COMP9024 24T1 Data Structures and Algorithms

Week 2 Tutorial

Elementary Data and Control Structures in C

[Show with no answers] [Show with all answers]

1. (Tutor introduction)
   a. What is your tutor's name and e-mail?
   b. How long have they been at UNSW?
   c. What are they studying?
   d. What is one interesting thing about them?

[show answer]

2. (Class introduction)
   - What are your classmates' names?
   - What are they each studying?
   - How many of them are in their first term at UNSW?
   - How many of them have programmed in C before?
   - What is one interesting thing about them?

[show answer]

3. (Course resources)
   a. How can you access the course site?
   b. Where can you find the lecture recordings?
   c. How can you join the course forum?

[show answer]

4. (Structures)

Suppose you want to write a C program to manage your contacts. Each contact has a name, a phone number, an address, a date of birth, and an email address. You decide to define a structure to represent a contact such that:

- The name is a nested structure with first name and last name fields. These are strings of up to 20 characters each.
- The phone number is a string of up to 10 characters.
- The address is a nested structure with street number, street name, suburb, state, and postcode fields. The street number and postcode should be short integers, while the street name, suburb, and state should be strings of up to 30, 15, and 3 characters, respectively.
- The date of birth is a nested structure with day, month, and year fields. These should be short integers.
- The email address is a string of up to 30 characters.

Write a C structure to represent a contact. All structures should be defined as new data types, and any character arrays should allow for end-of-string characters.

[show answer]

5. (Bracket matching)

Recall the bracket matching algorithm from the week 1 lecture. Consider the following stream of characters as input:

```
bool f(char a[], int n) { for (int i = 0; i < n; i++) { if (a[i] == 0) { return true; } return false; }
```

Trace the execution of the algorithm and determine the state of the stack at the point when the algorithm terminates (either with success or with failure).

[hide answer]

Next char	Stack	Check
-	empty	-
(	(	-
[	([	-
]	(	[vs] ✓
)	empty	(vs) ✓
{	{	-
(	{(	-
)	{	(vs) ✓
{	{{	-
(	{{(	-
[	{{([	-
]	{{(	[vs] ✓
)	{{	(vs) ✓
{	{{{	-
}	{{	{ vs } ✓
}	{	{ vs } ✓
eof	{	✗

When the end of the input stream is reached the stack still contains one element, "{", which is unmatched.

6. (Recursion)

Consider the following procedure `unknown`, that takes as input an array $A[l..r]$ of $n = r - l + 1$ numbers, with the left-index l and the right-index r:

```
unknown(A,l,r):
|  if l > r then
|     return −∞
|  else if l = r then
|     return A[l]
|  else
|     q = l + ⌊(r−l)/3⌋
|     ansL = unknown(A, l, q)
|     ansR = unknown(A, q+1, r)
|     if ansL > ansR
|        return ansL
|     else
|        return ansR
|     endif
|  endif
```

Let $A[1..8] = \{6, 4, 2, 9, 2, 8, 6, 5\}$.

What does a call to `unknown(A,1,8)` return?

[hide answer]

9 (the maximum value in A)

7. (String length)

Recall from the week 1 lecture that, in C, a string is an array of characters with a null character `'\0'` at the end. The standard C library includes a suite of functions for dealing with strings. These are declared in the header file `string.h`, which we can `#include` in our programs to then use.

One of the most commonly used string functions is `strlen`, which takes one argument, a pointer to the first character of a string, and returns the length of the string (not including the null character).

a. In computer programming, particularly in C, you'll often here references to "undefined behaviour". This is where the result of executing a program is unpredictable.

How might using `strlen` lead to undefined behaviour?

b. Suppose for this week's practical exercise you are tasked with writing your own version of `strlen`, called `mystrlen`. You want to make `mystrlen` available to programs by providing its interface in a header file, which will look like this:

`mystrlen.h`

```
#define MAXLEN 128

/**
 * @brief Calculates the length of the string.
 *
 * The mystrlen() function calculates the length of the string pointed to by s,
 * excluding the terminating null byte ('\0').  If no null byte is found within
 * the first MAXLEN bytes of the string, the function returns MAXLEN.
 *
 * @param s Pointer to the first character of a string.
 * @return  The length of the string s, or MAXLEN if no null byte found within
 *          MAXLEN characters.
 */
int mystrlen(char *s);
```

To begin development you sketch out a skeleton for your source code as below. It includes the function you're implementing, `mystrlen`, and a `main` function to test it.

`mystrlen.c`

```
#include "mystrlen.h"

int mystrlen(char *s) {
   // your code here to implement mystrlen

}

int main(void) {
   // your code here to test mystrlen
}
```

Once you've implemented `mystrlen` and finished testing it, you want to make it available to other programs. For example, you write a program in a file called `lencalc.c` that repeatedly prompts the user to enter a string, and each time prints the length of that string, using your `mystrlen` function. The code skeleton looks like this:

`lencalc.c`

```
#include <stdio.h>
#include "mystrlen.h"

int main(void) {
   char s[MAXLEN];
```

```
    int len;

    // your code here to repeatedly prompt the user for a string
    // and print its length using mystrlen

    return 0;
}
```

Once you've completed `lencalc.c` correctly, you attempt to compile it with the following command, but the compiler reports an error:

```
prompt$ dcc -Wall -Werror -std=c11 -o lencalc lencalc.c
```

Why won't the program compile, and how can you fix it?

Note: This tutorial exercise does **not** require you to write any code. That is left as a practical exercise.

c. Suppose that, as it turns out, this week's practical was actually split into two exercises:
   i. Write a library function, `mystrlen`, to calculate the length of a string.
   ii. Write a program to prompt the user to enter strings and print their lengths, as reported by `mystrlen`.

These two exercises will be assessed separately. First, your library function will be tested using a program we've written. Second, your program will be tested using a library function we've written.

You've completed both exercises, and you're satisfied with your solutions, so it's time to submit. You have a CSE terminal open, with your code in the current working directory, and you enter the following commands:

```
prompt$ give cs9024 week2 mystrlen.c
prompt$ give cs9024 week2 lencalc.c
```

What is the problem with this submission approach, and how can you fix it?

[hide answer]

a. `strlen` will continue to read memory until it finds a null character, so if the string is not null terminated then the behaviour is undefined. It may return the expected length, some garbage value, or it may attempt to access memory illegally and crash.
b. There are actually two issues here:
   i. An undefined reference to `mystrlen`. While the compiler knows the expected interface of `mystrlen` through `mystrlen.h`, which is included in `lencalc.c`, it doesn't have the actual `mystrlen.c` implementation. We need to explicitly compile and link `mystrlen.c` and `lencalc.c` together.
   ii. Both `lencalc.c` and `mystrlen.c` contain a `main` function. The `main` function is a special function in C. It is the entry point of a program, and a program must have exactly one `main`.

We can solve the first problem by including `mystrlen.c` in the compilation command:

```
prompt$ dcc -Wall -Werror -std=c11 -o lencalc lencalc.c mystrlen.c
```

This is typical for small projects, but for larger projects it is common to compile each `.c` file separately and then link them together. This is done by compiling each `.c` file to a `.o` object file, and then linking the object files together. For example:

```
prompt$ dcc -Wall -Werror -std=c11 -c mystrlen.c
prompt$ dcc -Wall -Werror -std=c11 -c lencalc.c
prompt$ dcc -o lencalc mystrlen.o lencalc.o
```

You'll also notice that we didn't specify any `.h` header files. The compiler only needs to be told which `.c` source files to compile. It will then look for any relevant `.h` header files based on the `#include` statements in those `.c` source files.

We can solve the second problem by deleting the `main` function from `mystrlen.c`. In testing the implementation, it would actually be better to separate that `main` into its own `.c` file (e.g. `mystrlenTester.c`), and in that file, `#include "mystrlen.h"`.

c. The submission of `lencalc.c` will overwrite the submission of `mystrlen.c`. To fix this, we should submit the two files together in a single command:

```
prompt$ give cs9024 week2 mystrlen.c lencalc.c
```

8. **Public service announcement**

Every term we have students who submit code for assessment that fails to compile or run when being tested on the CSE servers. This can be for many reasons, but unfortunately the end result is the same -- those students' marks suffer.

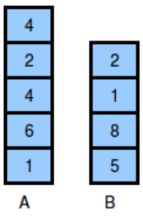
No matter your workflow, no matter your development environment, no matter your level of experience, it is **essential** that you test your code on the CSE servers before submitting it for assessment. This is the only way to ensure that your code is tested in the same environment as the assessment scripts.

Please give yourself the best chance for success and don't be one of the few who get caught out.

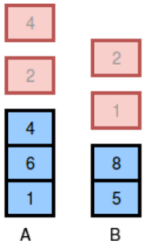
9. **Challenge Exercise**

Suppose that you are given two stacks of non-negative integers A and B and a target threshold $k \geq 0$. Your task is to determine the maximum number of elements that you can pop from A and B so that the sum of these elements does not exceed k.

Example:



Maximum number of elements that can be popped without exceeding k = 10 is 4:



If k = 7, then the answer would be 3 (the top element of A and the top two elements of B).

a. Write an algorithm (in pseudocode) to determine this maximum for any given stacks A and B and threshold *k*. As usual, the only operations you can perform on the stacks are pop() and push(). You *are* permitted to use a third "helper" stack but no other aggregate data structure.

b. Determine the time complexity of your algorithm depending on the sizes *m* and *n* of input stacks A and B.

Hints:

- A so-called greedy algorithm would simply take the smaller of the two elements currently on top of the stacks and continue to do so as long as you haven't exceeded the threshold. This won't work in general for this problem.
- Your algorithm only needs to determine the number of elements that can maximally be popped without exceeding the given *k*. You do not have to return the numbers themselves nor their sum. Also you do not need to restore the contents of the two stacks; they can be left in any state you wish.

[hide answer]

```
MaxElementsToPop(A,B,k):
   Input  stacks A and B, target threshold k≥0
   Output maximum number of elements that can be popped from A and B
          so that their sum does not exceed k

   sum=0, count=0, create empty stack C
   while sum≤k and stack A not empty do   // Phase 1: Determine how many elements can be popped just from A
      v=pop(A), push(v,C)                 //          and push those onto the helper stack C
      sum=sum+v, count=count+1
   end while
   if sum>k then                          // exceeded k?
      sum=sum-pop(C), count=count-1       //   then subtract last element that's been popped off A
   end if
   best=count                             // best you can do with elements from A only

   while stack B not empty do
      sum=sum+pop(B), count=count+1       // Phase 2: add one element from B at a time
      while sum>k and stack C not empty do   // and whenever threshold is exceeded:
         sum=sum-pop(C), count=count-1       //   subtract more elements originally from A (now in C)
      end while                              //   to get back below threshold
      if sum≤k and count>best then
         best=count                       // update each time you got a better score
      end if
   end while
   return best
```

Time complexity analysis: In phase 1, the worst case is when all *m* elements in stack A need to be visited, for a maximum of *m*+1 pop and *m* push operations. In phase 2, the worst case is when all elements have to be taken from both B and C, for a maximum of *m*+*n* pop operations. Assuming that push() and pop() take constant time, the overall complexity is O(m+n). This makes it a linear-time solution.