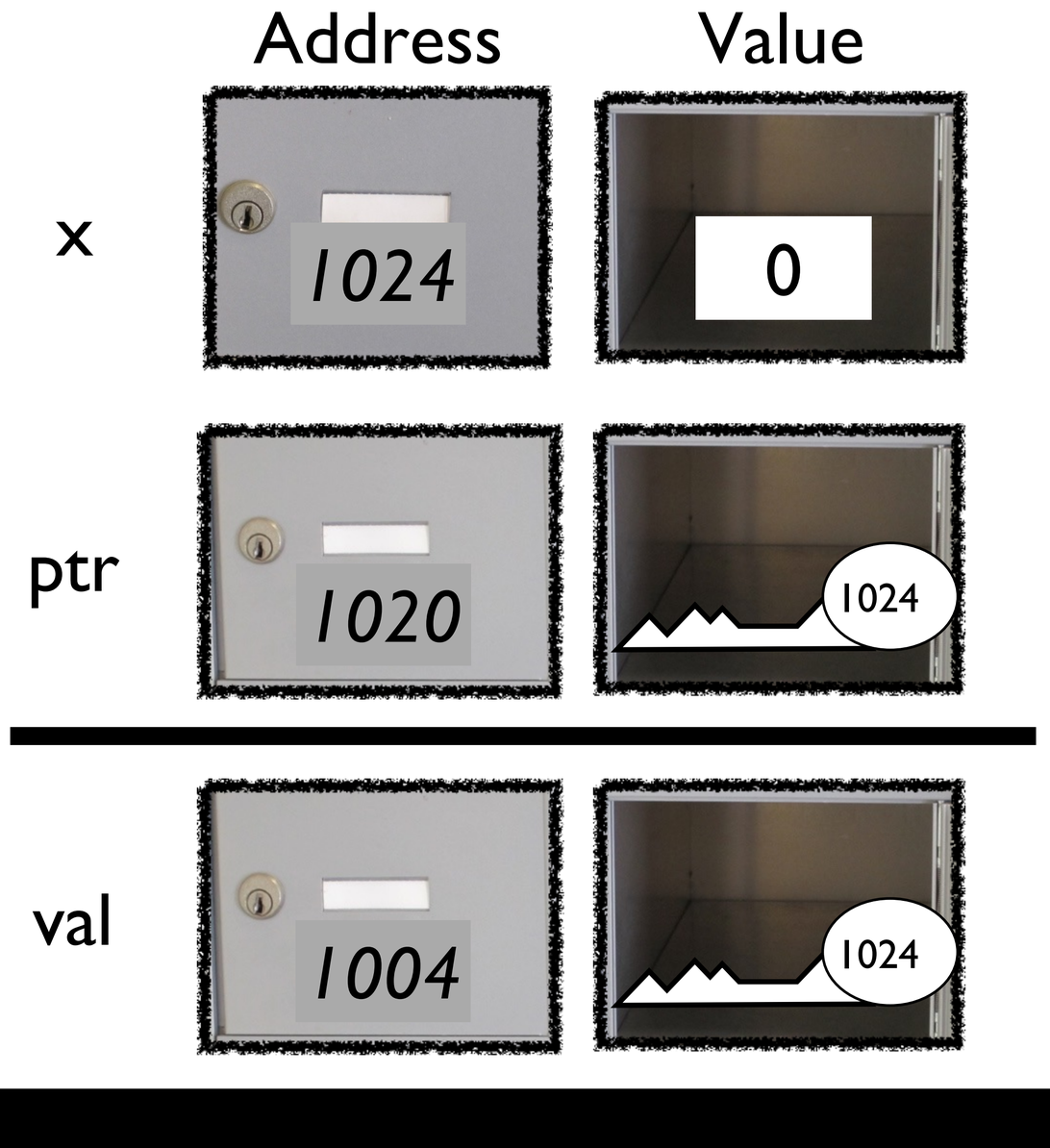




CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

```
#include <iostream>

void addFive(int *val);

int main()
{
    int x = 0;
    int *ptr = &x;

    addFive(ptr);

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```

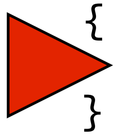


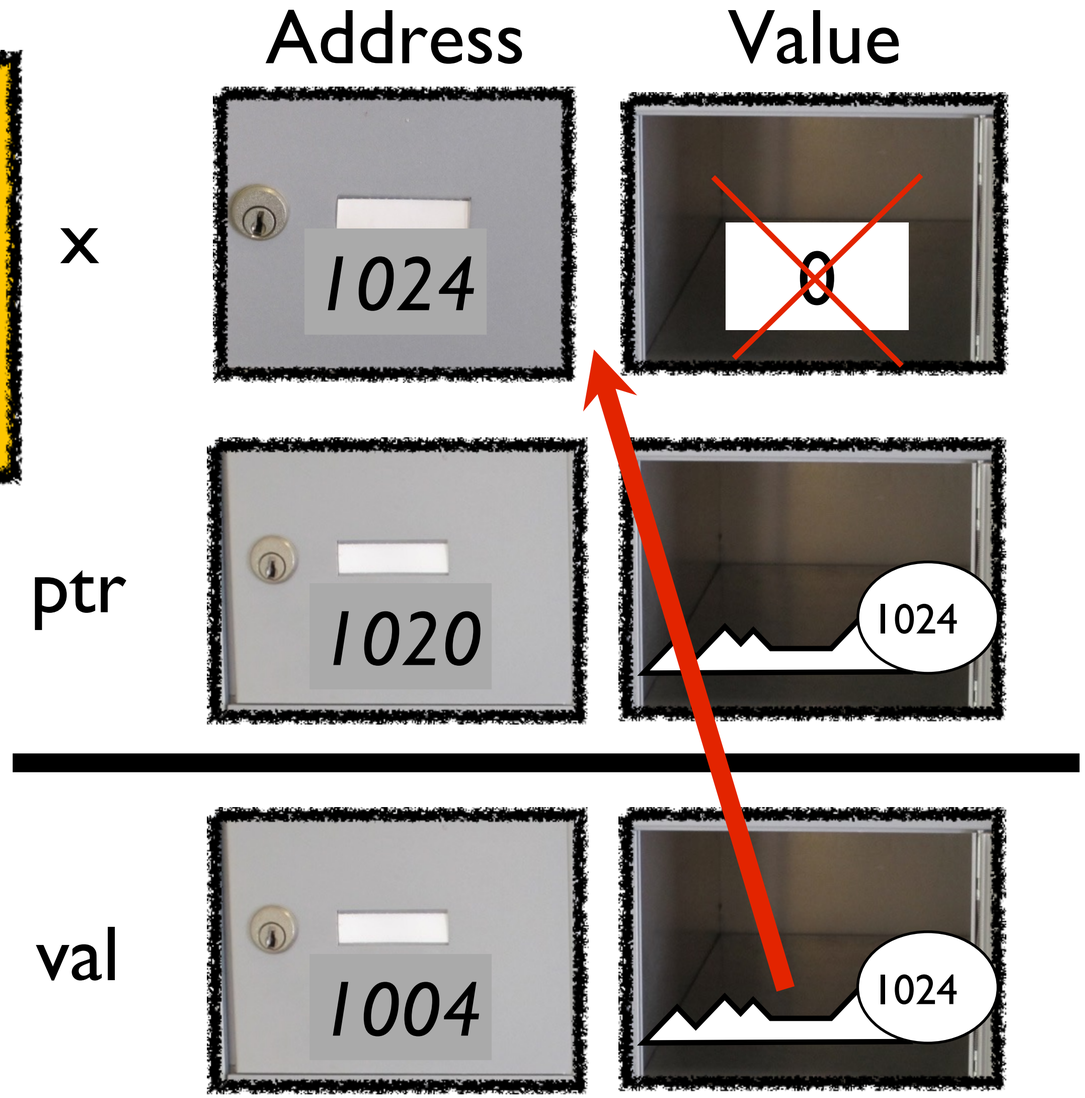


CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

"Obtain the value at memory location 1024 to obtain 0, add 5 to obtain 5, and set the value at memory location 1024 to be 5.

```
    addFive(ptr);

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```

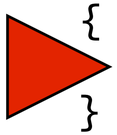


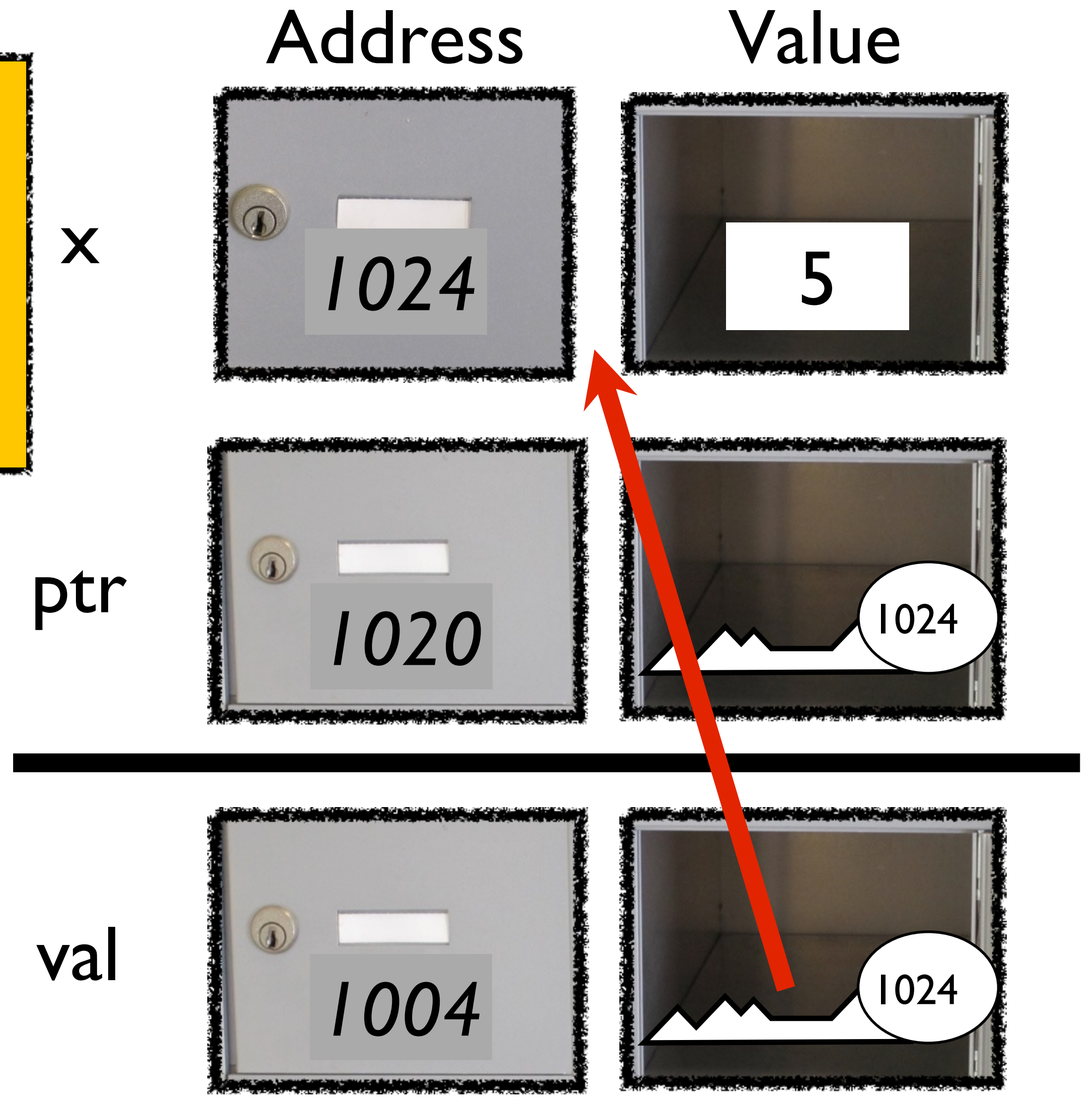


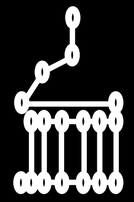
CQ: Suppose you have the partial code below (missing one line). Which added line of code would cause the value of **x** to equal 5?

"Obtain the value at memory location 1024 to obtain 0, add 5 to obtain 5, and set the value at memory location 1024 to be 5.

```
    addFive(ptr);

    cout << "x=" << x << endl;
    return 0;
}

void addFive(int *val)
{
    *val = *val + 5;
}
```







Simulated pass-by-reference

- ***Insight: All Arguments Are Passed By Value***
  - Passing a variable by reference with a pointer *does not actually pass anything by reference*—a pointer to that variable is *passed by value* and is *copied* into the function's corresponding pointer parameter.

  - The called function can then access that variable in the caller simply by dereferencing the pointer, thus accomplishing *pass-by-reference*.

Review of the use of arrays as function parameters

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

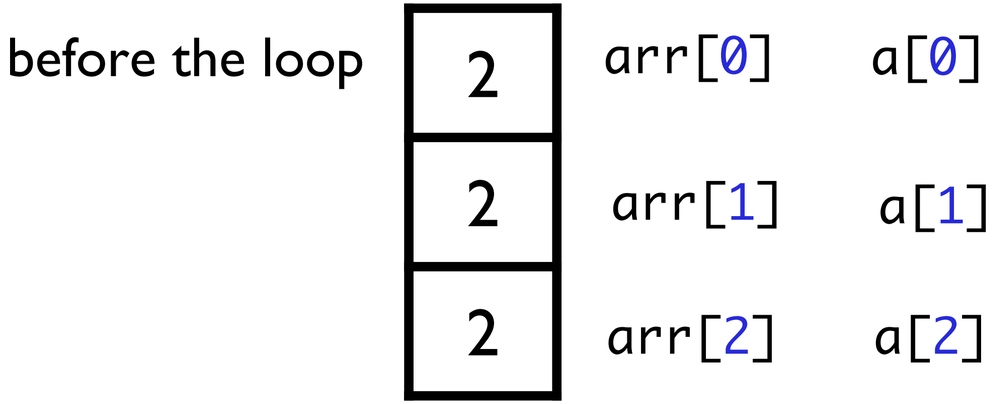
C: 4 2 4 2

D: 6 2 4 6

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];
}
```

If you modify arr, you are also modifying a...



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 = 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];
}
```

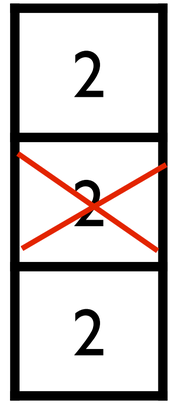
If you modify arr, you are also modifying a...

i = 1:

arr[1] = arr[1] + arr[0]

arr[1] = 2 + 2

arr[1] = 4



arr[0] a[0]

arr[1] a[1]

arr[2] a[2]

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 = 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];
}
```

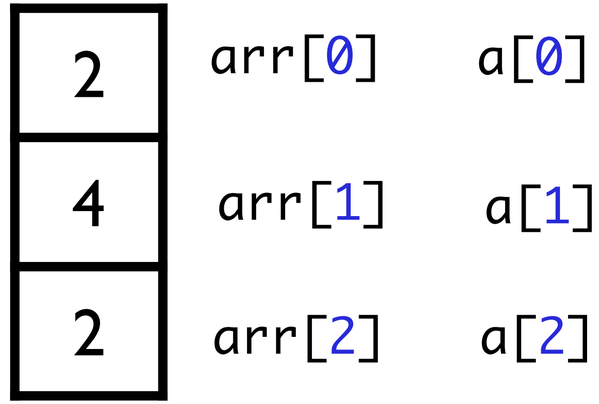
If you modify arr, you are also modifying a...

i = 1:

arr[1] = arr[1] + arr[0]

arr[1] = 2 + 2

arr[1] = 4



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 = 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];
}
```

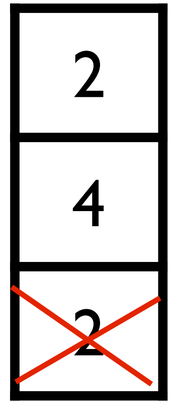
If you modify arr, you are also modifying a...

i = 2:

arr[2] = arr[2] + arr[1]

arr[2] = 4 + 2

arr[2] = 6



arr[0] a[0]

arr[1] a[1]

arr[2] a[2]

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 = 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];
}
```

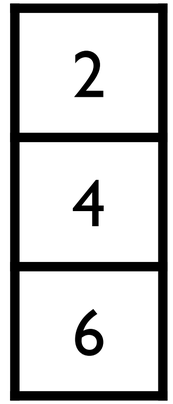
If you modify arr, you are also modifying a...

i = 2:

arr[2] = arr[2] + arr[1]

arr[2] = 2 + 2

arr[2] = 6



arr[0] a[0]

arr[1] a[1]

arr[2] a[2]

Similarities between arrays and pointers

- In function parameters, an array of *type T* is treated by the compiler as a constant pointer to *type T*

```
int arr[] ⟷ int * const arr
```

- In expressions, an array name is 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)
```

The bracket notation in a function parameter list can be replaced with a pointer

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

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

OR

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

The declaration of an array as a function parameter is actually treated as a pointer to the element type

```
int main()
{
    int a[4] = {1, 1, 1, 1};
    int n = 4;
    int sum;

    sum = arraySum(a, n);

    cout << sum << " " << a[0] << " " << a[1] << " " << a[2] << " " << a[3] << endl;

    return 0;
}

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

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

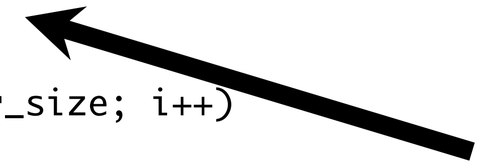
Output:

4 1 2 3 4

The declaration of an array as a function parameter is actually treated as a pointer to the element type

```
int main()
{
    int a[4] = {1, 1, 1, 1};
    int n = 4;
    int sum;

    sum = arraySum(a, n);

    cout << sum << " " << a[0] << " " << a[1] << " " << a[2] << " " << a[3] << endl;

    return 0;
}

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

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

Output:
4 1 2 3 4



This function expects a pointer to an integer...

The declaration of an array as a function parameter is actually treated as a pointer to the element type

```
int main()
{
    int a[4] = {1, 1, 1, 1};
    int n = 4;
    int sum;

    sum = arraySum(a, n);

    cout << sum << " " << a[0] << " " << a[1] << " " << a[2] << " " << a[3] << endl;

    return 0;
}

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

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

The address of the first element is passed...

Output:
4 1 2 3 4

This function expects a pointer to an integer...

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!