COSC 2P95 – Lab 5 – Section 01/02 – Structures and Allocation

(Do I need to keep telling you to use Linux? No? Good)

Note: so we can focus on what's really important, it's assumed that you have at least a cursory understanding of our last week's lecture material. In particular, we won't bother rehashing enumerations, unions, or void pointers. Please contact your instructor if there were any parts of those topics that you'd like clarified.

For this lab, we'll be focusing on two major topics: how to create a record, and how to allocate memory.

Structures

One of the most basic requirements of effective data processing is the ability to represent a *record*. There are many ways to achieve this, but a `struct` is a pretty effective option. We already know what a struct is, so let's just make sure we remember how to declare them.
Start by creating some `.cpp` file, **and an accompanying `.h` header file**.
In the header file, declare the following (after your `#include`'s, and possibly namespace line):

```
struct PRecord {
	long time;
	string entry;
};
```

(You'll also be expected to include function prototypes in this file as well, but it won't help us behaviourally)

How do we declare such a struct? Easy!
In your `main` function, declare the following:

```
PRecord single={23378L,"Simple example"};
```

How do we access a member of the struct? To answer that, let's write a function that displays the contents:

```
void displayRecord(PRecord pr) {
	cout<<"Time: "<<pr.time<<"\tEntry data: \""<<pr.entry<<'"'<<endl;
}
```

(Yes, I know this is more hand-holdy than usual. This first part is just foundation; the actual work comes later)

Just for ha-ha's, call the `displayRecord` procedure, giving it your record, `single`.
As you can see, accessing a struct's members is as simply as using the . notation.
The question is, did `displayRecord` receive the actual struct, or a copy of it? Let's find out!

```
void naiveModify(PRecord pr) {
	pr.time=33;
}
```

Call that procedure, passing along your record, and then invoke the display procedure again. What do you see?

The reason it doesn't change is that structs are, indeed, *copied* into parameters. If we wanted to be able to mutate the original record, this is a problem (also, for larger structs, this can be costly). One solution is references:

```
void refModify(PRecord &pr) {
	pr.time=33;
}
```

Note that accessing the member uses the same notation as before. That's because references aren't anything particularly special; they simply allow the parameter and the original variable to share the same place in memory.

In a vacuum, this solution would arguably be better than using pointers; however, structs often go hand-in-hand with dynamic allocation, which tend to already be using pointers anyway, so it doesn't hurt to take a look:

```
void ptrModify(PRecord *pr) {
     pr->time=11;
     (*pr).entry[0]='P';
}
```

Notice that there are two ways to access the members of a struct referenced by a pointer. Typically we just use the -> notation, but that's actually shorthand for *dereferencing*, followed by normal . member access.
(Both are shown in the example)

Of course, you **could** combine both pointers and references, but that's less common (basically, don't do it unless you have a reason to do it).

After each of these, display the struct again, to verify that you understand the basics.
Next, let's make some more additions...

Holding multiple records

An array is expected to hold a homogeneous type. However, there's no rule that type needs to be a primitive.
An array of structs is perfectly reasonable.

For example, building off the previous example:

```
PRecord multiple[]={{44L,"First"},{55L,"Second"},{66L,"Third"},{77L,"Fourth"}};
displayRecord(multiple[0]);
```

In this case, the display function doesn't even know that the struct is in an array.
What about trying to modify the contents of one of the elements? Do we still have to worry about copying?
Try passing the entire array into this:

```
void arrModify(PRecord *pr) {
     pr[0].time=100L;
     pr->entry[1]='a';
}
```

You might be wondering, what happens if we change the parameter to an array?
Nothing.
It's the same.

Strictly speaking, arrays and pointers aren't the same thing, but arrays have a tendency to *decay* into pointers.
However, note that, when using it in an array-notation, we access the member with a ., but in the pointer notation, we use ->.

Okay, so we can now store multiple records.
However, outside of some batch data processing, that's still not going to cover the major use cases.
Sometimes, we just don't know how many records we'll be holding.
Similarly, sometimes we need to make our own data structures, and that requires allocating new structs on the fly. We need a mechanism for requesting arbitrary additional memory.

Dynamic memory allocation from the heap

We've already addressed this in lecture, but it warrants revisiting.
Local variables are kept on the stack. They're allocated when you enter a function, and deallocated when you leave. But what do you do if you need something longer than that? Or if you don't have any way of expressing the space requirements before runtime?

The *heap* is another portion of memory, used to provide additional memory when needed.
One thing to remember is that, since we're losing the automatic deallocation, when you request memory, you need to (eventually) give it back. Otherwise, you'll end up with a *memory leak*.

Let's start small – allocating a single struct:

```
PRecord *rec=new PRecord;
rec->time=123456L;
rec->entry="Heyooo";
displayRecord(*rec);
```

When we're done with this record, cleanup is easy:
`delete rec;`

Note: Once you've deleted allocated memory, ***do not try to access it again***!
Maybe it'll work, maybe it won't. Maybe it's a dangling pointer. It's not uncommon to explicitly reset pointer values to, for example, NULL.
Of course, that increases the chances of a crash, but that's what we want. If we're using memory that the system thinks is empty, then we very much want it to shout at us.

Let's try allocating an integer array:
`int *arr=new int[32];`

As a reminder, when you're done with this array, the standard delete keyword isn't quite sufficient. Instead:

```
delete[] arr;
```

Of course, it should be clear to see how we can combine these concepts: dynamically allocating an array of structs is trivial.

However, there's one very important use for dynamic allocation we haven't addressed much yet: dynamic data structures, like linearly linked lists.

Linked Lists

We've already briefly discussed a linked list in lecture, and you've been provided with some sample code.
Let's revisit it, shall we?

First, we want a header file. After whatever inclusions/namespace lines you might need, add the following:

```
struct node {
      long item;
      node *next;
};

void push(const long &i, node* &n);

long pop(node* &n);
```

Then, in your main source file, add:

```
void push(const long &i, node* &n) {
      node *nn=new node;
      nn->item=i;nn->next=n;
      n=nn;
}

long pop(node* &n) {
      long fr;
      node *ptr=n;
      n=n->next;
      fr=ptr->item;
      delete ptr;
      return fr;
}

int main() {
      node *head;
      push(13,head);
      push(10,head);
      push(18,head);
}
```

Before we get any further, it's worth pointing out two things:

1. The code example you received from lecture was templated, to be more generalized
2. If the node were to hold anything more complicated, it would be in the form of a pointer to the actual record (this is actually the normal way of doing it). e.g. a pointer to another struct of some sort

What we've defined here is a *stack*.
A stack is one of the fundamental building-block collections of computer science.
Stacks are First-In-Last-Out (FILO), or Last-In-First-Out (LIFO). In this case, that means that, since the numbers were pushed in the sequence of 13, 10, 18, they'll come out in the sequence of 18, 10, 13.
You can verify this through three calls to `pop`; printing the results.

We achieved this through *insertion at the front* into the linked list. There are other forms of linked list manipulation; namely *insertion at the end*, and *insertion at the middle* (which generally assumes some predictable sequencing, such as sorted entries). Similarly, we're also employing *removal from the front*.

In a proper, more full-fledged linked list, you'll typically assign some form of terminator value to the →next pointer of the final struct, such as `NULL` (this allows you to easily test whether or not you've reached the end of the list during traversals, or to check if it's empty when at the front).

Of course, a stack can also be implemented via an array (sometimes referred to as a *contiguous* implementation, since all of the entries are back-to-back). Simply use a counter to keep track of the 'top' position; increment for additions, and decrement for removals.
Naturally, a real stack would also require some basic sanity-checking, to avoid crashes, etc.

Queue

A *queue* is another fundamental data structure of computer science.
Elements are added/removed in a First-In-First-Out (FIFO) sequence.
As with stacks, they may be linked or contiguous.
The linked implementation requires traversing to the end of the list to enqueue new entries, while dequeueing can be accomplished by simply removing from the front.
The contiguous version requires some clever counters (that can loop around the array boundaries).

Submission Task

For your submission task, all you need do is to write a simple queue.
It may be either linked or contiguous, and may hold whatever type you choose (I'd suggest something simple).
Be careful to avoid *memory leaks*!

Write a simple program to demonstrate it however you see fit.

Note: You don't need to go overboard here. This is simply practice for your next exercise/assignment.

Submission

To demonstrate that you understand the content from this lab, simply show a brief demonstration of your queue to your lab supervisor.