Started on	Wednesday, 7 August 2019, 1:23 PM
State	Finished
Completed on	Wednesday, 14 August 2019, 9:37 PM
Time taken	7 days 8 hours
Marks	28.60/30.00
Grade	**9.53** out of 10.00 (**95**%)

Information

Pointers and Strings

Goals

This lab introduces what is probably the most significant and troublesome feature in C: pointers. It also shows the relationship between pointers and arrays. C "strings", which are just null-terminated arrays of characters, are discussed as a specific example area. By the end of the lab you should:

- appreciate that pointers are just addresses in a linear address space,
- understand how a typical Linux program is laid out in that address space,
- be familiar with the '&' and '*' operators for obtaining addresses of items and for dereferencing addresses respectively,
- be able to use pointers to variables as parameters rather than values of variables,
- be able to perform simple array manipulations using pointer arithmetic rather than indexing,
- be able to use some of the C string functions, which usually take pointers as parameters,
- be able to implement your own versions of the simpler C string functions.

This lab covers the material in chapters 11 and 12 of the text by King, plus some of chapter 13.

A playlist of videos on pointers is here. And a playlist on strings is here.

Information

An introduction to pointers

Pointers are introduced in the two ENCE 260 videos 4.1 An introduction to pointers and 4.2 What pointers are used for, and also in the entertaining public video *Binky Pointer Fun*.

It is assumed that you are familiar with the idea that computer memory is a large sequence of storage locations numbered 0, 1, 2, 3, · · · several billion. The *address* of a storage location is its number in the sequence. Assignment statements in C such as `x = y` compile to machine code that says, effectively, "copy the contents of memory location `y` to memory location `x`". In some hypothetical assembly language[1] this might be written

```
mov y, x
```

If *y* is at location 14004 in memory and *x* is at location 16020, this machine instruction is equivalent to

```
mov 14004, 16020
```

Note that this meaning of assignment is very different from the Python meaning of "bind the name x in the current dictionary to the object y".

A pointer is a variable that holds a memory address. Consider the following fragment of C code in conjunction with Fig. 1 below.

```
    int num = 20;
    int* ptr = NULL;
    ptr = &num;
    printf("%d\n", *ptr);
```

- Line 1 declares a location called `num` that contains an integer and initialises its value to 20. In Fig. 1, it is arbitrarily assumed that the variable `num` has landed up at memory location 1020. Being an `int`, the variable actually occupies the four bytes of storage 1020 - 1023, with the least significant byte of the integer (containing decimal 20 in this case) at location 1020 and the most significant byte (zero) at 1023. However, its address is that of the lowest byte occupied. Note that addresses have been drawn increasing from right to left so that the layout of the bytes of `num` look right. The actual addresses aren't very realistic in a Linux C program, but we'll worry about how programs are actually laid out in memory in the next section of the lab.
- Line 2 declares a variable called `ptr` that points to an integer, and initializes its value to NULL. The value NULL is just 0, but it's a pointer not an *int*. Don't worry about the syntax for a moment, just read on. In Fig. 1, it's assumed that this new variable has landed up at location 3216 decimal. So location 3216 now "points at" memory location zero.
- Line 3 sets the value of the variable `ptr` to be the *address of* the variable `num`. So at last you get told the meaning of that '&' operator you've been using with `scanf` - it takes the address of its operand! So in Fig. 1, where `num` is at location 1020, the value of `ptr` becomes 1020.
- Line 4 prints out the value of the expression `*ptr`. The unary operator '*' is the *indirection operator* or *dereference operator*, which can be read as "the value at the location pointed to by". Thus, the expression `*ptr` means "the value at the location pointed to by `ptr`", and since `ptr` was set to point to `num` the value 20 gets printed out.

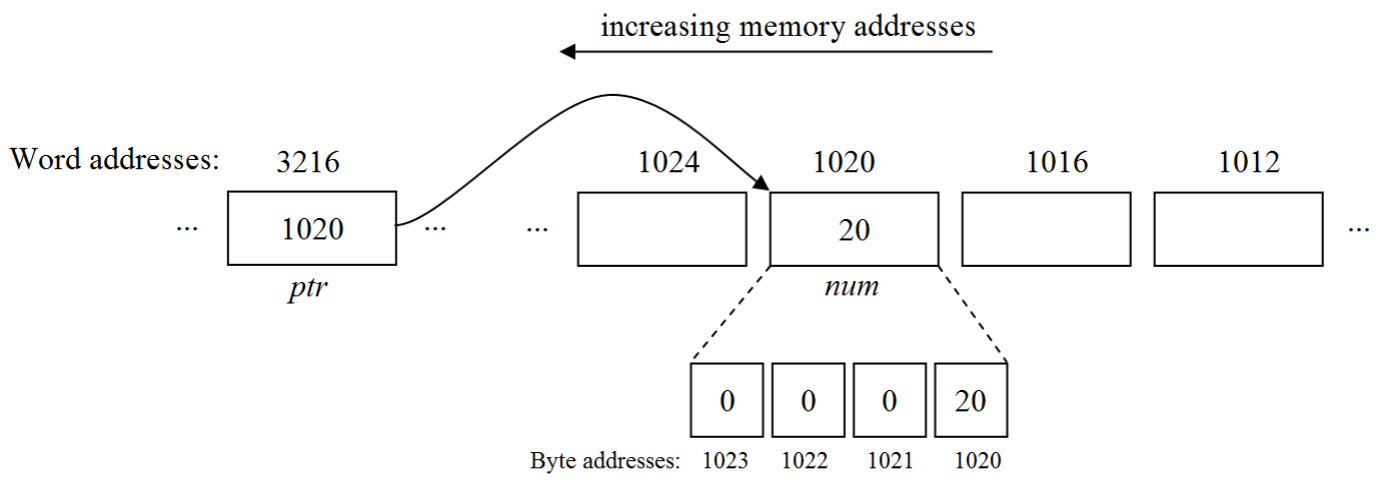


Figure 1: A pointer variable `ptr` pointing at a variable `num`. The numbers above or below the boxes are decimal memory addresses (which are arbitrary) and the numbers within the boxes are the contents of those memory locations.

Important note. In ENCE260 we use a slightly different convention from the textbook when writing pointer declarations. Where we declare a pointer, initialised to NULL (zero), as

```
int* p = NULL;
```

the textbook declares it with different spacing as

```
int *p = NULL;
```

The two are exactly equivalent; the textbook way is more common in the C community but our way is more common in the C++ community. We prefer the first declaration because the second one looks very like the assignment statement

```
*p = NULL
```

which sets *the value pointed to* by `p` to `NULL`. This is confusing because the declaration is initialising *the pointer itself* to NULL. However, you may use the textbook approach if you prefer.

You should, though, be aware of one nasty trap with our version. The declaration

```
int* p, q
```

does not, as you might expect, declare two int pointers `p` and `q`. Rather, it is taken by the compiler as

```
int *p, q;
```

which declares `p` to be a pointer to an `int` but `q` as just an `int`!

If you follow our style rule of declaring each variable separately, you won't ever hit that problem.

---

[1]Assembly language is a symbolic version of machine language in which we use mnemonics like 'mov' to mean 'a move instruction' and names like 'x' and 'y' instead of the memory addresses at which the variables reside.

Question **1**

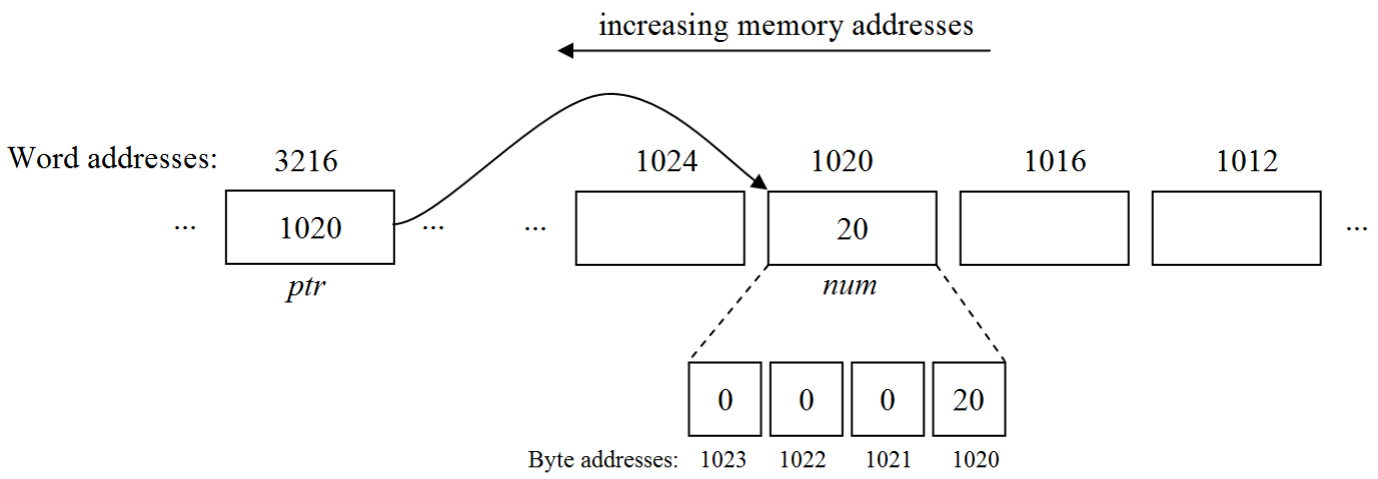
Correct

Mark 1.00 out of 1.00

Suppose the statement `*ptr = 50` was added to the end of the above code fragment, giving

```
int num = 20;
int* ptr = NULL;
ptr = &num;
printf("%d\n", *ptr);
*ptr = 50;
```

After execution of the code fragment, which of the following statements would be true, assuming the memory layout of Fig. 1 (shown again below):



Select one:

- a. Location 1020 contains the value 50. ✔
- b. Location 1020 contains the value 3216.
- c. Location 50 contains the value 3216.
- d. Location 1024 contains the value 50.
- e. Location 3216 contains the value 50.

Correct

Marks for this submission: 1.00/1.00.

Question **2**

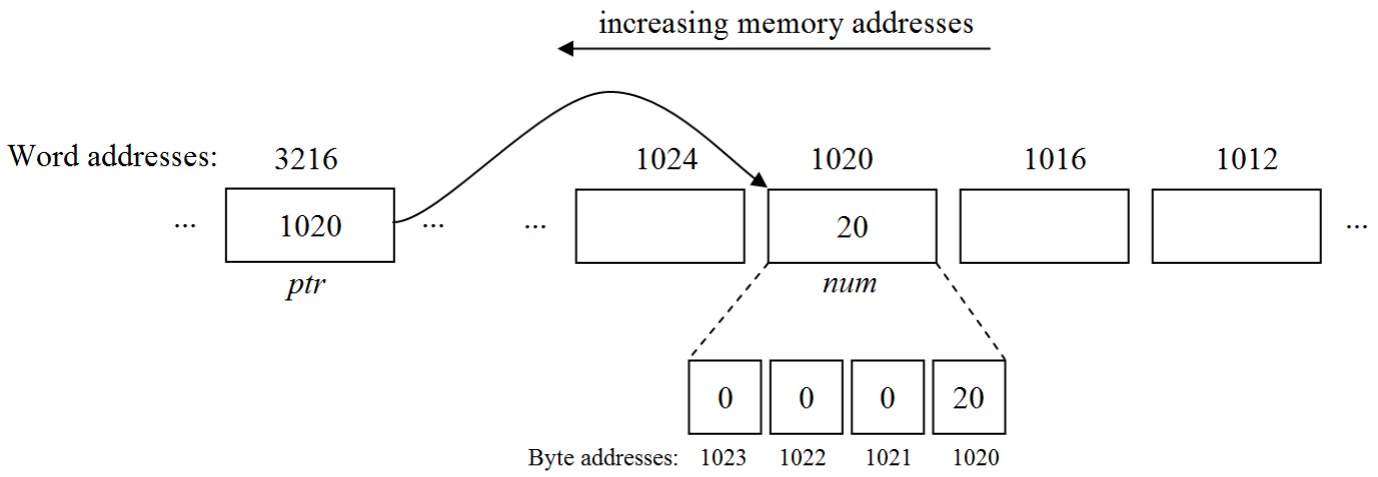
Correct

Mark 1.00 out of 1.00

Consider again the original code fragment given in the introduction, without the addition of question 1:

```
int num = 20;
int* ptr = NULL;
ptr = &num;
printf("%d\n", *ptr);
```

After execution of the code fragment, assuming the memory layout of Fig. 1 (shown again below), what are the values of the *expressions* in the left column below?



*ptr	20 ✔
&ptr	3216 ✔
ptr	1020 ✔
&num	1020 ✔
num	20 ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

A pointer variable is like a type of int variable, where the value is interpreted as the address of some data in memory. Most modern computers use byte addressing, i.e. memory is regarded as a sequence of bytes numbered 0, 1, 2 up to the maximum memory size, which might be 16 GB or more. A pointer variable has to be able to point to any memory address.

Our lab machines typically have 16 GB usable memory. The int data types are: char (8-bit), short (16-bit), int (32-bit) and long (64-bit). All may be either signed or unsigned.

Which of the following integer types is most nearly equivalent to a char* variable?

Select one:

- signed char
- unsigned char
- signed int
- unsigned int
- signed long
- unsigned long ✔ Yes, that's right. A 32-bit number isn't large enough to address 16 GB so you need a 64-bit number (pointer). And it should be unsigned as addresses can't be negative.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 1.00 out of 1.00

A function *printViaPtr*, which prints on a line by itself the integer pointed to by the parameter it is given, could be defined as follows:

```
void printViaPtr(const int* intPtr)
{
    printf("%d\n", *intPtr);
}
```

Write a function *void print2Ints(int i, int j)* that takes two *int* parameters *i* and *j* and prints *i* then *j* **using *printViaPtr()***.

Important: Paste only your *print2Ints* function. Do not include *printViaPtr*, which is supplied by the tester.

For example:

Test	Result
`print2Ints(11, -93);`	`11` `-93`

Answer: (penalty regime: 0, 10, ... %)

```
1
2  void print2Ints(int i, int j)
3  {
4      int* Pointer1 = &i;
5      int* Pointer2 = &j;
6      printViaPtr(Pointer1);
7      printViaPtr(Pointer2);
8  }
9
```

	Test	Expected	Got	
✔	`print2Ints(11, -93);`	`11` `-93`	`11` `-93`	✔

✔	`// Check with more hidden code that`
	`// printViaPtr is indeed being called.`
	`print2Ints(107, -10001);`
	`checkNumCalls();`
	`print2Ints(19, 2);`
	`checkNumCalls();`
		`107`
		`-10001`
		`printViaPtr called 2 times`
		`19`
		`2`
		`printViaPtr called 4 times`
			`107`
			`-10001`
			`printViaPtr called 2 times`
			`19`
			`2`
			`printViaPtr called 4 times`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

Interlude: the memory layout of a program

The layout in memory of a program and its various data areas is implementation dependent and code you write should not depend upon any particular layout properties. However, it is vitally important when using pointers that you have some sort of picture in your head of how memory is laid out, even if that picture doesn't exactly correspond to reality. In this section we look at the memory layout of a typical program running under a 32-bit Linux system. Our laboratory machines actually use 64-bit Linux, but the principles and general memory layout are the same. 64-bit addresses are very tedious to write in hexadecimal, e.g 0xFE1004ACEF10007F!

On a 32-bit machine, addresses are always 32-bit numbers in the range 0 - 0xFFFFFFFF. [If you don't understand hexadecimal numbers, ask a tutor.]. This four gigabyte range of addresses is called the *virtual address space* of the program. Any given program usually uses only a small portion of its virtual address space and a computer typically has multiple programs loaded into its memory at any one time. Memory management hardware maps the virtual addresses of the currently-running program to the program's real addresses in physical memory. This is a complex process, made even more complex by the fact that larger programs are usually only partially resident in physical memory with the rest of the code and/or data "swapped out" to disk.

At least in this section of ENCE260, you're not expected to understand how virtual memory systems work. Instead it is sufficient for now to imagine that your computer does in fact have a full 4GB of actual memory which is occupied by just a single program plus the operating system. In this simplified view, virtual addresses are the same as physical addresses.

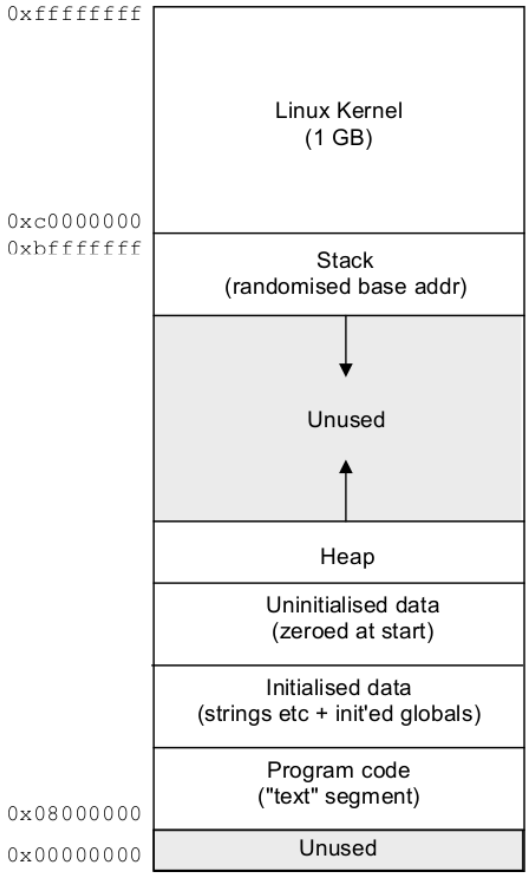


Figure 2: The memory layout of a program running on a 32-bit Linux system.

Figure 2 shows a slightly simplified representation of the layout of the virtual memory of a program running under a 32-bit Linux system. For example, consider the following program.

```
#include <stdio.h>
int uninitialisedArray[10];
int initialisedArray[] = {100, 200, 300};
int globalNum = 20;
int main()
{
    int data[10];
    int n = 30;
    char c;
    const char* name = "Angus";
    // Do things
    printf("%s\n", name);
    return 0;
}
```

We will assume the program is statically linked, i.e. all the library functions it needs are included in the executable file that is loaded when the program is run [Not the normal default case, but I'm trying to keep things tolerably simple]. Now, referring back to Figure 2.

- The lowest 128MB of virtual memory (`0 - 0x07ffffff`) is unused. Any attempt to access this area, for example, by attempting to dereference a null pointer (e.g. int* p = NULL; int x = *p;), will cause the program to crash with a "segmentation fault, core dumped" message from the operating system.
- The program code and global (external) data is at the bottom of the address space, just above the unused segment. In our example, this includes all the program code, statically-linked library functions, literal strings and constants and the global data.

- The top 1GB of virtual memory is occupied by the operating system kernel. This includes all the low-level code for task switching, memory management and I/O.
- All the initialised global data, i.e., global variables that have been given an initial value, e.g. `initialisedArray` and `globalNum`, is placed directly above the program. As a further subtlety, literals, such as the strings `"Angus"` and `"%s\n"` are located in a sub-area of the initialised global data, which is set to allow read-only access.
- Uninitialised global data, such as `uninitialisedArray`, is placed directly above the initialised global data.
- The program stack, which contains all the activation records or "stack frames" for each active function call, grows downwards from just below the Linux kernel. Functions' local variables are in the stack. Note, though, that the exact location of the run-time stack base is randomised at run time to make it harder for hackers to exploit buffer-overflow vulnerabilities. Thus you'll get different local variable addresses every time you run a Linux program nowadays.
- The program heap, which we don't use until we do dynamic memory allocation, grows upwards from the top of the code and data area until it hits the stack area. The heap is the area that was referred to as "the object store" in COSC121.

QUESTION

With reference to the program and figure above, match up the items in the column on the left with their possible addresses in the column on the right so that the entire set of addresses is consistent with the Linux layout described above. [The actual numbers don't necessarily match any actual Linux execution; all that matters is that the addresses are consistent with the assumed memory layout.]

Address of variable c	0xbef840c8 ✔
Address of variable `globalNum`	0x085103a8 ✔
Address of `main` function	0x0848fe30 ✔
Address of `uninitialised` array	0x085ffea0 ✔

Correct

Marks for this submission: 1.00/1.00.

Information

Using pointers as parameters

Now we'll look at some uses of pointers. For an introduction see the video 4.2 What pointers are used for.

In C, parameters to functions are passed "by value", meaning that the value of the argument is used to initialise the parameter variable within the function. Any changes the function makes to such a parameter have no effect on the original variable in the caller's space. Sometimes, however, we *want* the called function to be able to change our own local variables, as when we call `scanf` for example. To achieve this in C, pointers must be used: we explicitly pass the address of the local variable rather than its value. The called function can then explicitly manipulate the caller's variable.

Question **6**

Correct

Mark 2.00 out of 2.00

Write a function *void swap(int* pi, int* pj)* which swaps the values of the integers pointed to by the pointers *pi* and *pj*. Your answer should consist *only* of the function *swap*; do not include any test code.

HINT: *You can't program pointers if you can't visualise them.* And unless you're very unusual, when you're learning about pointers you can't visualise them just by staring into space or thinking about syntax. So: when programming with pointers you need to have a sheet of paper on which to sketch what the pointer variables are pointing at.

For example:

Test	Result

`int i = 10, j = 20;`
`swap(&i, &j);`
`printf("%d %d\n", i, j);`
	`20 10`

Answer: (penalty regime: 0, 10, ... %)

```
1  void swap(int* pi, int* pj)
2  //take in 2 pointer pi and pj and swop their reference addresses?
3  {
4      int tempa = 0;
5      tempa = *pi;
6      *pi = *pj;
7      *pj = tempa;
8
9
10
11 }
12
```

	Test	Expected	Got	

✔	`int i = 10, j = 20;`
	`swap(&i, &j);`
	`printf("%d %d\n", i, j);`
		`20 10`	`20 10`	✔

✔	`int i = 100, j = -1;`
	`swap(&i, &j);`
	`printf("%d %d\n", i, j);`
		`-1 100`	`-1 100`	✔

✔	`int i = 300;`
	`swap(&i, &i);`
	`printf("%d\n", i);`
		`300`	`300`	✔

Passed all tests! ✔

Correct

Marks for this submission: 2.00/2.00.

Question **7**

Correct

Mark 3.00 out of 3.00

The program below, which has also been preloaded into the answer box, has THREE places where the string '...' occurs. You should replace each of those '...' occurrences with a small fragment of C code such that the program reads a single character from standard input and (in a very roundabout way) prints it to standard output. Each code fragment must be only a few characters in length and cannot include any newline characters.

Your submission will be rejected if you change anything other than the three '...' strings.

```
#include <stdio.h>
void oof(char* p1, char* p2)
{
    ... = ...;
}

int main(void)
{
    char c = 0;
    char other = '#';
    scanf("%c", &c); // Reads a single character into c
    oof(&other, ...);
    printf("%c\n", other);
}
```

For example:

Input	Result
X	X
Y	Y
*	*

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
1  #include <stdio.h>
2  void oof(char* p1, char* p2)
3  {
4      *p1 = *p2;
5  }
6
7  int main(void)
8  {
9      char c = 0;
10     char other = '#';
11     scanf("%c", &c); // Reads a single character into c
12     oof(&other, &c);
13     printf("%c\n", other);
14 }
```

	Input	Expected	Got	
✔	X	X	X	✔
✔	Y	Y	Y	✔
✔	*	*	*	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Question **8**

Correct

Mark 5.00 out of 5.00

Write a function `void findTwoLargest(const int data[], int n, int* largest, int* secondLargest)` that stores the largest and second largest elements in the array `data` of length `n >= 2` in the variables pointed to by the given pointers `largest` and `secondLargest` respectively. If there is more than one occurrence of the largest value, that value should be used for `secondLargest` too. The array *data* must not be altered by a call to `findTwoLargest`.

The above signature introduces the `const` type qualifier. This qualifies the type of the parameter `data`, declaring that it is not just an array of ints, but an array of *constant* ints, i.e., their values will not be changed during the execution of the function. Using `const` qualifiers in function signatures in this way can be helpful to users (who now know that their data will not be altered by calling this function) and also prevents you as a programmer inadvertently assigning to the array.

The following program could be helpful in developing your solution to this problem. Make sure you understand it fully before starting to write code yourself.

```
// A test program for the "findTwoLargest" function

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void printArray(const int data[], int n)
// Print the first n elements of array data in braces, comma-separated
{
    if (n <= 0) {
        printf("{}");
    }
    else {
        printf("{%d", data[0]);
        for (int i = 1; i < n; i++) {
            printf(",%d", data[i]);
        }
        printf("}");
    }
}

void test_array(const int data[], int n)
// Test the function findTwoLargest on array 'data' of length 'n.
// It is assumed that n >= 2.

{
    int largest = 0, second = 0;

    findTwoLargest(data, n, &largest, &second);
    printf("The two largest elements from ");
    printArray(data, n);
    printf(" are %d and %d\n", largest, second);
}

// Next, a set of test arrays

int array1[] = { 1, 2, 3, 4, 5, 6 };
int array2[] = { 20, 19, 18 };
int array3[] = { 4, 4, 4, 4 };
int array4[] = { 17, 14 };
int array5[] = { 4, 45, 123, 3, 345, 27, 479 };

// Lastly, the main test routine.
int main()
{
    test_array(array1, 6);
    test_array(array2, 3);
    test_array(array3, 4);
    test_array(array4, 2);
    test_array(array5, 7);
    return 0;
}
```

For example:

Test	Result

`int data[] = {1, 3, 4, 0, 5, 3, 2, -1};`
`int result1 = 0, result2 = 0;`
`findTwoLargest(data, 8, &result1, &result2);`
`printf("%d %d\n", result1, result2);`
	`5 4`

Test	Result

int data[] = {5, 4};
int result1 = 0, result2 = 0;
findTwoLargest(data, 2, &result1, &result2);
printf("%d %d\n", result1, result2);
printf("%d %d\n", data[0], data[1]);
	5 4
	5 4

Answer: (penalty regime: 0, 10, ... %)

```
void findTwoLargest(const int data[], int n, int* largest, int* secondLargest)

{
    if (data[0] < data[1]) {
        *largest = data[1];
        *secondLargest = data[0];
    } else {
        *largest = data[0];
        *secondLargest = data[1];
    }
    for (int i = 2; i < n; i++) {
        //if data greater than largest, largest = data put old value of largest into second largest
        if (data[i] >= *largest) {
            *secondLargest = *largest;
            *largest = data[i];
        } else if (data[i] > *secondLargest) {
            *secondLargest = data[i];
        }
    }
}

```

	Test	Expected	Got	

✔	int data[] = {1, 3, 4, 0, 5, 3, 2, -1};
	int result1 = 0, result2 = 0;
	findTwoLargest(data, 8, &result1, &result2);
	printf("%d %d\n", result1, result2);
		5 4	5 4	✔

✔	int data[] = {5, 4};
	int result1 = 0, result2 = 0;
	findTwoLargest(data, 2, &result1, &result2);
	printf("%d %d\n", result1, result2);
	printf("%d %d\n", data[0], data[1]);
		5 4
		5 4
			5 4
			5 4
				✔

✔	int data[] = {50, 50};
	int result1 = 0, result2 = 0;
	findTwoLargest(data, 2, &result1, &result2);
	printf("%d %d\n", result1, result2);
		50 50	50 50	✔

Passed all tests! ✔

Correct

Marks for this submission: 5.00/5.00.

Information

Pointers and arrays

The relationship between pointers and arrays is covered in the videos 4.3 Pointers and Arrays and 4.4 Array parameters.

Pointers have three main uses in C:

1. To allow a called function to access the caller's variables, as introduced in the previous section.
2. As an alternative to traditional indexing (e.g. a[i]) when accessing arrays, usually with the goal of producing more compact or more efficient code.
3. As a way of accessing machine device registers which are located at specific known memory addresses. You'll see examples of this in the second term of the course when programming microcontrollers. It's not relevant in this term.

In this section of the lab we'll look briefly at the relationship between pointers and arrays, showing that an array variable is essentially just a pointer to a reserved area of memory. We'll then move on to use (2) above, showing how arrays can be manipulated directly via pointers rather than by using indices. This is *not* a technique we wish to encourage much in ENCE260, because:

1. In general, correctness and maintainability are much more important than performance. Code using pointers is more bug-prone and harder to read.
2. The performance gain from using pointers rather than array indexing is usually negligible with modern compilers and machine architectures. Indeed, hand-crafted pointer-based code will sometimes run significantly slower than code using subscripted arrays because the latter is more amenable to code optimisation within the C compiler.

Nonetheless, an understanding of pointer arithmetic and its application to array manipulation is fundamental to understanding C properly, and there is still a lot of code around that uses such techniques.

Pointer arithmetic

A pointer is just a variable that contains a memory address. So, if `p` is a pointer to an integer, what would you expect `p+1` to be? The most obvious answer--that it's a pointer to the next address in memory--isn't quite right. If `p` points to an integer, then with a typical 4-byte-per-integer architecture, `p+1` would then point at the second byte within the same integer. That certainly wouldn't be useful. Instead, C defines `p+1` to point to the next *integer* in memory, assuming `p` is defined as being a pointer to an integer. Hence if `p` points to element `a[0]` of some integer array `a` then `p+n` points to element `a[n]`. This holds true for arrays of any type.

In C if you use the name of an array as a stand-alone variable it is interpreted as a pointer to the base of the array. Hence, instead of accessing an array element by writing `a[n]` we can instead write `*(a+n)` or `*(n+a)`. The familiar subscript notation is just shorthand for the pointer-arithmetic expression. You can refer to the 11th item of array `a` as `a[10]`, `*(a+10)`, `*(10+a)` or `10[a]`. The last of those comes as a particularly rude shock to most students.

If array elements are being accessed sequentially, as is often the case, then a common C idiom is to repeatedly evaluate the expression `*p++`, where `p` is initially set to the base of the array. This is evaluated as *(p++) meaning that the access occurs indirectly through the pointer *p* after which the pointer is incremented to point to the next item in the array. The first evaluation yields the zeroth element of the array, the next yields the first element, and so on. Although the code may look compact it isn't encouraged in this course because it is error prone, hard to read and rarely offers a significant performance gain.

As an example, consider the following code. Predict what it will do. Copy it to a file *arrayprinters.c*, compile and run it. Were you right?

```
// Program to demonstrate use of pointers to manipulate arrays

#include <stdio.h>

void arrayPrinter1(const int a[], int n)
// A simple Java-like version using array indexing
{
    printf("arrayPrinter1:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n\n");
}

void arrayPrinter2(const int* a, int n)
// A version using pointer arithmetic instead of indexing
{
    printf("arrayPrinter2:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", *(a + i));
    }
    printf("\n\n");
}

void arrayPrinter3(const int* a, int n)
// A version using autoincrement on a pointer
{
    printf("arrayPrinter3:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", *a++);
    }
    printf("\n\n");
}

void arrayPrinter4(const int* a, int n)
// A version that dispenses with a loop control variable
{
    const int* endOfArray = a + n;
    printf("arrayPrinter4:\n");
    while (a < endOfArray) {
        printf("%d ", *a++);
    }
    printf("\n\n");
}

int main(void)
{
    int data[] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
    arrayPrinter1(data, 10);
    arrayPrinter2(data, 10);
    arrayPrinter3(data, 10);
    arrayPrinter4(data, 10);
    return 0;
}
```

As explained in the lecture notes and videos, a function declaration like

```
void arrayPrinter1(const int a[], int n)
```

can be written exactly equivalently as

```
void arrayPrinter1(const int* a, int n)
```

Question **9**

Correct

Mark 1.00 out of 1.00

Two of the array-printing functions in the program *arrayprinters.c* modify the value of the pointer parameter *a* (i.e., the address at which the pointer *a* points) during execution of the function. Which two?

Select one:

- a. arrayPrinter1 and arrayPrinter2
- b. arrayPrinter2 and arrayPrinter3
- c. arrayPrinter3 and arrayPrinter4 ✔
- d. arrayPrinter1 and arrayPrinter3
- e. arrayPrinter1 and arrayPrinter4
- f. arrayPrinter2 and arrayPrinter4

Correct

Marks for this submission: 1.00/1.00.

Question **10**

Correct

Mark 0.00 out of 1.00

As noted in the previous question, two of the array-printing functions modify the value of the parameter *a*. Which of the following statements is true?

Select one:

- a. It works correctly in this case but would fail if the name of the parameter were made the same as the name of the argument array in the main program, e.g., if the word 'data' were changed to 'a' throughout the main function.
- b. Modifying the parameter in this way has no effect on the state of the array as seen by the caller. ✔ This is correct. Parameters in C are passed by value, always. This means that a parameter to a function is essentially a local variable which is initialised to the argument value passed in. If a parameter is of a pointer type, the pointer value is passed by value and can be used to modify whatever it points at, but changing the parameter itself (e.g. the expression **a++* increments the parameter *a*) has no effect outside the function.
- c. Modifying the parameter in this way is technically correct within the array-printing function (in that it yields the right results), but would cause subsequent accesses to the array by the caller to yield wrong results.

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.00/1.00**.

Question **11**

Correct

Mark 1.00 out of 1.00

If the last executable statement in each array printing function were changed from

```
printf("\n\n");
```

to

```
printf("First array element: %d\n\n", a[0]);
```

which functions would behave correctly, i.e., correctly print the first array element?

Select one or more:

- a. arrayPrinter4
- b. arrayPrinter1 ✔
- c. arrayPrinter2 ✔
- d. arrayPrinter3

Correct

Marks for this submission: 1.00/1.00.

Question **12**

Correct

Mark 1.00 out of 1.00

Assuming the existence of a global declaration *char data[100]*, write a function *bool isInData(char* p)* that returns *true* (1) if and only if p points somewhere within the global *data* array (including at either end of it). You should assume the global data array has already been declared (**don't** declare it yourself) and that it contains exactly 100 bytes (= 100 chars).

Hints: Your function should compare the given address with the address of the first and last elements of *data* The body of the function should be just a single one-line statement. If you're writing a loop, you're definitely on the wrong track.

For example:

Test	Result
`printf("%d\n", isInData(&data[0]));`	`1`
`printf("%d\n", isInData(&data[17]));`	`1`
`printf("%d\n", isInData(&data[99]));`	`1`
`// Test with a hidden global thing1` `printf("%d\n", isInData(&thing1));`	`0`

Answer: (penalty regime: 0, 10, ... %)

```
bool isInData(char* p)
{
    //array is just allocated memory with pointer at the start
    //just check memory address at start and end of allocation?
    //how do i check address of

    //char* endOfData = &data[100];
    //if address of value p is before the address at data[100] and greater than the address at data[(
    return ((p < &data[100]) && (p >= &data[0]));
}
```

	Test	Expected	Got	
✔	`printf("%d\n", isInData(&data[0]));`	`1`	`1`	✔
✔	`printf("%d\n", isInData(&data[17]));`	`1`	`1`	✔
✔	`printf("%d\n", isInData(&data[99]));`	`1`	`1`	✔
✔	`// Test with a hidden global thing1` `printf("%d\n", isInData(&thing1));`	`0`	`0`	✔
✔	`// Test with a hidden global thing2` `printf("%d\n", isInData(&thing1));`	`0`	`0`	✔

✔	`// Testing with all elements of data`
	`// plus a lot of other addresses`
	`outside data.`
	
	`int numInside = 0;`
	`for (int i = -100; i <= 200; i++) {`
	`  if (isInData(&data[i])) {`
	`  numInside += 1;`
	`  }`
	`}`
	
	`printf("%d values were within data`
	`array\n", numInside);`
		`100 values were within data array`	`100 values were within data array`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **13**

Correct

Mark 1.00 out of 1.00

Write a variant of *isInData* with signature *bool isInData2(char data[], int arraySize, char* ptr)* that is given a (pointer to) an array of char, the size of the array and a pointer to a character somewhere in memory. The function returns true if and only if the variable *ptr* points somewhere within the array *data* (including at either end of it).

Your function cannot use any loops and should ideally have just a single line of code as its body (not including braces, of course).

For example:

Test	Result

`char x;`
`char thing[3];`
`char y;`
`printf("%d\n", isInData2(thing, 3, &x));`
`printf("%d\n", isInData2(thing, 3, &thing[0]));`
`printf("%d\n", isInData2(thing, 3, &thing[1]));`
`printf("%d\n", isInData2(thing, 3, &thing[2]));`
`printf("%d\n", isInData2(thing, 3, &thing[3]));`
`printf("%d\n", isInData2(thing, 3, &y));`
	`0`
	`1`
	`1`
	`1`
	`0`
	`0`

Answer: (penalty regime: 0, 10, ... %)

```
bool isInData2(char data[], int arraySize, char* ptr)
{
    char* endOfData = &data[arraySize];

    return ((ptr >= &data[0]) && (ptr < endOfData));
}

```

	Test	Expected	Got	

✔	`char x;`
	`char thing[3];`
	`char y;`
	`printf("%d\n", isInData2(thing, 3, &x));`
	`printf("%d\n", isInData2(thing, 3, &thing[0]));`
	`printf("%d\n", isInData2(thing, 3, &thing[1]));`
	`printf("%d\n", isInData2(thing, 3, &thing[2]));`
	`printf("%d\n", isInData2(thing, 3, &thing[3]));`
	`printf("%d\n", isInData2(thing, 3, &y));`
		`0`
		`1`
		`1`
		`1`
		`0`
		`0`
			`0`
			`1`
			`1`
			`1`
			`0`
			`0`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **14**

Correct

Mark 1.00 out of 1.00

Write a function *int myIndex(int data[], int* element)* that takes an *int* array data plus a pointer to an element within that array and returns the index of that element within the array.

For example:

Test	Result

`int data[30];`
`int* p = &data[17];`
`printf("Index is %d\n", myIndex(data, p))`
	`Index is 17`

Answer: (penalty regime: 0, 10, ... %)

```
int myIndex(int data[], int* element)
{
    // can i get end of array using [-1]
    int counter = 0;
    int *startPoint = &data[0];
```

	Test	Expected	Got	

✔	`int data[30];`
	`int* p = &data[17];`
	`printf("Index is %d\n", myIndex(data, p))`
		`Index is 17`	`Index is 17`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

String basics

C strings are introduced in the following three videos: 5.1 strings, 5.2 The String Library and 5.3 Arrays of Strings.

In C we use character arrays to implement strings - we've already seen some simple examples. A zero byte, written as either 0 or '\0', is used to terminate the string. The length of the string is not stored explicitly; it has to be determined by searching the character array until the first zero - the *terminator* - is encountered.

The C language lets us define *string literals* in our programs, e.g., in `printf("Hello world")`, but there is no explicit *string* type, no concatenation operator (like '+' in Python and Java), and no ability to dynamically create new strings from old strings. We have to do all the hard work ourselves, with the aid of a library of about 20 functions declared in `string.h`. Type `man string` in a terminal window for a summary of these functions. Also note that appendix D of King has an excellent summary of all the C library functions; look at the ones whose names begin with *str* for this lab.

When you read the documentation, you'll see frequent reference to a type `size_t`, denoting "size type". This is an integral type that's the same size as the addresses on the machine you're using. It isn't actually part of the base C language, but is defined as soon as you include any of the libraries; it's either `unsigned int` on a 32-bit machine or `unsigned long int` on a 64-bit machine. You should make a point of using `size_t` for buffer lengths because, at least in principle, on a 64-bit machine they could be longer than can be represented by a 32-bit `int`.

Representation of string literals

String literals are sequences of characters enclosed in double quotes. They can be used anywhere that a variable of type "pointer to char" can be used, except that they are placed in read-only memory and so cannot be altered.

Inspect the following program. Try to predict its output. Copy it to a file `stringliterals.c` and run it to confirm your predictions.

Hide `stringliterals.c` code

```
// Some examples of the uses of string literals
#include <stdio.h>

#define NUM_CHARS_TO_PRINT 20

/* printChars prints out its parameter using a printf with a %s format.
 * It then interprets the parameter as a char array, and prints
 * the first NUM_CHARS_TO_PRINT values of that array (both the character value
 * and the decimal integer value). Note that the string must be null-terminated.
 */
void printChars(char* s)
{
    int length = -1;
    printf("Parameter s = '%s'\n", s);
    printf("Now, as an array of individual chars ...\n");
    for (int i = 0; i < NUM_CHARS_TO_PRINT; i++) {
        printf("s[%d] = '%c' (%d)\n", i, s[i], (int) s[i]);
        if (s[i] == '\0') {
            length = i;
            break;
        }
    }
    if (length == -1) {
        printf("That string has %d or more characters.\n", NUM_CHARS_TO_PRINT);
    } else {
        printf("String length = %d\n", length);
    }
}

/* A "roll-your own" version of the puts ("put string") function.
 * Really just a simplified version of printChars above.
 * Note that the parameter type is char array, but that's equivalent
 * to char*
 */
void myPuts(char s[])
{
    int i = 0;
    while (s[i]) {
        putchar(s[i++]);
    }
}


// Another version of puts, using a char* parameter and pointer
// arithmetic.
void myPuts2(char* s)
{
    while (*s) {
        putchar(*s++);
    }
}


int main(void)
{
    char* s = "This is a string literal.\n";
    char s2[] = {'T','h','i','s',' ','i','s',' ','a','n',' ',
                 'a','r','r','a','y','\n','\0'};
    puts("'puts' prints a string. Its parameter is of type char*.\n");
    char s3[] = "This is an array too!\n";
    puts(s);  /* So we can pass in a variable of type 'char*' */
    puts(s2); /* ... or an array of chars... */
    puts(s3); /* ... and another array, with different initialiser syntax */
    puts(&s3[6]);  /* Think carefully about this one! */
    printf("s2 = %s\n\n", s2); /* We can format a string for output with %s */
    printChars("A test string");
    myPuts("A test string printed with myPuts\n");
    myPuts2("A test string printed with myPuts2\n");
    return 0;
}
```

Question **15**

Correct

Mark 1.00 out of 1.00

How is a function, whose single parameter is a character array containing a standard C string, able to determine the length of the string? [Note: more than one of the following may be correct. You must select all correct answers to get full marks.]

Select one or more:

- [x] a. It can search through the array looking for a null byte. The number of non-null characters before the null is the length of the string. ✔
- [ ] b. It can't determine the length of the string, which *must* be passed as an additional argument.
- [ ] c. The length of a string s can be obtained by the syntax s.length.
- [x] d. It can call the *strlen* function, passing the array as an argument. ✔
- [ ] e. The first byte of the array is the string length, i.e., if the array is *s*, s[0] is the length.

Correct

Marks for this submission: 1.00/1.00.

Question **16**

Correct

Mark 1.00 out of 1.00

What would the following bit of code do? [*Think* about it, don't just run it. Pretend you're in an exam.]

```
char* s = "Ummm: ";
while (*s) putchar(*s++);
printf("s[0] = %c", s[0]);
```

Note: this is horrible code. Do not copy it in subsequent questions!

Select one:

- ( ) a. Cause a compilation error.
- ( ) b. Print "Ummm: s[0] = U".
- (•) c. Print "Ummm: s[0] = ". ✔ The code compiles and executes just fine. Note how a string literal is assignable to a `char*` variable. When run, the while loop correctly prints out all characters up to but not including the null byte. Remember that a while loop iterates until the condition expression evaluates to zero, which in this case is when *s == 0, i.e. we've hit the terminator byte. Thus, when the loop exits, *s (which is equivalent to s[0]) is zero; printing a zero with a `%c` format produces no visible output.
- ( ) d. Print "s[0] = U".
- ( ) e. Fail with a segmentation fault.

Correct

Marks for this submission: 1.00/1.00.

Information

The string library

The library of functions declared in `string.h` simplifies the task of manipulating strings as character arrays. Copy the following code into a file *stringlibexamples.c*, read through it carefully, and predict what it will do. You will have to know what the individual string library functions do to understand the program. Then compile and run the program to see if you were right in your predictions.

Hide `stringlibexamples.c` code

```
// A few examples of what can be done with the string library.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* ss = "\101\141";

// Although the library contains a function strcasecmp it's not part of the
// C99 standard, so although it can be linked in just fine we have to
// declare its type ourselves ...
int strcasecmp(const char* s1, const char* s2);

#define BUFFSIZE 100
#define TERMINATORS " ()."   /* Token terminators */

// A function for you to ponder over!

/* Note that a "definite article" is the syntactic name given to the
 * word "the" in English (distinguishing it from the "indefinite
 * articles "a" and "an").
 */
void doStuff(char* s1, char* s2)
{
    char buff[BUFFSIZE];        /* workspace */
    char token[BUFFSIZE];
    int countDefiniteArticles = 0;
    int buffIndex = 0;
    int buffLen = 0;
    int tokenLen = 0;
    int len = strlen(s1);

    printf("The length of s1 = %d\n", len);
    strncpy(buff, s1, BUFFSIZE);
    strncat(buff, s2, BUFFSIZE - len - 1);
    printf("The concatenation of s1 and s2 is: %s\n", buff);

    buffLen = strlen(buff);  /* Length of concatenated string */
    buffIndex = 0;           /* Current index into the buffer */

    printf("\n%s\n", "The words in the concatenated string are:");

    // Loop to extract all words from the concatenated string

    while (buffIndex < buffLen) {
        tokenLen = strcspn(&buff[buffIndex], TERMINATORS);
        if (tokenLen != 0) {
            strncpy(token, &buff[buffIndex], tokenLen);
            token[tokenLen] = '\0';
            printf("%s\n", token);
            if (strcasecmp(token, "the") == 0) {
                countDefiniteArticles++;
            }
        }
        buffIndex += tokenLen + 1;  /* Step over token and its terminator */
    }
    printf("\n%d definite article(s) found.\n", countDefiniteArticles);
}

int main(void)
{
    printf("Length of s = %ld\n%s\n", strlen(ss), ss);

    doStuff("The quick (brown) fox ", " jumped over the moon.");
    return EXIT_SUCCESS;
}
```

Note that computing with strings is much harder in C than in Python or Java because all the string functions operate on character arrays of fixed size, and you must:

1. declare your own character arrays to receive the results of the various string functions,
2. make sure that your arrays are big enough to handle the computed results and
3. make sure that all your strings are always null-terminated (which won't be the case if, for example, `strncpy` reaches the specified maximum character count before the end of the source string).

Furthermore, if you make a mistake in C the result is liable to be a crash (segmentation fault) rather than a nice clean exception message.

Be aware that functions like `strcpy` and `strcat`, which copy string data without checking for buffer overflow, are dangerous and should never have been invented! You will be penalised for bad style if you use them in an assignment. You'll also increase the risk that your assignment code will crash with a segmentation fault when we test it with our tricky test data! However, even the versions with 'n' in them (`strncpy`, `strncat` etc) are easily misused if you pass in an incorrect length. For example, the length you pass to `strncat` is the maximum number of characters to copy, `not` the length of the destination buffer!

The requirement that you always use the 'n' version of string functions applies only to those functions that copy data, not to those that simply inspect data. For example, `strlen` and `strcmp` also have 'n' versions with an extra length parameter: `strnlen` and `strncmp`. In these cases the 'n' version exists mainly to provide extra functionality rather than to provide extra security. I'm happy with the use of `strlen` and `strcmp` in ENCE260.

Question **17**

Correct

Mark 1.00 out of 1.00

Which of the following statement is correct? You must choose all correct statements to get full marks.

Select one or more:

- [ ] a. The execution time of the *strlen* function is independent of the length of the string.
- [ ] b. The execution time of the *strlen* function increases as log(n), where n is the length of the string.
- [x] c. The *strlen* function returns the number of characters in a string, up to but not including the first null character. ✔
- [ ] d. The *strlen* function returns the zero-origin index of the last non-null character in a string.
- [x] e. The execution time of the *strlen* function increases linearly with the length of the string. ✔

Correct

Marks for this submission: 1.00/1.00.

Question **18**

Correct

Mark 0.90 out of 1.00

Match up the task to be performed on the left with the best choice of C string library function to use for the task, assuming the ENCE 260 style rules.

Compare two strings for equality.	strcmp ✔
Copy a string from one place in memory to another.	strncpy ✔
Find the first character that's *not* in a known set of characters.	strspn ✔
Extract a substring out of a given string to a given destination.	strncpy ✔
Determine the count of the number of characters in a string.	strlen ✔
Append one string to the end of another.	strncat ✔
Find the first character that is in a known set of characters.	strcspn ✔

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.90/1.00**.

Information

Reading strings

We have seen the use of *scanf* for reading numbers from standard input, but what about reading strings?

The previous lab showed how you can read characters into a *char* array using *getchar* in a **while** loop. The loop condition must check for three separate things:

1. A string terminator, such as the newline character.
2. End of file (EOF).
3. A full buffer.

The last of those is particularly important to avoid *buffer overrun vulnerabilities.* All code written in ENCE260 should be robust against buffer overruns.

There are (at least) three other ways of reading strings:

1. Using scanf with a "%ns" format, where *n* is at most one less than the buffer size. This reads up to but not including the first white space character or until *n* characters have been read or until end-of-file, whichever comes first. A terminating zero byte is then added (which is why the buffer size must be 1 more than *n*). For example:
   ```
   char token[81];
   scanf("%80s", token); // Read a token (word)
   ```
2. Using scanf with a "%nc" format, where *n* is the buffer size. This reads n characters or until end-of-file. No terminating zero byte is added. This is very rarely useful. If *n* is 1, use *getchar*.
3. Using the *fgets* function to read a single line of input. **NB: you *must* use *fgets*, never *gets*** which is deprecated, dangerous and evil, and has been removed altogether from the most recent standard.

If the input is line-oriented, the last of the above is preferred. You can then process the line with the usual string library functions or use *sscanf* (a variant of *scanf* that takes its input from a null-terminated string rather than standard input).

WARNING: If you use scanf with an *s* format you **must** specify the maximum field width (i.e. the maximum token size) or you are vulnerable to buffer overruns. For example "%80s" as above is safe, but "%s" is lethal!

Question **19**

Correct

Mark 1.00 out of 1.00

According to the above, which approach to reading input is most appropriate in each of the following situations?

Reading white-space separated numbers. ✔	scanf with %d, %f or %g format specifiers.
Reading lines of input. ✔	fgets
Reading input character-by-character ✔	getchar (or fgetc)
Reading white-space separated tokens (words). ✔	scanf with a format specifier of "%ns", where n is the maximum token length.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **20**

Correct

Mark 1.00 out of 1.00

Write your own version of the standard *strlen* function that returns the length of a string. Your function must have the signature *size_t mystrlen(const char s[])* and must not call any other functions, including of course the library *strlen* function!

There are two versions of this question, one requiring you to write *mystrlen* without any pointers (i.e., using array indexing) and the other requiring you to use **only** pointers (i.e., without using any array indexing). This is the first of these, meaning **your answer must not contain any occurrences of '*' or '&', including within comments.** Using array indexing rather than explicit pointers is generally preferred from a style standpoint, although most of the remaining questions in this lab will require the use of pointers in order to give you practice with them.

Note: *size_t* is a type used to represent the sizes of objects in memory. It is an alias for *unsigned long int* on a 64-bit machine and for *unsigned int* on a 32-bit machine. Since it's unsigned, *size_t* values can never be negative. When printing *size_t* values you should use the format specifier *%zd.*

Paste only your function definition into the answer box.

For example:

Test	Result
`printf("%zd\n", mystrlen("ENCE260"));`	`7`

Answer: (penalty regime: 0, 10, ... %)

```
size_t mystrlen(const char s[])
{
    int counter = 0;
    for (int i = 0; s[i] != '\0'; i ++) {
        counter += 1;
    }
    return counter;

}
```

	Test	Expected	Got	
✔	`printf("%zd\n", mystrlen("ENCE260"));`	7	7	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **21**

Correct

Mark 1.00 out of 1.00

Write your own version of the standard *strlen* function that returns the length of a string. Your function must have the signature *size_t mystrlen(const char* s)* and must not call any other functions, including of course the library *strlen* function!

There are two versions of this question, one requiring you to write *mystrlen* without any pointers (i.e., using array indexing) and the other requiring you to use **only** pointers (i.e., without using any array indexing). This is the second of these, meaning **your answer must not contain any occurrences of '[', including within comments.**

Note: *size_t* is a type used to represent the sizes of objects in memory. It is an alias for *unsigned long int* on a 64-bit machine and for *unsigned int* on a 32-bit machine. Since it's unsigned, *size_t* values can never be negative. When printing *size_t* values you should use the format specifier *%zd.*

Paste only your function definition into the answer box.

For example:

Test	Result
`printf("%zd\n", mystrlen("ENCE260"));`	`7`

Answer: (penalty regime: 0, 10, ... %)

```
size_t mystrlen(const char* s)
{
    const char* pointer1 = s;
    //for (;
    int counter = 0;
    for (; *pointer1 != 0; pointer1++) {
        counter += 1;
    }
    return counter;
}

```

	Test	Expected	Got	
✔	`printf("%zd\n", mystrlen("ENCE260"));`	`7`	`7`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **22**

Correct

Mark 1.00 out of 1.00

Write **your own** version of the standard library *strrchr* function, with the signature `char* mystrrchr(char* s, int c)`. Note the extra 'r' in the name; you're replicating the *strrchr* function, not the *strchr* function.

The function should return a pointer to the **last** occurrence of the character c in the string s or NULL if no matching character exists. **Your implementation must use only pointer operations with no array indexing**. Although in general using array indexing is fine (preferred, even), this is an exercise in using pointers. Your code will be rejected if it contains any occurrences of '[', even in comments. Your function must not call any other functions, including of course the library *strrchr* function! It also must not use any #define, #include or other preprocessor commands.

Make sure you name your function *mystrrchr* not *strrchr* or *mystrchr.*

NOTE: the test case(s) below print the integer obtained by subtracting the string start address from the function return value (unless the latter is NULL). The integer obtained in this way is the index into the string. It is printed using a "%zu" format in order to ensure it behaves the same on both 32-bit architectures (CodeBox) and 64-bit architectures (the quiz server).

For example:

Test	Result

`char* s = "ENCE260";`
`char* foundAt = mystrrchr(s, 'E');`
`if (foundAt == NULL) {`
`  puts("Not found");`
`}`
`else {`
`  printf("%zu\n", foundAt - s);`
`}`
	3

Answer: (penalty regime: 0, 10, ... %)

```
1
2  char* mystrrchr(char* s, int c)
3  {
4      char* lastchar = 0;
5      int counter = 0;
6      for (; *s != 0; s++) {
7          if (*s == c) {
8              lastchar = s;
9          }
10         counter += 1;
11     }
12     return lastchar;
13
14
15
16 }
17
18
19
```

	Test	Expected	Got	

✔	`char* s = "ENCE260";`
	`char* foundAt = mystrrchr(s, 'E');`
	`if (foundAt == NULL) {`
	`  puts("Not found");`
	`}`
	`else {`
	`  printf("%zu\n", foundAt - s);`
	`}`
		3	3	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **23**

Correct

Mark 0.70 out of 1.00

Write a function *int tokenCopy(char* dest, const char* src, int destSize*) which copies characters from the given source string *src* into the given destination buffer *dest*, which is of size *destSize,* until either:

- the end of the source string occurs, or
- the destination buffer is full (allowing for the terminator that will be needed), or
- a space character is found in the input string

whichever comes first. If the copying finishes because a space is found, the space is not copied. The destination string must always be correctly terminated. If there is insufficient space in the destination for the source string, the destination string must be a correctly terminated leading substring of the source string.

The return value is the number of characters copied, not including the terminating null byte.

To give you more practice with pointers, **the function must be implemented without using array indexing,** i.e. the characters '[' and ']' must not appear anywhere within your answer, including in comments. Please realise that normally an array indexing approach is just fine, and is probably preferred over the use of pointers.

Your function is not permitted to call any other functions, particularly the string library functions.

For example:

Test	Result

`char buff[5];`
`int n = tokenCopy(buff, "This is a string", 5);`
`printf("%d '%s'\n", n, buff);`
	`4 'This'`

`char buff[3];`
`int n = tokenCopy(buff, "This is a string", 3);`
`printf("%d '%s'\n", n, buff);`
	`2 'Th'`

Answer: (penalty regime: 0, 10, ... %)

```
int tokenCopy(char* dest, const char* src, int destSize)
{
    int counter = 0;
    //pointer pointing to start of string src to destSize
    while ((counter < destSize -1) && (*src != 32) && (*src != '\0')) {
        //printf("%d, %p\n", *src, dest);
        //printf("%d hihiih\n", *src);
        *dest = *src;
        counter += 1;
        dest++;
        src++;
    }
    *dest = '\0';
    //printf("%d, %p\n", *src, dest);
    return counter;
}
```

	Test	Expected	Got	

✔	`char buff[5];`
	`int n = tokenCopy(buff, "This is a string", 5);`
	`printf("%d '%s'\n", n, buff);`
		`4 'This'`	`4 'This'`	✔

✔	`char buff[3];`
	`int n = tokenCopy(buff, "This is a string", 3);`
	`printf("%d '%s'\n", n, buff);`
		`2 'Th'`	`2 'Th'`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.70/1.00**.

◄ Quiz3: Arrays and Functions Jump to... Quiz5: Code and Data Structures ►