UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science & Applied Mathematics

Introduction to Algorithms & Programming

1 Functions

Functions are groups of statements that, taken together, perform a task. Breaking a program up into these modular blocks makes it easier to understand, maintain and test for errors. All C++ programs have at least one function (`main()`), and even the most trivial programs can define additional functions. The way code is divided into functions depends on the programmer and task at hand, but the most logical course of action is to ensure that each function performs only a single task.

1.1 Declaration

Declaring a function informs the compiler about the function's name, the data type it returns (if any), and the parameters it accepts (if any). A function must be declared before it is used in another part of the program. The general form of a function definition in C++ is as follows:

```
<return_type> <function_name>(<parameter_list>);
```

The above is known as a function prototype (or header) and consists of the following:

1.1.1 Return Type

A function may return a value. The return type specifies the data type of the value the function returns. However, if the function does not return a value, the return type is the keyword `void`.

1.1.2 Function Name

This is the actual name of the function. The choice of name is completely up to the programmer, but should explain what the function actually does. The naming rules for functions follow that of variables.

1.1.3 Parameter List

This represents the values passed to the function when it is used (*invoked*). These parameters are also known as *formal arguments*, and list the type, order and number of parameters of a

function. The parameter list, together with the function name, constitute the *function signature*. If you declare multiple functions, they must all have their own unique signature.[1] They must differ either by parameter list or function name (or both).

1.2 Definition

Declaring a function lists its name, inputs and outputs, but doesn't actually define what it does. A function definition in C++ specifies the function header, as well as its actual functionality. We may choose to first declare a function, and then define it, or we may simply define it immediately. If we define a function without declaring it, it must again happen before the function is actually used. Function definitions take the following form:

```
<return_type> <function_name>(<parameter_list>){
    <function_body>
}
```

where the only additional concept is the *function body*, which contains the collections of statements that describe what the function actually does.

The two examples below illustrate a function that is first declared and the later defined, and then the case where the function is simply defined.

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

/*
We tell the compiler the function header,
but we don't actually provide implementation
*/
int add(int, int);

int main(){
    int x = 1;
    int y = 2;
    cout << add(x,y) << endl;
    return 0;
}

/*
The definition of add() The header here must
match the declaration
*/
int add(int a, int b){
    return a + b;
}
```

[1] Strictly, this only applies to functions within the same *scope*.

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

/*
We define the function header and its implementation
all at once
*/
int add(int a, int b){
    return a + b;
}

int main(){
    int x = 1;
    int y = 2;
    cout << add(x,y)
         << endl;
    return 0;
}
```

1.3 Invoking Functions

In order to actually use a function, we need to *invoke* or call it. When a function is invoked, control is passed to the function which performs the defined task. When the function terminates, control returns back to the line where we first invoked the function. There are three ways a function can end:

- Running into its closing bracket (for `void` functions)
- Running into the statement `return;` (for `void` functions)
- Running into the statement `return <exp>;` (for non-`void` functions)

To invoke a function, we simply need to pass the required parameters (**of the correct type**) along with the function name. Additionally, if the function returns a value, we can store it in a variable. The following example illustrates this for a function that calculates the minimum of two numbers.

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


int min(int a, int b){
    if (a < b){
        return a;
    }
    //why don't I need an else here?
    return b;
}

int main(){
    int x, y;
    cin >> x >> y;
    int m = min(x,y);
    cout << "The min of " << x << " and " << y <<" is " << m << endl;
    m = min(x,2); //we can pass literals or variables
    cout << "The min of " << x << " and 2 is " << m << endl;
    return 0;
}
```

Question: what happens if I changed my variable `m` to `min`?

2 Function Arguments

If a function accepts parameters, these must be declared in the function header. These parameters behave like local variables created within the function body — they are created when the function is entered, and destroyed when the function exits. There are two ways to pass an argument to a function: by value or by reference.[2]

2.1 Pass-By-Value

By default, C++ passes arguments to a function by value. This means that the value of the argument is copied into the parameter of the function. In this case, any change to the parameter within the function has no effect on the original argument. In general, this means that the function cannot alter the arguments passed into the function.

The following is an example of a function that exchanges the values of two variables:

[2] If you're convinced there are more than two, come debate it with me!

```cpp
#include <iostream>

using namespace std;

void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int a = 100;
       int b = 200;

       cout << "Before swap, value of a :" << a << endl;
       cout << "Before swap, value of b :" << b << endl;

       // calling a function to swap the values.
       swap(a, b);

       cout << "After swap, value of a :" << a << endl;
       cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

When run, the above code produces the following output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

2.2 Pass-By-Reference

This method sends the original argument to the function. Any change made to the parameter (which now refers to the actual argument) will affect the variable that was passed to the function.

The following is the call-by-reference version of the above code:

```cpp
#include <iostream>

using namespace std;

void swap(int &x, int &y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int a = 100;
       int b = 200;

       cout << "Before swap, value of a :" << a << endl;
       cout << "Before swap, value of b :" << b << endl;

       // calling a function to swap the values.
       swap(a, b);

       cout << "After swap, value of a :" << a << endl;
       cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

When run, the above code produces the following output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

2.3 Default Arguments

In C++, functions can also have optional parameters. If we do not provide an argument, the parameter will take on a specified default value. We could therefore have a function with three parameters which we could call with only two. Default values can be specified for the each of the last parameters (we cannot have a situation, for instance, where the first parameter has a default value, while the second does not). If we provide an argument to a parameter with a default value, the default value is ignored.

The following code illustrates the concept of default arguments:

```cpp
#include <iostream>

using namespace std;

int mult(int x, int y = 2){
    return x * y;
}

int main(){
    cout << mult(10) << endl; //equivalent to mult(10, 2) = 20
    cout << mult(10, 4) << endl; //produces 40
}
```

Submission 1: Upper Case

A common-enough task is to convert all characters in a string to upper case. Download the source file provided on Moodle and complete the missing parts of it. Your completed program should take in a number of strings and convert them to their upper case variants. **Note that for this task, a presentation error will receive a mark of 0%!**

Input

The first line of input is a single number N, which specifies how many strings will be input to your program. N lines of input follow, with a single string on each line. You may assume the strings do not contain spaces.

Output

For each string received, output the result when it is converted to upper case.

Example Input-Output Pairs

Sample Input #1

```
2
Hello
Goodbye
```

Sample Output #1

```
HELLO
GOODBYE
```

Sample Input #2

```
3
Hello.
How?
fine.
```

Sample Output #2

```
HELLO.
HOW?
FINE.
```

Sample Input #3

```
1
123456789
```

Sample Output #3

```
123456789
```

Submission 2: Twin Primes

A prime number is a number that has no positive divisors, aside from 1 and itself. A *twin prime* is a prime number that is either 2 less or 2 more than another prime. The *Twin Prime Conjecture*, attributed by some to Euclid (circa 300 BC), proposes that there are an infinite number of twin primes. This is still an open problem in Number Theory, and proving (or disproving) it will earn you $1 000 000!
Download the source file provided on Moodle and complete the missing parts of it. Your completed program should take in a number of integers and determine whether they are twin primes.

Hint

If you receive a number x, you will have to check first if x is prime, and then whether $x - 2$ or $x + 2$ is prime.

Input

The first line of input is a single number N, which specifies how many positive integers will be input to your program. N lines of input follow, with a single positive integer on each line. You can assume each integer is greater than 2.

Output

For each number, output `true` if it is a twin prime. Otherwise, output `false`.

Example Input-Output Pairs

Sample Input #1

```
2
4
9
```

Sample Output #1

```
false
false
```

Sample Input #2

```
1
11
```

Sample Output #2

```
true
```

Sample Input #3

```
2
29
27
```

Sample Output #3

```
true
false
```

3 Arrays

An array is a series of elements of the same type placed in contiguous memory locations. Each element can be individually referenced by considering its index in the array.

That means that, for example, five values of type `int` can be declared as an array without having to declare five different variables (each with its own identifier). Instead, using an array, the five `int` values are stored in contiguous memory locations, and all five can be accessed using the same identifier (the array), with the proper index.

In some ways, this is very similar to a Python list, but there are a few key differences. In a Python list, items can be added or removed at will, changing the size of the list during runtime. Additionally, they can store values of any type. In C++, arrays can only store values of a particular fixed type, and the size of the array is set at compile time (i.e. before the program is executed) and cannot be changed. This makes arrays in C++ more efficient in terms of memory access and performance, but also less flexible. We will later cover the concept of a vector, which *does* allow for the size to be changed during execution.

In order to use an array, we must include the `<array>` header. Furthermore, we will have to compile our code with C++11 features activated. This can be done using the `-std=c++11` flag:

```
g++ -std=c++11 file.cpp -o file
```

However, modern compilers are typically set to use newer standards by default (such as C++14, C++17, or later), so in most cases, it is not necessary to explicitly include the C++11 flag to use arrays or other C++11 features.

To create an array called `arr` containing 5 string values, we would write:

```
1 array<string, 5> arr;
```

The size of the array must be a **constant expression**. That means that, before the program is run, the compiler should be able to determine the size of the array. Regular variables are not constant expressions! We can access the size of an array using the `size()` function, e.g.

```
1 array<string, 5> arr;
2 cout << arr.size(); //prints 5
```

3.1 Initialisation

We can set the values of each element in the array directly using either initialiser lists, or uniform initialisation. If we do not initialise the elements, their values are *indeterminate* and could therefore be anything! The following illustrates how to initialise the values of an array directly:

```
array<int, 5> myarray = { 9, 7, 5, 3, 1 }; // initialization list
array<int, 5> myarray2 { 9, 7, 5, 3, 1 }; // uniform initialization
```

Note that if less than 5 values on the right-hand side are provided, the remaining elements will be set to zero. Providing more than 5 values is illegal — you will not be able to compile your code.

3.2 Indexing

Accessing individual elements in the array is done using the subscript operator. Note that the subscript (or index) starts from 0 and runs up to one less than the length of the array.

The subscript operator does not do any bounds-checking. If an invalid index is provided, bad things will probably happen.

```
cout << myarray[1]; //get and display the 2nd element
myarray[2] = 6; //modify the third element
```

3.3 Iterating

There are primarily two ways of iterating over each element of an array. We could use a `for` loop, with a counter that starts at 0 and runs over the whole array, as in the following example:

```
array<int, 5> myarray = { 9, 7, 5, 3, 1 };
for (int i =0; i < myarray.size(); ++i){
    myarray[i] = myarray[i] + i;
    cout << myarray[i] << endl;
}
```

The advantage of this approach is that, within our loop, we have access to current index, which we can then use to get the element at that position. However, we must be careful that our loop terminates appropriately, so that we do not try access or modify an invalid element.

If we do not need the index, a safer way of iterating is to use the **enhanced `for` loop**, a feature also introduced in C++11. For example:

```
array<int, 5> myarray = { 9, 7, 5, 3, 1 };
for (int e : myarray){
    cout << e << endl;
}
```

Here we say "for each `int e` in the array, print it out". The advantage of this is that we no longer need to worry about having to check bounds; however, we lose access to the index.

As a final point, we could also use the enhanced loop in conjunction with references. This would allow us to change each element as we loop over them:

```
array<int, 5> myarray = { 9, 7, 5, 3, 1 };
for (int &e : myarray){
    e = 0;
}
//after the loop, all the elements of the array are zero!
```

Submission 3: Pass

The department wishes to calculate how many people have passed a subject based on their mark. However, this is made slightly trickier because politicians keep changing the pass mark! Write a program that accepts 10 integers and the pass mark, and determines how many people have passed.

Input

Input consists of 10 integers, all on the same line. Each integer represents a student's mark for a course. The next line contains a single integer, which represents the pass mark. You can assume that all integers are in the range $[0, 100]$.

Output

Output the number of students who passed. Students only fail if they do not attain *at least* the pass mark.

Example Input-Output Pairs

Sample Input #1

```
10 20 30 40 51 60 65 27 19 100
50
```

Sample Output #1

```
4
```

Sample Input #2

```
0 0 0 0 0 0 0 0 0 0
10
```

Sample Output #2

```
0
```

4 Vectors

Like arrays, a vector represents a series of elements of the same type placed in contiguous memory locations. However, vectors also provide additional flexibility in that they are able to dynamically resize themselves. We therefore do not need to know beforehand how many elements will be stored in the vector — we can simply add them to the vector as necessary, and the vector will take care of allocating more space for them. In practice, there is often no need to use arrays at all — vectors give us everything arrays can, and more!

4.1 Construction

There are a few options when it comes to creating vectors. To begin, we must first include the `<vector>` header. The following examples create a vector of type integer, but the same applies to vectors of any type.

Example	Comments
`vector<int> vec;`	Creates an empty (size 0) vector
`vector<int> vec(4);`	Creates a vector with 4 elements. Each element is initialised to zero. If this were a vector of strings, each string would be empty.
`vector<int> vec(4, 42);`	Creates a vector with 4 elements. Each element is initialised to 42.
`vector<int> vec(4, 42);` `vector<int> vec2(vec);`	The second line creates a new vector, copying each element from the `vec` into `vec2`.

4.2 Initialisation

Like arrays, we can also provide the vector with initial values. We can do this using either initialiser lists, or uniform initialisation. The following illustrates how to initialise the values of a vector directly:

```
vector<int> myvector = { 9, 7, 5, 3, 1 }; // initialization list
vector<int> myvector2 { 9, 7, 5, 3, 1 }; // uniform initialization
```

4.3 Iteration and Indexing

Indexing and iteration function exactly the same for vectors as they do for arrays. We repeat what was said in the previous lab here for ease of reference.

4.3.1 Indexing

Accessing individual elements in the vector is done using the subscript operator. Note that the subscript (or index) starts from 0 and runs up to one less than the length of the vector.

```
cout << myvector[1]; //get and display the 2nd element
myvector[2] = 6; //modify the third element
```

The subscript operator does not do any bounds-checking. If an invalid index is provided, bad things will probably happen. The `at()` function, available to both arrays and vectors, can be used instead to access an element in a safe way, by first checking that the index provided is a valid one.

4.3.2 Iterating

There are primarily two ways of iterating over each element of a vector. We could use a `for` loop, with a counter that starts at 0 and runs over the whole vector, as in the following example:

```
vector<int> myvector = { 9, 7, 5, 3, 1 };
for (int i = 0; i < myvector.size(); ++i){
    myvector[i] = myvector[i] + i;
    cout << myvector[i] << endl;
}
```

The advantage of this approach is that, within our loop, we have access to current index, which we can then use to get the element at that position. However, we must be careful that our loop terminates appropriately, so that we do not try access or modify an invalid element.

If we do not need the index, a safer way of iterating is to use the **enhanced `for` loop**, a feature introduced in C++11. For example:

```
array<int> myvector = { 9, 7, 5, 3, 1 };
for (int e : myvector){
    cout << e << endl;
}
```

Here we say "for each int `e` in the vector, print it out". The advantage of this is that we no longer need to worry about having to check bounds; however, we lose access to the index.

As a final point, we could also use the enhanced loop in conjunction with references. This would allow us to change each element as we loop over them:

```
vector<int> myvector = { 9, 7, 5, 3, 1 };
for (int &e : myvector){
    e = 0;
}
//after the loop, all the elements of the vector are zero!
```

Submission 4: Percentage

Given a list of **non-negative** numbers, work out what percentage each number makes up of the total list. Formally, assume we are given a list of numbers $x_1, x_2, \ldots x_N$. For each element x_j, calculate

$$\frac{x_j}{\sum_{i=1}^{N} x_i}$$

Input

The first line of input is an integer N. N non-negative integers follow, each on their own line.

Output

On their own line, output the percentage each number represents.

Example Input-Output Pairs

Sample Input #1

```
2
1
3
```

Sample Output #1

```
0.25
0.75
```

Sample Input #2

```
4
10
10
10
10
```

Sample Output #2

```
0.25
0.25
0.25
0.25
```

4.4 Member Functions

Vectors have additional functionality above and beyond what arrays possess. These functions most commonly deal with operations that require some kind of resizing — something arrays are not capable of. A full list of functions can be found here `http://en.cppreference.com/w/cpp/container/vector`. We list some of the more useful functions below:

Example	Comment

`vector<int> vec;`
`int x;`
`cin >> x;`
`vec.push_back(x);`
	Add an element to the end of the vector

`//20 elements of value 2`
`vector<int> vec(20, 2);`
`vec.clear();`
	Clears the contents of the vector. It now contains no elements.

`//20 elements of value 2`
`vector<int> vec(20, 2);`
`vec.resize(2);`
	The vector now contains the first two elements of the vector before we resized it.

`//20 elements of value 2`
`vector<int> vec(20, 2);`
`vec.pop_back();`
	The opposite of `push_back()`: removes the last element of the vector

4.5 Additional Functions

In class, we discussed some of the additional functions provided by the `algorithm` header file. As we saw, these functions do not accept the object itself, but rather the `begin()` and `end()` markers (known formally as *iterators*) of the object. Thus as long as as whatever data structure we're working with (whether it be arrays, vectors, or some other thing) provides a `begin()` and `end()` function, they can be used with the `algorithm` header.

For instance, to sort a vector and array, we can simply do the following:

```
array<int, 4> arr = {2,1,3,4};
sort(arr.begin(), arr.end());

vector<int> vec = {2,1,3,4};
sort(vec.begin(), vec.end());
```

This illustrates that the sort function is independent of the data type its sorting. It does not care whether its an array or vector — from its point of view, they're the same thing.

Question: how do we sort an array or vector in descending order?

A full list of functions is available here `en.cppreference.com/w/cpp/header/algorithm`.

Submission 5: Hi-Lo

Given a list of numbers, determine the second largest **and** second smallest element.
Hint: one of the functions provided by `algorithm` might make this task much easier.

Input

The first line of input is a single integer $N \geq 2$. N real numbers follow, one per line.

Output

Output the second smallest and second largest value on their own lines.

Example Input-Output Pairs

Sample Input #1

```
4
1.2
2
3
4
```

Sample Output #1

```
2
3
```

Sample Input #2

```
3
-1
0
1
```

Sample Output #2

```
0
0
```

Submission 6: Dynamic Matrix

In class, we looked at 2 dimensional arrays: how to create them and how to loop over them. Refer back to the lecture slides if you unsure of how to do so.
Like arrays, we can also have 2 dimensional vectors. This task requires you to create a 2D vector, representing a matrix initially populated with zeroes. Based on the user input, you will then need to change the value of some of these elements.

Input

The first line of input consists of two integers: the number of rows and columns of the matrix. The next line contains a single integer N, which specifies the number of lines to follow. Each of the following N lines contains 3 integers in the format `<row> <column> <value>`. Your program should set the element at this specified row and column to the given value. Note that the row and column are zero-indexed.

Output

Display the resulting matrix in a grid, with the appropriate elements altered according to the input received.

Example Input-Output Pairs

Sample Input #1

```
2 2
2
0 0 12
1 0 -1
```

Sample Output #1

```
12 0
-1 0
```

Sample Input #2

```
3 3
1
1 1 1
```

Sample Output #2

```
0 0 0
0 1 0
0 0 0
```