# Tutorial #05

**Tasks set in tutorials should be finished in your own time!**

## PRE-CONDITIONS AND POST-CONDITIONS

Stating these conditions should be the first step in designing and writing your algorithms. They provide a contract between the programmer who writes the procedure/function and the (client) programmer who uses it.

* Pre-condition - a statement indicating any condition(s) that should be true when the function is called. The function is not guaranteed to perform as it should unless the pre-conditions have been met.
* Post-condition – a statement describing the condition that will be true when the function has completed its task. If the pre-condition(s) met (and the function is correct), then the post-condition is guaranteed to be true.

The client programmer must ensure that the pre-condition is met when the procedure is called; the function programmer must guarantee that, assuming the precondition has been met, the post-condition will be satisfied when the procedure ends.

1. Look at the following functions and complete the information provided in the comments given for each function, i.e., describe its purpose, identify the data items it uses (**IN**), produce (**OUT**) or modify (**IN/OUT**) and specify its pre- and post-conditions.

**/\*\***

**\* convert temperature from Celsius to Fahrenheit**

**\* (assuming temperature is above absolute zero)**

**\* IN: tC (double)**

**\* OUT: tF (double)**

**\* pre-condition:**

**\* post-condition:**

**\*/**

**void convertCelsiusToFahrenheit(double tC, double& tF)**

**{**

**tF = (((tC \* 9) /5) + 32);**

**}**

**/\*\***

**\***

**\* IN:**

**\* OUT:**

**\* pre-condition:**

**\* post-condition: result is true when (min <= x) && (x <= max),**

**\* false otherwise**

**\*/**

**bool inRange(int x, int min, int max)**

**{**

**return (min <= x) && (x <= max);**

**}**

**/\*\***

**\***

**\* IN:**

**\* OUT:**

**\* pre-condition:**

**\* post-condition: number = (d1 \* 100) + (d2 \* 10) + d3**

**\*/**

**void isolateDigits(int number, int& d1, int& d2, int& d3)**

**{**

**d1 = number / 100;**

**d2 = (number / 10) % 10;**

**d3 = number % 10;**

**}**

**/\*\***

**\* Take a single digit as a char and transform it as a numerical digit**

**\* (e.g., '5' as 5)**

**\* IN: letter (char)**

**\* OUT:**

**\* pre-condition:**

**\* post-condition:**

**\*/**

**void convertLetterToDigit(char letter, int& digit)**

**{**

**digit = letter - '0';**

**}**

**/\*\***

**\***

**\***

**\* pre-condition:**

**\* post-condition: c changed to uppercase if was a lowercase letter,**

**\* otherwise remains the same**

**\*/**

**void convertLetterToUpper(char& c)**

**{**

**if (('a' <= c) && (c <= 'z'))**

**c -= 'a' - 'A';**

**}**

**/\*\***

**\* Add one second to a time (in 24 hour format)**

**\***

**\* pre-condition:**

**\* post-condition:**

**\*/**

**void tick(int& h, int& m, int& s)**

**{**

**long long secs((h\*3600) + (m \* 60) + s);**

**++secs;**

**h = (secs / 3600) % 24;**

**m = (secs % 3600) / 60;**

**s = secs % 60;**

**}**

**/\*\***

**\***

**\***

**\* pre-condition:**

**\* post-condition: result is 1, 2 or 3**

**\*/**

**char getOption()**

**{**

**cout << "\nEnter your option (1, 2 or 3): ";**

**cin >> opt;**

**while ((opt != 1) && (opt != 2) && (opt != 3))**

**{**

**cout << "\nERROR: Invalid option. Try again: ";**

**cin >> opt;**

**}**

**return opt;**

**}**

1. Look at the following (incomplete) program which calculates and displays the square root of a number given by the user:

**#include <iostream> //for cin >> and cout <<**

**using namespace std;**

**double x, estimatedSqrt;**

**int main()**

**{**

**void calculateEstimatedSquareRoot();**

**void getNumber();**

**cout << "\nTo estimate the square root of a number...\n\n";**

**char answer;**

**do {**

**getNumber();**

**calculateEstimatedSquareRoot();**

**cout << "\nSquare root of " << x << " is " << estimatedSqrt;**

**cout << "\nDo you want to calculate another one? (Y/N)";**

**cin >> answer;**

**} while (tolower(answer) == 'y');**

**cout << "\n\n";**

**system("pause"); //to hold the output screen**

**return(0);**

**}  
void getNumber()**

**{**

**cout <<"\nEnter a floating point value: ";**

**cin >> x;**

**}**

**void calculateEstimatedSquareRoot() {**

**//Stub function**

**cout << "\n Stub function: ";**

**cout << "When implemented, it will estimate sqrt(" << x << ")...";**

**cout << "\nFor now produces 0";**

**estimatedSqrt = 0;**

**}**

1. Remove any global variables, using parameters and return value where appropriate, to communicate data across functions.
2. Write the C++ code for the **calculateEstimatedSquareRoot** function following the method described below:

* *Start with the following initial estimates:*

*high guess = number*

*low guess = 0*

* *Calculate the average of the high and low guesses.*
* *If this average squared is greater than the number, make the high guess equal to the average. If, on the contrary, this average squared is less than the number, make the low guess equal to the average.*
* *Repeat the refinement process using the new average of the new guesses ... until the result is accurate enough.*

*NOTE: Make sure you understand this method before trying to develop an algorithm for it and try to think about how to decide if 'the result is accurate enough'.*

1. Identify the pre and post condition for the two functions used in this program.
2. Add **assert** to enforce any precondition needed in your program.
3. Modify the client code (**main** function here) to ensure that this program will use the **calculateEstimatedSquareRoot** function safely.
4. ***(In your own time)*** There is better solution. Look up the *Newton-Raphson approximation* and try to implement it.
5. Find out which function provided by C++ standard library **<cmath>** can be used to calculate square root. Check its precondition and post condition.

## ADDITIONAL FLOW OF CONTROL CONSTRUCTS

Other statements are available in C++ that you need to be aware of. In particular, look at the **break**, **continue**, **return**, **exit** and **goto** statements.

* We have already seen how the **break** statement is used inside the **switch** statement to create a true multi-branching selection statement and how to use the **return** statement at the end of a function to pass the control back to the calling function. However, all of these statements are to be used with great caution as they have the capacity to break the controlled flow of code execution provided in structured programming. See <http://www.learncpp.com/cpp-tutorial/58-break-and-continue/>.
* The **exit** statement should be limited to handling of errors. See <http://www.learncpp.com/cpp-tutorial/712-handling-errors-assert-cerr-exit-and-exceptions/>.
* Also check the **goto** statement. This one is to be avoided at all cost. See <http://www.learncpp.com/cpp-tutorial/54-goto-statements/>.

1. In the following piece of code, the **break** statement is used to stop the loop if a negative number is entered.

**int number, sum(0):**

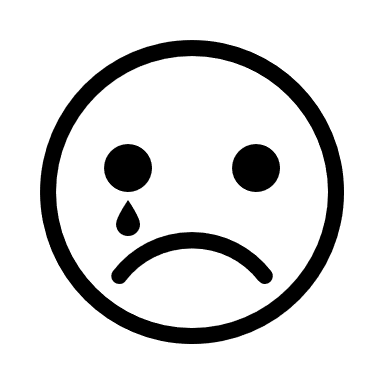
**for (int i(0); i < 10; ++i)**

**{**

**cin >> number;**

Use a while

loop



**if (number < 0)**

**break;**

**sum += number;**

**}**

1. Try to draw the flowchart for this program!
2. Rewrite this program without the **break** statement.
3. In the following piece of code, the **continue** statement is used to skip any negative number entered by the user.

**int number, sum(0):**

**for (int i(0); i < 10; ++i)**

Shuffle the if condition



**{**

**cin >> number;**

**if (number < 0)**

**continue;**

**sum += number;**

**}**

1. Try to draw the flowchart for this program!
2. Rewrite this program without the **continue** statement.
3. In the following piece of code, the **return** statement is used to terminate the function when the user enters option 0.

**void runGameApplication() {**

Shuffle the if condition



**int option:**

**displayMenu();**

**cin >> option;**

**if (option == 0)**

**return;**

**const int score = playGame();**

**displayResults(score);**

GoTo?

Just no!



**return;**

**}**

1. Try to draw the flowchart for this program!
2. Rewrite this program without the **return** statement.

## C++ OPERATORS, enum and typedef

1. C++ provides various facilities for programmers to 'create' their own data types, including using **enum** and **typedef**, structures and classes (next year).  
   Make sure to read up on these… They will become useful!

* The **enum** keyword can be used to create an **enumerated type** (also called an enumeration) which is a data type where every possible value is defined as a symbolic constant (called an enumerator). See <http://www.cplusplus.com/doc/tutorial/other_data_types/> or <http://www.learncpp.com/cpp-tutorial/45-enumerated-types/>.
* C++11 introduced a more elegant enum facility with “enum classes”, prefer these to old style enums when you aren’t planning to use the enum as a placeholder for constant numbers. <https://www.cprogramming.com/c++11/c++11-nullptr-strongly-typed-enum-class.html>
* The **typedef** keyword can be used to create a **type alias**. See <http://www.cplusplus.com/doc/tutorial/other_data_types/> or <http://www.learncpp.com/cpp-tutorial/46-typedefs/>.

1. Write a program that defines an enumerated type to represent the various colours of traffic lights and displays a random traffic light colour. Implement with enum and then enum classes, which do you prefer?
2. Given the following function prototype:

**int printData();**

Convert the **int** return value to a **typedef** named **error\_t** using the **typedef** keyword. Include both the **typedef** statement and the updated function prototype.

1. Write a program to play a number guessing game (i.e., the user enters a guess, you print YES or TOO HIGH or TOO LOW) until the number has been identified.
2. allow the user to specify a given range.
3. keep track of the number of guesses and display a message telling the user how many guesses they used up.
4. stop after 5 guesses if the number has not been reached and display an appropriate message.

# one-dimensional (1D) ARRAYS

## QUESTIONS

Arrays are homogenous complex data structures used to store a finite collection of data items of the same type. They offer a direct, fast and efficient, way of accessing data in memory and work well when it is meaningful and useful to access items knowing their position in the array.

**Declaring an array**The declaration of an array gives the type of its items (base type), its name and size.  
e.g., to create an (undefined) array of 3 integers:

**const int SIZE(3);**

**int tab[SIZE];**

The first item will be in position 0 and is referred to as **tab[0]**, the second in position **1** is **tab[1]**, and the last item, in position **SIZE-1**, is **tab[SIZE-1]**.   
The array’s size must be an integer constant because the array is created at compile time and the compiler needs to know the size of memory required (i.e., here **SIZE \* sizeof(int)**)

**Initialising an array**

The values in the array can be initialised using an initialiser list.  
e.g., to create an array of 3 integers and initialise its items to 10, 20 and 30 respectively:  
 **int tab[SIZE] = {10, 20, 30};**

**Accessing items in an array**  
Each individual item in the array can be accessed by indicating the name of the array followed by its position in the array, aka subscript or index, between **[]**. The subscript can be a variable, but it must have an integer value (or one that can be converted to an integer).  
e.g., to read in a value in last position of the array:  
 **cin >> tab[SIZE-1];**

e.g., to double up the value of the first item in the array:  
 **tab[0] = tab[0] \* 2;**

e.g., to add up all items in the array:

**int sum(0);  
 for (int i(0); i < SIZE; ++i)**

**sum += tab[i];**

**Manipulating the contents of arrays**The name of the array does not give direct access to all items in the array (more on this later). So there are operations, such as input, output, assignment and comparison, which must be done item per item.  
e.g., **cin >> tab** is illegal - to read in data into an array\*:

**for (int i(0); i < SIZE; ++i)**

**cin >> tab1[i];**

e.g., **cout << tab** is legal but does not produce excepted result - to display the content of an array\*:

**for (int i(0); i < SIZE; ++i)**

**cout << tab1[i];**

e.g., **tab1 == tab2** is legal but does not produce excepted result - to compare the content of two arrays (of same size)\*:

**bool same(true);  
 for (int i(0); i < SIZE; ++i)**

**if (tab1[i] != tab2[i])**

**same = false;**

e.g., **tab1 = tab2** is illegal - to assign the content of one array into another (of same size)\*:

**for (int i(0); i < SIZE; ++i)**

**tab1[i] = tab2[i]);**

*\*These examples assume, respectively, that the operators* ***>>****,* ***<<****,* ***=*** *and* ***==*** *are available on the type of items stored in these arrays.*

**Passing an array as a parameter to a function**  
Passing an array as a parameter to a function is different than passing a primitive type (such as **int**, **float**, **double**, **char** or **bool**). The formal parameter (in the function declaration and definition heading) shows the base type of the array and the square brackets **[]** - the size is omitted (*more on this later*). That’s all that is needed to pass the array as an **IN/OUT** or **OUT** parameter. However, when the array is to be passed as an **IN** parameter, the **const** keyword should be added in the front of the array parameter declaration to protect its content inside the function body.  
e.g., to add up all items in the array:

**int addUpItems(const int t[]) { //IN parameter  
 double sum(0.0);  
 for (int i(0); i < SIZE; ++i) //assume SIZE is a global const**

**sum += t[i];  
 return sum;**

**}**

e.g., to reset all items in the array:

**void resetAllItems(int t[]) { //OUT parameter  
 for (int i(0); i < SIZE; ++i)**

**t[i] = 0;  
 }**

e.g., to change item at given position in the array to given value:

**void changeItem(int pos, double value, int t[]) { //IN/OUT parameter  
 t[pos] = value;  
 }**

**const int SIZE(3); //maximum size of array**

**int main() {**

**int addUpItems(const int t[]) //prototype  
 void resetAllItems(int t[]); //prototype**

**void changeItem(int pos, double value, int t[]); //prototype**

**double tab[SIZE] = {10.0, 20.0, 30.0}; //create array**

**cout << "Sum is: " << addUpItems(tab); //call  
 resetAllItems(tab); //call**

**changeItem(2, 12.5, tab); //call**

**//...**

**return 0;**

**}**

1. Look at the following C++ code extract:

**int scores[5]; //line 1**

**for (int i(0); i < 5; ++i) //line 2**

**scores[i] = i \* 10; //line 3**

1. Which type of data structure is declared in line 1? How many elements can it contain? Which type?
2. Describe what happens in lines 2 and 3?
3. After line 3 what are the type and value(s) of **scores[0]**, **scores[3]**, **scores[5]** and **scores**?
4. Write a C++ code extract that displays the current content of **scores**.
5. Modify the code given on line 1 so that it initialises the array to the following 5 values: 1, 4, 5, -2, 0.
6. Modify the code given so that it fills in the **scores** array with 5 random values between 1 and 6 (represent dice values).
7. Write a C++ code extract that displays the highest and lowest values stored in **scores**.
8. Modify the code you have written so far so that the size of the array is stored in a global constant called **SIZE** which represents the number of dice values the array can hold. Explain the advantages of doing so.
9. Place your code into functions, each of them handling a separate tasks, so that they could be called as follows from the **main** function:

**setRandomScores(scores);**

**displayScores(scores);**

**cout << "\nThe highest value is: " << highestScore(scores);**

**cout << "\nThe lowest value is: " << lowestScore(scores);**

1. Look at the following code that has been written to display a message indicating whether a value entered by the user is or not present in the array. This code does not work. Explain why and produce a working, clear and efficient solution to that problem.

**int value;**

**cout << "\nEnter the value you are looking for: ";**

**cin >> value;**

**for (int i(0); i <= SIZE; ++i)**

**if (scores[i] = value)**

**cout << "\n" << value << " is present!";**

**else**

**cout << "\n" << value << " is NOT present!";**

Then place your code in a function called **isPresent** that can be used as follows:

**int value;**

**cout << "\nEnter the value you are looking for: ";**

**cin >> value;**

**if (isPresent(value, scores))**

**cout << "\n" << value << " is present!";**

**else**

**cout << "\n" << value << " is NOT present!";**

1. ***(In your own time)*** Write the code for a function called **position** that asks for and reads in a value representing a valid dice from the user then returns the first position of the dice value given in **scores**. If the value is not there it returns -1. It can be called as follows:

**cout << "\nEnter the dice value you are looking for: ";**

**const int value(getValidDice());**

**const int pos(position(value, scores));**

**if (pos != -1)**

**cout << "\nThe dice " << value << " is at position " << pos;**

**else**

**cout << "\nThe dice " << value << " is NOT there!";**

1. ***(In your own time)*** Write a C++ function called **sortScores** that rearranges all score values in the array in ascending order and could be called as follows from the **main** function:

**setRandomScores(scores);**

**cout << "Before sorting...";**

**displayScores(scores);**

**sortScores(scores);**

**cout << "After sorting...";**

**displayScores(scores);**

Classical sorting algorithms (bubble sort, selection sort, insertion sort, shell sort, merge sort and quick sort) are described on <https://www.khanacademy.org/computing/computer-science/algorithms>.

More on this later!

# Two-dimensional (2D) ARRAYS

## QUESTIONS

A two-dimensional array can be thought as a one-dimensional array of arrays. The notation for multidimensional arrays extends that used for simple arrays.

**Declaration**The declaration gives the type of the items (base type), the name and size of the array.  
e.g., to create an (undefined) array of 3 integer sub-arrays, each containing 2 values:

**const int DIM1(3);**

**const int DIM2(2);**

**int tab[DIM1][DIM2];**

The first item in sub-array **tab[0]** is **tab[0][0]**, the second is **tab[0][1]**, and the last item, which will be in sub-array **tab[2]** is **tab[DIM1-1][DIM2-1]**.   
The array dimensions must be constant integer expressions as the array is created at compile time and the compiler needs to know the size of memory required (i.e., here **DIM1 \* DIM1 \* sizeof(int)**)

**Initialisation**

The values in the array can be initialised using an initialiser list.  
e.g., to create an integer 2D array of 3 by 2 integers (three sub-arrays) and initialise its items to 10, 20, 30, 40, 50 and 60 respectively:  
 **int tab[DIM1][DIM2] = { {10, 20}, {30, 40}, {50, 60}};**

**Access to various items in a 2D array**  
The individual items in the array can be accessed by indicating the name of the array followed by its positions, aka subscripts or indices, in the array between **[]**. The subscripts can be any valid integer expression (or one that can be converted to an integer).  
e.g., to read in a value in last position of the array:  
 **cin >> tab[DIM1-1][DIM2-1];**

e.g., to double up the value of the first item in the array:  
 **tab[0][0] = tab[0][0] \* 2;**

e.g., to add up all items in the array:  
 **int sum(0);  
 for (int i(0); i < DIM1; ++i)  
 for (int j(0); j < DIM2; ++j)  
 sum += tab[i][j];**

**Manipulating contents of 2D arrays**As for 1D arrays, there are operations, such as insertion (**>>**), extraction (**<<**), comparison (**==**) and assignment (**=**), which must be performed item per item.

**Pass a 2D array as a parameter to a function**  
To pass a 2D array as a parameter to a function the array declaration must indicate the size of the second dimension (more on this later).  
e.g., to add up all items in the array:  
 **int addUp(const int t[][DIM2]) //as IN parameter  
 int sum(0);  
 for (int i(0); i < DIM1; ++i) //assume DIM1 is global const  
 for (int j(0); j < DIM2; ++j) //assume DIM2 is global const**

**sum += t[i][j];  
 return sum;**

**}**

1. Look at the following C++ code extract:

**const int students(5); //line 1**

**const int modules(2); //line 2**

**double marks[students][modules]; //line 3**

**double value(1.0f); //line 4**

**for (int studno(0); studno < students; ++studno) //line 5**

**for (int modno(0); modno < modules; ++modno) { //line 6**

**marks[studno][modno] = value; //line 7**

**value \*= 2; //line 8**

**}**

1. Which type of data structure is declared in line 3? How many elements can it contain? Which type?
2. Describe what happens in lines 5 to 7?
3. After line 8 what are the type and value of **marks[0][0]**, **marks[1][1]**, **marks[5][1]**, **marks[1]** and **marks**?
4. Write the code to display all the values stored in **marks**?
5. Use an initialiser to set all the values in **marks** to **0.0**.
6. Modify the code to put 10 random values between **0.0** and **100.0** in **marks**.
7. Keeping **students** and **modules** as global constants, modify the code to create functions that handle these separate tasks and can be called as follows from the **main** function:

**setRandomMarks(marks);**

**displayMarks(marks);**

1. Write the definition of a function called **getValidMark** that reads in a given (valid) mark. If the mark is invalid it displays an error message and asks for another one until a valid value has been entered. The valid mark is then returned and could be used as follows inside the function **recordMarks** to read in all the marks in the array **marks**.

**void recordMarks(double m[][modules]) {**

**double getValidMark();**

**cout << "\nEnter marks...\n";**

**//for each student**

**for (int studno(0); studno < students; ++studno)**

**{**

**//for each mark for that student**

**for (int modno(0); modno < modules; ++modno)**

**{**

**//ask for a mark for that student and that module**

**cout << "student " << studno << " module " << modno << ": " ;**

**//record the (valid) value given by the user**

**m[studno][modno] = getValidMark();**

**}**

**}**

**}**

Test these functions in the **main** part of the program.

1. Write the code for a function called **isPresent** that asks for and reads in a value representing a valid mark from the user then finds whether it is stored in **marks**. It can be called as follows:

**cout << "\nEnter the mark you are looking for: ";**

**const double value(getValidMark());**

**if (isPresent(value, marks))**

**cout << "\nThe mark " << value << " is in the array!";**

**else**

**cout << "\nThe mark " << value << " is NOT in the array!";**

1. Modify the program so that is can store the marks for 20 students in 4 modules. What do you need to change, when?

|  |
| --- |
| 1. **ASSESSED HOMEWORK** Write a program that displays a square grid (size 10 by 10) on the screen surrounded by a simple frame of stars (**'\*'**) and showing a little character (let’s call it Blob: **'@'**) in the middle. The player can control Blob’s moves on the grid (using **'A'**, **'W'**, **'S'** and **'D'** keys on the keyboard or arrows, whether you have CAPSLOCK on or not). Blob can only move up, down, left or right, one position at a time. When it hits a border it should stop there until the user moves it in a different (suitable) direction. The program should redraw the grid between each move to update the position of Blob at its new coordinates. 2. Get the code from GiT [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/blobstarter.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/blobstarter.git) if you don’t know how to do this in Visual Studio, check out the lecture video “how to download from git.mp4” 3. Can you figure out how the starter program works and what all the functions do? Try writing pseudo code, flowcharts, pDDs, decomposition diagrams – anything that helps you get it straight in your head as to what is going on. Plus, you can use them in the design doc anyway. 4. Make the grid 10x10 5. Is it more fun if Blob hits a border and reappears on the other side of the grid? 6. Add a second item (let’s say a zombie: **'Z'**). It automatically follows Blob, moving up, down or in diagonal towards Blob. When the zombie catches up with Blob the program terminates. 7. Position the two characters at random of the grid, making sure they do not overlap. 8. Create the usual design docs and test plan – use the design doc template. 9. Use different colours for the grid background, borders and the character(s) moved by the player. 10. Display a score for the player, they get one point every time they move, also display the score at the end. So it’s a survival runner! 11. **Optional** – is it too easy? That’s no fun. Possible solution – at a certain point the zombie starts to run (warn the player) or extra zombies start appearing (complicated though)? Obstacles in the map to make it harder for the player? Any other ideas?   Competencies: 1.1,1.2.1,1.2.2,1.2.3,1.2.4,1.2.5,1.2.7,1.3,1.6,1.7,1.9,2.2,2.3 |