CT4019 PROGRAMMING & MATHEMATICS FOR GAMES

Arrays and Strings

Attendance
PIN

xxxx

Reminder

Each class in this module covers a core programming topic.

What is an Array?

"An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier"

"same type"

"contiguous memory"

"index to a unique identifier"

Creating an Array

```
int aIntegerArray[ 5 ] = { 10, 11, 12, 13, 14 };
```

0th	1st	2nd	3rd	4th
10	11	12	13	14

- We declare an array similarly to how we declare any variable, it starts with a type.
- Next we give it a unique name.
- Next we use square brackets and a constant integer value, this value is the size of our array (i.e. how many elements it can contain).
- Lastly I'm just going to initialise each element in the array.
- This table shows the values stored in each element of the array after initialisation.

Accessing Individual Elements of an Array

```
int aIntegerArray[ 5 ];

aIntegerArray[0] = 10;

aIntegerArray[1] = 11;

aIntegerArray[2] = 12;

aIntegerArray[3] = 13;

aIntegerArray[4] = 14;
```

0th	1st	2nd	3rd	4th
10	11	12	13	14

"index to a unique identifier"

- We can also access each individual element of the array using the square brackets.
- In this example we declare the array as last time.
- This time however we individually assign each value by using it's unique index (0-4).
- In this way we can reference and assign individual values to and from that element.
- Ultimately the array ends up the same as before.

Out of Bounds

```
int aIntegerArray[ 5 ];

aIntegerArray[0] = 10;

aIntegerArray[1] = 11;

aIntegerArray[2] = 12;

aIntegerArray[3] = 13;

aIntegerArray[4] = 14;

aIntegerArray[5] = 15;
```

- Consider the same array again.
- What happens if I try to assign a value to index 5?
- C++ **will** allow us to go off the end of the array and keep assigning values.
- These 'out of bounds' errors essentially overwrite other memory blocks and can cause huge problems.

0th	1st	2nd	3rd	4th	5th
10	11	12	13	14	???

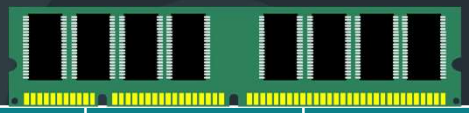
What happens when we assign a variable?

- Consider some basic code that just creates and assigns a couple of variables.
- As soon as line 3 is executed a request is sent to our memory management unit (MMU).
- The request essentially says, please give me some memory big enough to hold a signed integer (int), i.e. 4 bytes of memory.
- For our program to access this 4 bytes of memory we need to know the address of that memory.

```
1   int main()
2   {
3       int i;
4
5       i = 0;
6
7       int j;
8
9       j = 1;
10
11      return 0;
12  }
```

Get a memory address for ‘i’

```
1   int main()
2   {
3       int i;
4
5       i = 0;
6
7       int j;
8
9       j = 1;
10
11      return 0;
12  }
```

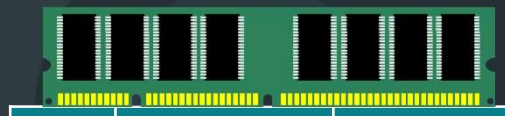


Name	Memory Address	Value
i	0x0039fb84	-858993460

- So after executing line 3, we should now have a memory address to our integer variable ‘i’.
- This memory address 0x0039fb84 corresponds to a position in our system memory.
- Memory addresses are represented by a hexadecimal value.
- As we haven’t initialised it to anything the value will be garbage (depending on the compiler).
- So next we have to initialise that value for it to be of any use to us.

Assigning a value to our memory

```
1  int main()
2  {
3      int i;
4
5      i = 0;
6
7      int j;
8
9      j = 1;
10
11     return 0;
12 }
```

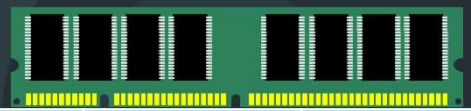


Name	Memory Address	Value
i	0x0039fb84	0

- So after executing line 5 at runtime we follow the memory address we have for the variable ‘i’.
- Notice the memory address stays the same here, all we’ve done is assign a value to it.

Get a memory address for ‘j’

```
1  int main()
2  {
3      int i;
4
5      i = 0;
6
7      int j;
8
9      j = 1;
10
11     return 0;
12 }
```

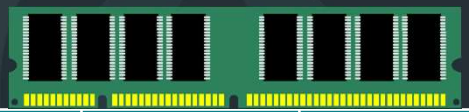


Name	Memory Address	Value
i	0x0039fb84	0
j	0x0093fe50	-858993460

- Just like last time, executing line 7 goes to our MMU and asks for a memory address for an integer.
- Again until we initialise this one the value is garbage.
- Now our program has two memory addresses in which to store values for ‘i’ and ‘j’.

Lastly assign a value to ‘j’

```
1  int main()
2  {
3      int i;
4
5      i = 0;
6
7      int j;
8
9      j = 1;
10
11     return 0;
12 }
```



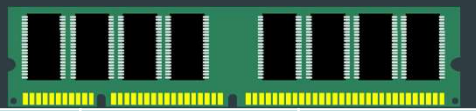
Name	Memory Address	Value
i	0x0039fb84	0
j	0x0093fe50	1

- Again just like before, the equals operator will assign a value to this memory address.
- Now we have two fully initialised variables in our program we can change the value of.
- The memory addresses will remain the same.
- These two memory addresses will probably be in **very different locations**, with other memory blocks between them.

Arrays are different

- So let's make our first array.
- Just like creating a variable we are given a memory address.
- This time we have told the MMU (using the square brackets) we want a memory address large enough to accommodate 5 integer values.
- After executing this line we will have a memory table that looks a bit like this.

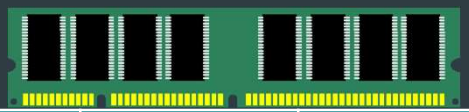
```
1   int main()
2   {
3       int i[5];
4
5       i[0] = 100;
6       i[1] = 200;
7       i[2] = 300;
8       i[3] = 400;
9       i[4] = 500;
10
11      return 0;
12  }
```



Name	Memory Address	Value
i	0x0115fa64	…
i[0]	0x0115fa64	-858993460
i[1]	0x0115fa68	-858993460
i[2]	0x0115fa6c	-858993460
i[3]	0x0115fa70	-858993460
i[4]	0x0115fa74	-858993460

Arrays are different

- Notice that the memory address for ‘i’ and ‘i[0]’ are the same.
- An array is just a memory address to the location of the very first element in your array, hence why the memory addresses of these two are the same.
- Also notice that the memory addresses after that are only 4 bytes apart from one another.
- 4 Bytes being the maximum size of the signed integer type we requested.
- In this way we know the memory is ‘contiguous’, i.e. there are no gaps between the memory blocks, it is one full block.



Name	Memory Address	Value
i	0x0115fa64	…
i[0]	0x0115fa64	-858993460
i[1]	0x0115fa68	-858993460
i[2]	0x0115fa6c	-858993460
i[3]	0x0115fa70	-858993460
i[4]	0x0115fa74	-858993460

“contiguous memory”

Assignment and Visual Studio

```
1   int main()
2   {
3       int i[5];
4
5       i[0] = 100;
6       i[1] = 200;
7       i[2] = 300;
8       i[3] = 400;
9       i[4] = 500;
10
11      return 0;
12  }
```

Watch 1

Name	Value
&i[0]	0x010ff728 {100}
&i[1]	0x010ff72c {200}
&i[2]	0x010ff730 {300}
&i[3]	0x010ff734 {400}
&i[4]	0x010ff738 {500}

- Lines 5 through to 9 use square brackets and an index number to assign a value to a specific element in the array.
- We can actually see this directly in Visual Studio by adding a breakpoint.
- In the watch window we can add all of our array elements.
- By prefixing them with an '&' it displays their memory location (and their actual value in the brackets)

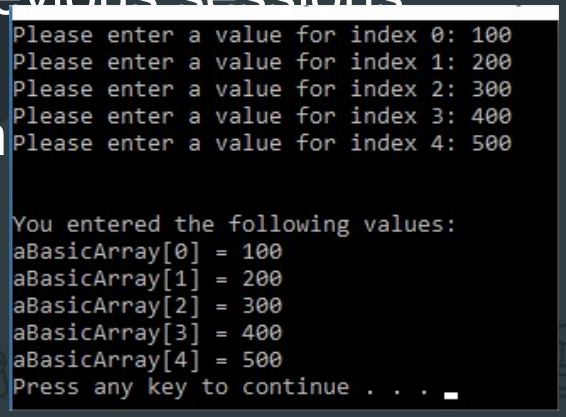
"index to a unique identifier"

Task #1

Create a program that will:

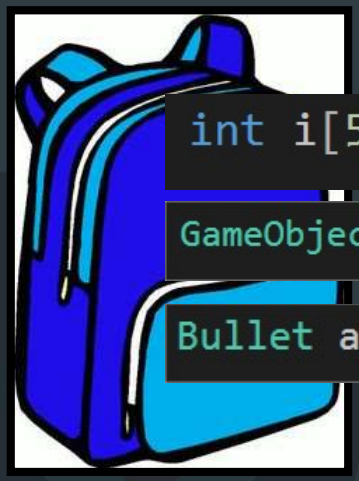
1. Ask the user to enter 5 integer values.
2. Store these values in an array.
3. Once all the values have been collected, print out each value.
4. Make use of the ‘for’ loop you learnt in previous sessions

The final output should look something like th



So now we can use one… what are they for?

- Arrays are the most basic form of ‘container’.
- A container is essentially just a holder object that stores a collection of other objects (elements).
- While there are a great many useful containers (vectors, deque, lists, etc.) arrays are the most basic.
- Arrays are great for storing collections of objects in our game worlds that we need to keep track of.
- In this mock example we’re iterating over an array of enemy soldiers, if the soldier is dead we delete their body.



```
int i[5];
```

```
GameObject aGameObjectArray[100];
```

```
Bullet aActiveBullets[750];
```

```
EnemySoldier aEnemySoldiers[35];
for( int i = 0; i < 35; ++i )
{
    if( aEnemySoldiers[i].IsDead() )
    {
        aEnemySoldiers[i].DeleteBody();
    }
}
```

Sorting and Searching

- An array is a contiguous block of memory which makes reading it pretty quick as its all in one place.
- This means arrays are useful for groups of values we need to sort in a particular order or search through.
- As all the memory is in one contiguous block our sorting and searching should be faster.
- For example sorting a list of possible waypoints an enemy AI can use to pathfind.
- Or searching through a large list of game objects to find a particular enemy.

Name	Memory Address	Value
i	0x0115fa64	…
i[0]	0x0115fa64	-858993460
i[1]	0x0115fa68	-858993460
i[2]	0x0115fa6c	-858993460
i[3]	0x0115fa70	-858993460
i[4]	0x0115fa74	-858993460

“contiguous memory”

Task #2

Create a program that will:

1. Ask the user to enter 5 float values.
2. Store these values in an array.
3. Then ask the user the index of a value they want to delete.
4. Shift each element of the array so the value of that index is overridden.
5. Add a value of ‘0’ to the last element in the array.

```
Enter a float value: 1.1
Enter a float value: 2.2
Enter a float value: 3.3
Enter a float value: 4.4
Enter a float value: 5.5

The array looks like this:
aFloatArray[0] == 1.1
aFloatArray[1] == 2.2
aFloatArray[2] == 3.3
aFloatArray[3] == 4.4
aFloatArray[4] == 5.5

Enter the index of a value to remove: 2

Now the array looks like this:
aFloatArray[0] == 1.1
aFloatArray[1] == 2.2
aFloatArray[2] == 4.4
aFloatArray[3] == 5.5
aFloatArray[4] == 0
Press any key to continue . . .
```

Multidimensional Arrays

```
// A one dimensional array.
int aIntegerArray[2]{ 100, 150 };
```

```
// A two dimensional array.
int aIntegerArray[2][2]
{
    { 100, 150 },
    { 600, 950 }
};
```

```
// A two dimensional array.
int aIntegerArray[5][3]
{
    { 100, 150, 150 },
    { 600, 950, 150 },
    { 111, 167, 191 },
    { 683, 950, 286 },
    { 124, 780, 380 }
};
```

- The arrays we have used up to now have been ‘one dimensional’ arrays (or 1D).
- Another useful thing about arrays is that they can actually have multiple dimensions.
- In this way we can represent a table of data using a two dimensional array (i.e. ROWS and COLUMNS).
- 2D arrays don’t have to be ‘square’ (i.e. they can have different numbers of rows and columns).

What are multidimensional arrays for?

- One thing you might use these for in your own assignments is for storing position.
- Here we have 4 sets of X and Y coordinates, each row represents the position of a different enemy.
- The first column holding their X coordinate and the second column their Y coordinate.
- This allows us to hold all our enemy position data in one place for ease of use.

```
// An array of X and Y positions.
float aPositionsIn2DSpace[4][2]
{   // X     Y
    { 54.6, 18.4 }, // Position of enemy 0
    { 11.2, 67.1 }, // Position of enemy 1
    { 78.8, 11.0 }, // Position of enemy 2
    { 5.11, 16.7 }, // Position of enemy 3
};
```

Nested Loops

```
// An array of X and Y positions.
float aPositionsIn2DSpace[4][2]
{   // X     Y
    { 54.6, 18.4 }, // Position of enemy 0
    { 11.2, 67.1 }, // Position of enemy 1
    { 78.8, 11.0 }, // Position of enemy 2
    { 5.11, 16.7 }, // Position of enemy 3
};
```

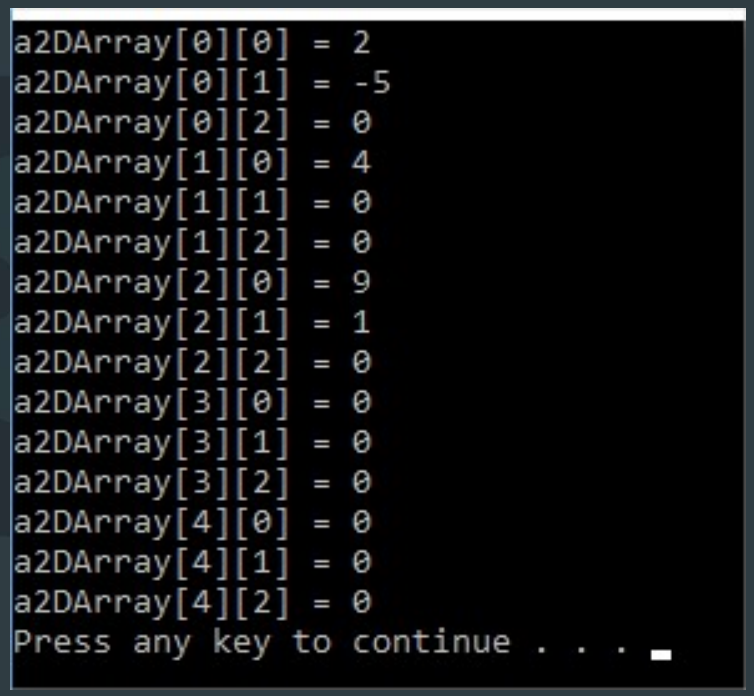
```
for( int i = 0; i < 4; i++ )
{
    for( int j = 0; j < 2; i++ )
    {
        float f = aPositionsIn2DSpace[i][j];
        std::cout << "f = " << f;
    }
}
```

- In order to iterate over multidimensional arrays you will often need to use ‘Nested’ loops.
- Consider the same 2D array of float values from before.
- In order to iterate over these values and print out each individual one I use a loop.
- Inside another loop.
- Both use different counters, the outer loop is essentially iterating us through each **ROW**.
- The inner loop is iterating us through each **COLUMN**.
- In this way we iterate over and print each value.

Task #3

Create a program that will:

1. Create a 2D integer array that has 5 rows and 3 columns.
2. Initialise the array with any integer values you like
3. Implement a nested loop that will iterate over and print each value.



char[ ]

- A character array (or char array) is the same as any other array.
- It contains a contiguous set of data of a given type (in this case, chars).
- In ye olde C programming this was how you would represent a series of characters (i.e. a word).
- Very often programming tests will require you to solve problems using char arrays.
- However thankfully now we have Strings…

```
char aWord[5] = {'H', 'E', 'L', 'L', 'O' };

// OR

char aWord[5];

aWord[0] = 'H';
aWord[1] = 'E';
aWord[2] = 'L';
aWord[3] = 'L';
aWord[4] = 'O';
```

Strings

- Strings are similar to arrays but they specifically represent a sequence of **characters**.
- In a lot of ways strings are the same as having an array of type **char**.
- They do however have some extra functionality that a **char** array does not.
- To use the string class, #include it at the top of your file.
- Next just declare a new string like any other variable, here I have created a string to represent my full name and printed it out to the console.

```
#include <string>
```

```
string sFullName = "Will Masek";
cout << sFullName << endl;
```

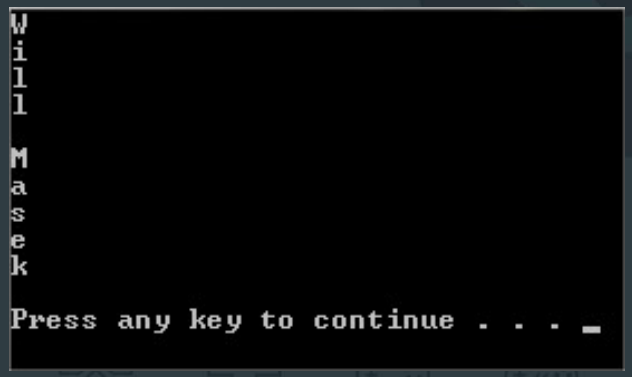
```
Will Masek

Press any key to continue . . .
```

Treating Strings like Arrays

```
string sFullName = "Will Masek";

cout << sFullName[ 0 ] << endl;
cout << sFullName[ 1 ] << endl;
cout << sFullName[ 2 ] << endl;
cout << sFullName[ 3 ] << endl;
cout << sFullName[ 4 ] << endl;
cout << sFullName[ 5 ] << endl;
cout << sFullName[ 6 ] << endl;
cout << sFullName[ 7 ] << endl;
cout << sFullName[ 8 ] << endl;
cout << sFullName[ 9 ] << endl << endl;
```



- As mentioned before, Strings are very similar to arrays.
- Just like arrays we can access each individual element of our string using square brackets.
- In this program again I create a string representing my full name.
- I then access each individual character element and print it to the screen.

Useful String Functions

- Strings can be a pain to work with but they do have a few useful functions to help us.
- The size() function returns the length of the string.
- I can use this to make a more streamlined version of our last program, using the value returned by **size()** to iterate over the string and print it.
- The **substr()** function creates a sub-string of the specified indexes.
- The first parameter is the index where I want my sub-string to start.
- The second parameter is how many characters from that index I want my sub-string to end.
- Using **substr()** I can create two separate strings, one for my first name and another for my surname.
- Then I can print them out separately.

```
string sFullName = "Will Masek";
int iStringSize = sFullName.size();
```

```
for( int i = 0; i < iStringSize; ++i )
{
    cout << sFullName[ i ] << endl;
}
```

```
std::string sFullName = "Will Masek";
std::string sFirstName = sFullName.substr(0, 4);
std::string sLastName = sFullName.substr(5, 5);

std::cout << "First name: " << sFirstName << std::endl;
std::cout << "Last name: " << sLastName << std::endl;
```

```
Your first name is: Will
Your surname is: Masek

Press any key to continue . . .
```

Other Useful String Functions

- **compare()** – Compares two strings and returns an integer value.
- **resize()** – Change the length of a given string.
- **empty()** – Test if a string is empty.
- **clear()** – Erase the contents of a string.
- **erase()** – Remove one or more characters from a string.
- For a full list see: *http://www.cplusplus.com/reference/string/string/*

Task #4

Create a program that will:

1. Take a string input from the user.
2. Print out the entered string.
3. Iterate over the string and remove any vowels.
4. Print out the string now the vowels have been removed.

Final Task

Write a program that reverses the order of the **words** in a string.

For example, your function should transform:

“Do or do not there is no try”

To

“try no is there not do or Do”

Assume all words are separated by a space.