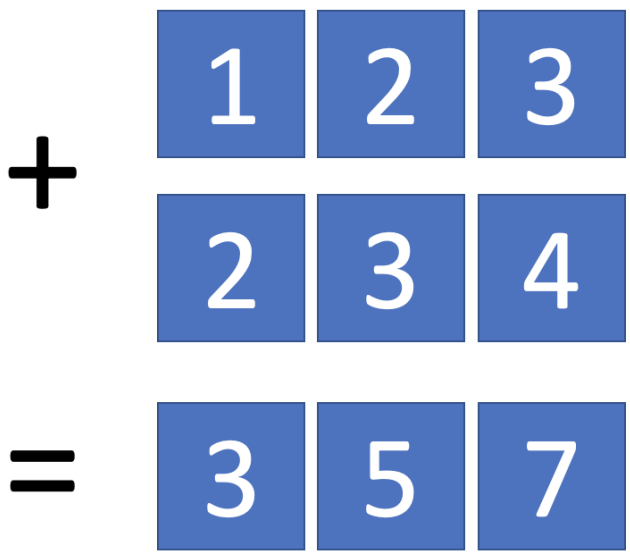
Array and Loops

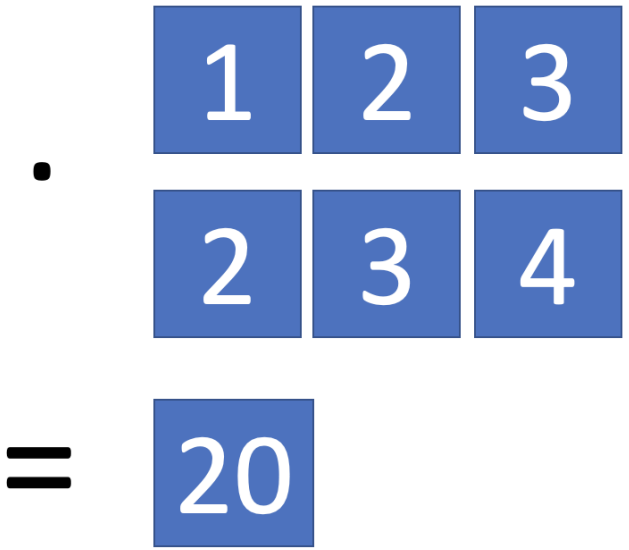
Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

Vector Calculation

- Vectors are a sequence of numbers.
  - $\boldsymbol{a} = [1, 2, 3]$ is a three-dimensional vector.
- Vector calculations: addition, subtraction, dot product, etc.



Vector Addition



Dot Product

Vector Calculation

- Consider the following program tries to compute the three dimensional vector addition

```
#include <stdio.h>
void main(){
    double a1=1.0, a2=2.0, a3=3.0;
    double b1=2.0, b2=3.0, b3=4.0;
    double c1, c2, c3;
    c1 = a1+b1; c2 = a2 + b2; c3= a3 + b3;
    ...
}
```

- Neither the vector initialization nor the computations are automated in the above code.
- What if you have a 100-dimension vector?
    - Handwrite `c1 = a1 + b1; ... c100 = a100 + b100;` ?

Vector Calculation

To automate vector calculations, we need two things:

- A way to access an element in vector via an integer index.
  - e.g. `a[i]` represents the `i`-th element of vector a.
- A way to perform operations on all elements in a vector without explicitly writing down each operation.

Imagine we can do something like:

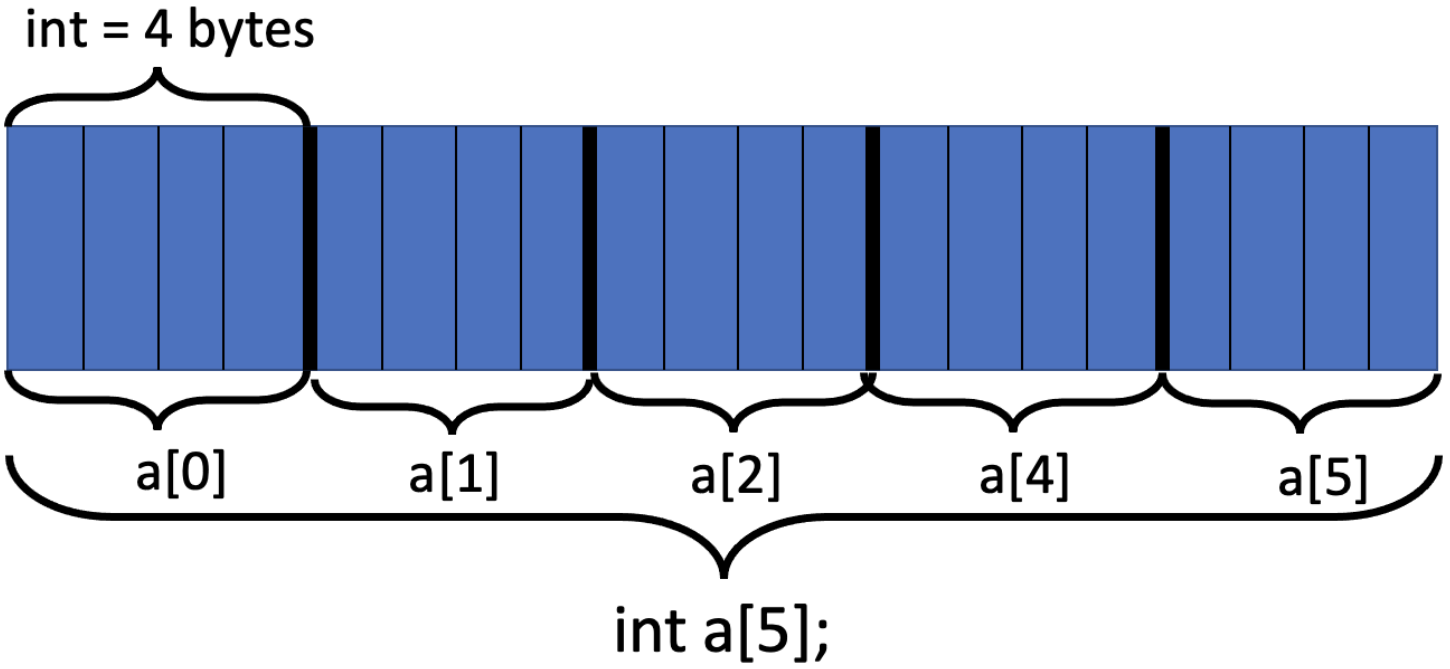
```
i = 1;
Repeat N times
    c[i] = a[i] + b[i];
    i = i + 1;
```

Array

- Array is a fundamental data structure in C programming language that stores a sequence of elements.
- You can declare an array using the syntax:
  - `data_type variable_name[array_size];` .
  - e.g., `int a[100];` declares an 100 int elements array.
- You can initialize it using the syntax:
  - `data_type variable_name[] = {elements};` .
  - e.g., `int a[] = {1,2,3};` . No need to specify the array size.

Array

- **You refer to the first element in the array as** `a[0]`, the second element as `a[1]`, and so on.
  - e.g., `a[2] = 3;` assigns 3 to **the third element** of `a`.
- Array is stored in the memory as a continuous chunk of bytes.



Loops

- To actually operate on arrays, we need a mechanism to carry out operations on elements of arrays.
- In programming language, this mechanism is called loop.
- When encounter loops, the CPU will continue to execute a block of code, until certain exit conditions are met.
    - If the exit conditions are not set properly, CPU may stuck in a loop and will not exit, wasting computational resources.
    - This situation is called infinite loop.

While Loop

The simplest loop is while-loop and its syntax is:

```
while(expression){
    statements
}
```

The statements inside of the brackets will only run if `expression` is true. 3D vector addition can be written as

```
int i = 0;
while(i<3){
    c[i] = a[i] + b[i];
    i = i + 1;
}
```

Vector Addition, Revisited

```c
#include <stdio.h>
void main(){
    double a[] = {1.0, 2.0, 3.0},
           b[] = {2.0, 3.0, 4.0};
    double c[3];
    //addition
    int i = 0;
    while(i<3){
        c[i] = a[i] + b[i];
        i = i + 1;
    }
    //display each element in the array c
    i = 0;
    while(i<3){
        printf("%f\n", c[i]);
        i = i + 1;
    }
}
```

Early Loop Exit

Using `break` statement to exit the loop immediately.

```
// finding the smallest number <= 1000000 that
// can be divided by 32 and 23.
int i = 1;
while(i <= 1000000){
    if(i%32 == 0 && i%23== 0){ // & is logical "AND"
        break;
    }
    i = i + 1;
}
printf("%d\n", i);
//displays "736"
```

Early Loop Exit

Without using `break`, you can write it like this

```
// finding the smallest number <= 1000000 that
// can be divided by 32 and 23.
//! means logical NOT
int i = 1;
while(i <= 1000000 && !(i%32 == 0 && i%23 == 0)){
    i = i + 1;
}
printf("%d\n", i);
//displays "736"
```

Early Loop Restart

Use the `continue` to restart the next loop immediately.

```
// finding the number of occurrences of 'l' in
// the word "hello". All string in C programming
// ends with '\0' implicitly.
int count = 0, i = 0;
char str[] = "hello";
while(str[i] != '\0'){ //! is logical NOT
    if(str[i] != 'l'){
        i = i + 1;
        continue;
    }
    count = count + 1; i = i + 1;
}

printf("%d\n", count);
//displays "2"
```

For Loop

- In previous examples, we all maintained a variable `i`, which increases by one at each iteration.
- The initialization of `i`, exit condition check and increment of `i` are all scattered in the code and are difficult to spot.
- for loop would gather these three pieces all in one place.

```
for(init; exit_condition_check; increment){
    ...
}
```

Vector Addition, Revisited 2

```
#include <stdio.h>
void main(){
    double a[] = {1.0, 2.0, 3.0},
           b[] = {2.0, 3.0, 4.0};
    double c[3];
    //addition
    for(int i = 0; i< 3; i=i+1){
        c[i] = a[i] + b[i];
    }
    //display each element in the array c
    for(int i = 0; i< 3; i=i+1){
        printf("%f\n", c[i]);
    }
}
```

- This code does the exactly the same thing as the previous `while` loop, but is arguably more compact.
- Notice `i` is only accessible inside each for loop!

Increment

- `i++` is a shorthand for `i = i + 1` .

However, the expression `i++` has a value `i` , but `i = i + 1` has a value `i+1` .

```c
#include <stdio.h>
void main(){
    int i = 1;
    printf("%d \n", i++); // prints 1
    i = 1;
    printf("%d \n", i=i+1); // prints 2
}
```

- Similarly, `i+=2` is a shorthand for `i = i + 2` .

Recursion or Loop?

- Recursion can also perform repeating operations as we have seen in previous lectures.
- However, when performing "counting operations" (i.e., counting elements in an array), loop is more often used.
- In some other operations, such as sorting and non-linear searching, recursion is a lot more natural.

Array as Input Argument

You can pass array as input variables of a function:

```
//compute dot product between a and b.
double dot(double a[], double b[]){
    double s = 0;
    for(int i = 0; i< 3; i++){
        s+ = a[i]*b[i];
    }
    return s;
}
```

If you specify the size of array,

```
double dot(double a[3], double b[3]){
    ...
}
```

The size will be ignored by the compiler.

Array as Input Argument

If I do not know the length of the array, what can I do?

- Pass another input argument, specifying the array length.

```
//compute dot product between a and b.
double dot(double a[], double b[], int len){
    double s = 0;
    for(int i = 0; i< len; i++){
        s+ = a[i]*b[i];
    }
    return s;
}
```

Pass by Value

When you pass an input argument to a function, you are passing by value: The program will copy the value to the input variable.

```
double square(double a){
    a = a*a; return a;
}
void main(){
    double n = 2; nn = square(n);
    printf("%f %f\n", nn, n);
    //display 4 2
}
```

The value of `n` is copied to the input argument `a`, thus operations on `a` has no effect on `n`.

Pass by Reference

- However, comparing to ordinary variables, the array occupies a much bigger memory space, thus pass by value can be expensive.
- In C, array is passed by reference.
    - If callee changes the array, caller's array will also be changed.

Pass by Reference, Example

```
//add all elements in an array by 1
void addone(double a[], int len){
    for(int i = 0; i< len; i++){
        a[i] += 1;
    }
}
void main(){
    double a[] = {1.0, 2.0};
    addone(a,2);
    printf("%f %f\n", a[0], a[1]);
    //display 2 3, NOT 1, 2!!
}
```

Return an Array

- Array cannot be returned by a function.
- However, since a function can make changes to caller's array, you can pass an array as input argument, and store results in that array.

```
//compute a+b and store the result in c
void add(double a[], double b[], double c[], int len){
    for(int i = 0; i< len; i++){
        c[i] = a[i] + b[i];
    }
}
void main(){
    double a[] = {1.0, 2.0}, b[] = {2.0, 3.0};
    double c[2];
    add(a,b,c,2);
    printf("%f %f\n", c[0], c[1]);
    //display 3 5
}
```

Multidim. Array

Matrix is a rectangle of numbers, arranged in rows and columns.

$$\boldsymbol{A} = \begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{bmatrix}.$$

We can use multidimensional array to store a matrix.

```
//an integer 2D array used to store a 3 by 3 matrix.
//The initialization is row-first.
int A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

Multidim. Array Example

We can use multidimensional array to store a matrix. The following function `trace` computes the trace of a matrix.

```
#include<stdio.h>
int trace(int nrow, int ncol, int A[nrow][ncol]){
    if (nrow != ncol) {
        printf("not a square matrix!\n");
        return 0;
    }
    int tr = 0;
    for(int i =0; i<nrow; i++){
        tr += A[i][i];
    }
    return tr;
}
void main(){
    int A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
    printf("%d\n", trace(3, 3, A));
}
```

Multidim. Array Example

- It is important to perform a sanity check at your code!
  - Trace operation is only defined on square matrices!
- The compiler does need to know all dimensions of the array!
  - If you want to use other arguments (`nrow` and `ncol` in the example above) to specify the dimensions, make sure they are declared **before** the array.
  - `int trace(int A[nrow][ncol],int nrow, int ncol)` will not work!
- Multidimensional array is also passed by reference.

String = Char Array

Read This Before Doing Homework!

C compiler treats strings (character sequence inside of double quotation) as `char` array.

- The compiler will treat `"hello"` as a `char` array `{'h','e','l','l','o','\0'}`.
- `'\0'` (ASCII code 0) is the end marker of a string and is automatically added by the compiler.

```
// display ascii code of string "hello"
#include <stdio.h>
void main(){
    char text[] = "hello";
    for(int i= 0; i<6; i++){
        printf("%d\n", text[i]);
    }
}
//display 104, 101, 108, 108, 111, 0
```

Homework: Counting Letter

Read the previous slide.

1. Write a function, `int count_letter(char text[],char letter)`. It returns the number of occurrences of `letter` in the string `text`. Use the following skeleton:

```
int count_letter(char text[],char letter){
    int i = 0;
    int count = 0;
    while(___){
        if(____){
            count++;
        }
        i = i + 1;
    }
    return count;
}
```

For example, `count_letter("hello",'l')` returns 2.

Homework: Length of a Vector

2. (submit) Write a function, `double calc_length(double vec[], int len)`. It returns the length of a vector.
    - Given a vector $a = [1, 2, 3]$, its length is computed via $\sqrt{1^2 + 2^2 + 3^2} = 3.741$.
    - In C programming language, you can calculate the square root of a number by calling the `sqrt` function.

Homework: Compare Vec

3. (submit) Write a function, `int compare_vec(int[] vec1, int[] vec2, int len)`. It outputs 0 if two vectors are exactly the same. Otherwise, it outputs an integer which is the difference between the first elements in these two arrays that differ.
    i. e.g., given `v1 = {0,1}, v2 = {0,1}`, `compare_vec(v1, v2)` outputs 0.
    ii. e.g., given `v1 = {0,1,1}, v2 = {0,1,2}`, `compare_vec(v1, v2)` outputs -1.
    iii. `vec1` and `vec2` has the same length.
    iv. Do you see any problem when converting this function for `double` vectors?

Homework: Flat Array

4. Write a function, that "flattens" a 2D array into 1D array. For example, 2D array `{{1,2},{3,4}}` is flattened into `{1,2,3,4}`.
    i. `void flatten(int nrow, int ncol, int a[nrow][ncol], int a_f[]);`
    ii. After the execution, `a_f` stores the flattened array.
5. Write a function that reads an element from the flattened array as if it is reading from the corresponding 2D array.
    i. `void get_elem(int a_f[], int i, int j)`
    ii. Given a 2D array `a`, and its flattened version `a_f`, `get_elem(a_f[], i, j)` gives you `a[i,j]`.