



UNIVERSITY  
*of York*

SCHOOL OF PHYSICS,  
ENGINEERING AND TECHNOLOGY

# Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret



# Laboratory 4

## Data Storage in C

4.1	Aims . . . . .	43
4.2	C99 integer types . . . . .	44
4.3	Structures . . . . .	45
4.3.1	Padding . . . . .	45
4.3.2	Declaration, initialisation and assignment . . . . .	46
4.3.3	Pointers to <code>structs</code> . . . . .	47
4.4	Arrays as Stacks, Queues and Heaps . . . . .	48
4.4.1	Stacks . . . . .	48
4.4.2	Circular Buffers . . . . .	49
4.4.3	Binary Heaps . . . . .	51
4.4.4	A Generic Heap: Function Pointers . . . . .	52
4.5	Linked Lists . . . . .	55
4.5.1	A circular buffer . . . . .	55
4.6	Summary and Conclusions . . . . .	56
	<b>Appendices</b> . . . . .	57
4A	Void Pointers . . . . .	57
4A.1	Pointers-to-Pointers-to-Void . . . . .	57

### 4.1 Aims

In this laboratory session, you will:

- Experiment with structures in C;
- Use pointers to access data indirectly;
- Make use of stacks and heaps to store data.



Ensure that the ARM board is connected to your PC, and Launch  $\mu$ Vision and CoolTerm as before.

Copy your project files from your H: drive, or wherever you put them last week, to your local C: drive, perhaps in C:\tmp. Remember that the debugger may not function correctly if you don't do this.

## 4.2 C99 integer types

❗ In the most recent video you were introduced to the C99 integer types and the `stdint.h` header file. From this point onwards **you will be expected to use these**. The only native C types you should ever use are:

- `char`, but only ever for ASCII character data
- `float` and `double`

The most useful C99 types consist of:

- The **fixed-width** types, `intx_t` and `uintx_t`, which *may* exist for  $x = 8, 16, 32, 64$  (and probably will, on all “normal” platforms)
- The **fast** types, `int_fastx_t` and `uint_fastx_t`, guaranteed to exist for  $x = 8, 16, 32, 64$ ; these may be larger than the requested size, and will be of a size that allows for fast arithmetic on the target platform
- The **integer pointer** types, `intptr_t` and `uintptr_t`, intended to be integer types capable of holding pointer values

If you want to print C99 integer types using `printf()`, the format specifiers are given in Table 4.1.

Format	[u]intx_t	[u]int_fastx_t	[u]intptr_t
Signed decimal	PRIdx	PRIdFASTx	PRIdPTR
Unsigned decimal	PRIdx	PRIdFASTx	PRIdPTR
Unsigned octal	PRIOx	PRIOFASTx	PRIOPTR
Unsigned lowercase hex	PRIx	PRIXFASTx	PRIXPTR
Unsigned uppercase hex	PRIXx	PRIXFASTx	PRIXPTR

Table 4.1: Format specifiers for C99 integer types. ‘x’ should be replaced by the width of the type (8, 16, 32, 64).

You will need to include `<stdint.h>` to use the C99 types, and `<inttypes.h>` to use the format specifiers. Remember that the format specifiers must be used *outside* the double-quotes, e.g.

```
printf("Value is %" PRId32 "\r\n", value);
```

## 4.3 Structures

You should know about C's `struct` keyword, and how it can be used with `typedef` to create new data types that can contain other items of data.



Use Project→Open Project... to open `Storage\structs\structs.uvprojx`.

Have a look at the source code. There is a `struct` type declared, and a variable of this type is declared in `main()`.

Build this code, and observe the warnings. There is one about the unused `test` variable, which we can ignore for now because the code is incomplete. But the other is interesting:

```
src/main.c(6): warning: padding struct 'test_t' with 3 bytes to
      align 'y' [-Wpadded]
      int32_t y;
              ^
```

Let's investigate what this means.

### 4.3.1 Padding



Add a line to print out the size of the `test` variable, or equivalently the size of the `test_t` type. Build and run your code. What result do you get? Is it consistent with what you'd expect, given the sizes of the fields of `test_t`?

**i** The structure has been *padded* by the compiler, in order to align the 32-bit `y` field onto a word boundary. This can help significantly with performance. The structure therefore consists of three parts:

- The 8-bit (one byte) `x` field;
- Three bytes of *padding*, as indicated by the warning;
- The 32-bit (four byte) `y` field.

Normally, this padding is not a problem; many project authors will elect to disable the warning globally, and most compilers won't issue this warning at all unless it's specifically enabled. But padding can sometimes be very important, and it's a good idea to be aware when padding has been applied. This is particularly true when interoperating between C and assembly language, because it's essential to know exactly where in a struct each field is located. We'll see more about that soon.

**i** Note that the warning will still be issued even if you swap the `x` and `y` fields. This is because although the fields are now in their expected locations, the struct is still padded so that its size is a multiple of 32 bits, and this could be important if you were to create an array of them, for example.

To suppress the warning, there are several options:

- The warning could be forcibly disabled, of course. This is not recommended.
- You could insert extra dummy fields to control the padding manually.
- You could instruct the compiler not to pad the structure.



Try adding the field

```
uint8_t _padding[3];
```

to the structure, between the `x` and `y` fields. Build the code and confirm that the warning has gone. Run it and check that the size of the structure is the same as before.



Now remove the extra padding. Above the structure declaration, add the two lines

```
#pragma pack(push)
#pragma pack(1)
```

and below it, the single line

```
#pragma pack(pop)
```

Build and run the code. You should find that the warning has gone, but also that the structure size has changed because the compiler has not added padding as before.

**i** `#pragma` is a preprocessor directive, but confusingly its role is to control the behaviour of the compiler. Its operation is *not* specified by the C standards, and is explicitly implementation defined. However, the semantics of `#pragma pack` are very stable and supported across all major toolchains.

**!** **Manual padding is a better option than packing in most cases.** Structure padding is added for performance reasons and its impact can be quite significant. Use packing only when it is essential that padding is not added, or to create code that is more portable because it is not sensitive to different alignment standards on different compilers or platforms.

### 4.3.2 Declaration, initialisation and assignment

**i** Recall that it is best practice to initialise all variables when they're declared, whether or not sensible values are known for them at the time. So, how are structures initialised?

## Designated initialisers

In C99, the ability to initialise structures using field names was introduced. This should always be used if C99 or later is in use (which it should be — 1999 was a while ago now — but you’d be surprised!). The syntax is known as a **designated initialiser**.

Fields are prefixed by dots, just as they would normally be when accessing a field in a structure, and field assignments are separated by commas. For example:



```
test_t test = { .x = 100, .y = -1 };
```

Newlines can be used to make long initialisations more readable:

```
test_t test = {  
    .x = 100,  
    .y = -1  
};
```

Designated initialisers can even be nested, if one structure contains another as a field.



Try setting the values of `test.x` and `test.y` before using a `printf()` call to print them out, and check that the values you have set are successfully reported. Now try using a designated initialiser instead, and check again that the values you have specified are successfully reported.

## Compound literals

Also introduced in C99 was a syntax for structure literals, known as **compound literals**.



This is generally similar to the designated initialiser, but with one major difference: a compound literal requires a type cast to indicate the type of the literal.

For example:

```
test = (test_t) { .x = 100, .y = -1 };
```



Leave the initialisation in place, and use a compound literal to subsequently assign different values to the structure’s fields.

### 4.3.3 Pointers to structs

As you have seen in videos, the access syntax for the fields of a structure is different if you have a *pointer* to a structure.



Declare a new variable, `test_p`, of type ‘pointer-to-`test_t`’ (you’ll need an asterisk in the variable declaration). Use the referencing operator to make `test_p` point to `test`. Now, following the existing `printf()` call, insert the lines

```
test_p->x = 5;  
test_p->y = 20;
```

Now duplicate the `printf()` line, so that it prints out the values of the `test` fields again afterwards. Verify that the values change as you'd expect.

## 4.4 Arrays as Stacks, Queues and Heaps

In this section we will use arrays as simple data structures, to create a *stack*, a *queue* (implemented as a circular buffer), and a *heap*.

### 4.4.1 Stacks



Use Project→Open Project... to open `Storage\stack\stack.uvprojx`. In `main.c` you will notice that there is an array called `stack` declared, big enough to store 10 integers. There is also a pointer, `stack_p`, that is initialised to point to the start of the array.

Your task is to write two functions. You'll find them both in `stack.c`. Their prototypes are:

```
void push(int32_t **sp, int32_t value);
int32_t pop(int32_t **sp);
```

**i** Note that the first argument to each function is not the stack pointer: it is a *pointer* to the stack pointer.

The idea that you can pass a pointer to a value into a function to allow the function to modify that value *in situ* is a common pattern and we've covered it previously, but the fact that the value in question was *already* a pointer leads to this confusing double-pointer. The use of a double pointer here is logical and correct, but there is no denying that it does not aid readability, even if you understand completely how and why it works.

**!** If you don't understand this, please ask before you go any further!

The `push()` function must:

- Store the supplied value in the stack by storing to `**sp` (remember, `sp` is a pointer-to-pointer-to-**int**, so `*sp` is a pointer-to-**int**, and `**sp` is an **int**);
- Increment the stack pointer by incrementing `*sp`

The `pop()` function must read the next value from the stack, and decrement the stack pointer.



Which order should these operations occur in?



Implement and test your code. Experiment with your new stack. Use the line

```
push(&stack_p, 5);
```

to push a value to the stack. Now check that it has worked by popping it and printing it out:



```
printf("Popped value: %d\r\n", pop(&stack_p));
```

Try pushing and popping more values. What happens if you push more than 10? Try writing a loop to push, say, 100 values. What happens?

⚠ By pushing more values to the stack than it can hold, your code has started writing to other parts of the CPU's memory. The effect of this is completely unpredictable. It is a condition known as *stack overflow*, and it has been the cause of innumerable bugs and security flaws in code across the world. We'll explore this a little more another time.

## 4.4.2 Circular Buffers

Queues, or First-In First-Out (FIFO) buffers, are useful whenever you need to process items of data in the order in which they arrive but you need to store them for a while first. Queues are also useful for storing a set of data items that need to be repeatedly iterated in order; items are dequeued, processed, and then immediately enqueued again – or in some cases, as you'll see, the queue can be built such that this enqueue/dequeue can be done implicitly. The most common queue implementation is the *circular buffer*.



Use Project→Open Project... to open `Storage\queue\queue.uvprojx`.

This time the project is already almost complete. It is a very simple implementation of a 10-element queue using a circular buffer. The main points to notice are:

- The function `queue_put()` is used to add items to the queue. It does two things:
  - The item is stored in the array at the location given by the `insert` field;
  - The `insert` field is increased by one, modulo-10.<sup>2</sup>
- The function `queue_get()` removes items from the queue. It also does two things:
  - The item stored in the array at the location given by the value of the `retrieve` field is extracted;
  - The `retrieve` field is increased by one, modulo-10;
  - The extracted value is returned.

⚠ There are some ways in which this implementation is not well written. In particular the queue structure contains the data store. This means:

- The size of the store is fixed and must be known at compile-time, so if multiple queues are created they will all be the same size;
- The data type is also fixed.

However, there are some aspects of good practice that can be identified too:

---

<sup>2</sup>The `%` operator in C is the *modulo* operator – it gives the remainder after a division. In this case it's used to ensure that the `insert` variable does not increment beyond the array length.

- A structure is used to hold all relevant information pertaining to an individual queue;
- The queue size is set using a preprocessor constant, so it can be modified easily;
- An initialiser is supplied for the `queue_t` type so that it can be initialised in a consistent way that is not related to the implementation.



Try building and running the code.

- Does it work as you expect?
- Try putting more than 10 items into the queue and then getting them out again. What happens? Why?



Queue implementations usually prevent items from being added when the queue is full, or removed when the queue is empty. The way this is handled varies from one implementation to another:

- Some implementations use *blocking calls*: a call to `queue_get()` when the queue is empty, or to `queue_put()` when it's full, will block (wait) until the queue is in a state to allow them to continue. This is only any use in a system that can do more than one thing at a time, for example if an operating system is being used.
- Other implementations use *return codes* to indicate success or failure. This is easy with `queue_put()`; it can be modified to return an `int`, and the returned value can indicate success or failure. With `queue_get()` it's more complicated, because it already returns a value. C has no facility for returning more than one value from a function. The options are:
  1. Change the way that `queue_get()` operates, so that it returns a success or failure value like `queue_put()` does, and that it returns the actual queued value by accepting a pointer-to-`int` as an argument;
  2. Choose a 'special value' that will never be inserted into the queue, and which `queue_get()` can return to indicate that the list is empty. For example, if you know that the numbers in the queue should always be positive, then returning `-1` could indicate failure. If all numbers are allowed in the queue though, it's not possible to choose one that will never be used.




Your task is to implement one of the two 'return code' options – preferably option 1.

You will need to:

- Modify the `queue_put()` and `queue_get()` function declarations as appropriate, to return an `int` if they don't already. If you're implementing option 1, you'll also need to add a pointer-to-`int` parameter to `queue_get()`.
- Work out how to return an appropriate value from `queue_get()` when the queue is empty.

- Notice how, when the queue is first created, the `insert` and `remove` values are both set to zero. This is a clue: when the list is empty, the `insert` and `remove` values will *always* be equal to each other.
- If you’re implementing option 1, you can return ‘true’ (1) and ‘false’ (0) for success and failure respectively.<sup>2</sup> If you’re implementing option 2, you’ll have to choose a special value to return when the list is empty, and otherwise return the retrieved value.
- Work out how to return an appropriate value from `queue_put()` when the queue is full.
  - Whichever option you’re implementing, you can return ‘true’ (1) and ‘false’ (0) for success and failure respectively this time.
  - Determining when the list is full is harder than determining when it’s empty. Think of it this way: if the `insert` and `remove` values are ever allowed to become equal, the list will appear to be empty. So the `queue_put()` function can never allow the `insert` value to equal the `remove` value – and so the queue is considered to be full if the `insert` value is one less than the `remove` value. But be careful: this is modulo-10, so 9 is one less than 0! See if you can devise a way to use the modulo operator as part of the comparison, so that this isn’t a problem.

As ever, please ask for help if you have any questions. In particular, the simplest solution for identifying a full queue will come at the cost of a single queue space. I’m happy to entertain discussions about whether or not this is a worthwhile tradeoff.

-  Once again, the code you have written is intended for the purposes of learning to solve problems of this type, and is not intended to be representative of how to write a circular buffer. In fact in many implementations, a different method of determining whether the queue is full or empty is employed, which helps allow concurrent access to the queue from multiple threads and/or asynchronous interrupt service routines in more complex systems. We’ll look at these things in another lab session.

### 4.4.3 Binary Heaps

Binary heaps are a useful data structure for storing items that must be retrieved in some sort of order. They are useful because:

- Their algorithmic complexity is  $\mathcal{O}(\log n)$  for both insertion and removal (compare this to the  $\mathcal{O}(n)$  insertion and  $\mathcal{O}(1)$  removal complexity of insertion-sorted lists);
- They are easy to implement using an array, and have zero storage overhead.



Use Project→Open Project... to open `Storage\heap\heap.uvprojx`.

<sup>2</sup>In C there is no boolean type, so integers are used instead. By convention, a zero represents ‘false’ and any other value represents ‘true’.

As you learned in the video, a binary heap relies on two operations – *up-heap* and *down-heap* – for its insertion and removal procedures. In `heap.c` you will find most of an implementation of a simple heap to hold integers, but the `heap_up()` and `heap_down()` functions are not implemented.

The `heap_t` structure contains two fields. The `store` pointer points to an array that holds the elements in the heap, and `size` gives the number of elements present in the heap.



As you could probably have guessed, your task is to implement the `heap_up()` and `heap_down()` functions. This is intended to be a *min-heap*, with the smallest element at the root, so the `heap_up()` function must:

1. Start with the last element in the heap
2. If it's the root element, stop
3. Compare it with its parent
4. If the parent is smaller or equal, stop
5. Swap the element with its parent
6. With the element in its new location, go back to step 2

The `heap_down()` function must:

1. Start with the root element
2. If it has no children, stop
3. Compare it with its children
4. If the element is smaller than or equal to both children, stop
5. Swap the element with the smaller of the children
6. With the element in its new location, go back to step 2

Remember how a heap is navigated. Using a 1-indexed array (remember, arrays in C are zero-indexed, so you'll need to account for that):

- The children of node  $n$  are at  $2n$  and  $2n + 1$
- The parent of node  $n$  is at node  $n/2$  if  $n$  is even or  $(n - 1)/2$  if  $n$  is odd, which can be calculated just by computing  $n/2$  using integer arithmetic

#### 4.4.4 A Generic Heap: Function Pointers

This section is included as a way to introduce you to the syntax and usage of *function pointers* (i.e. pointers to functions). A function exists somewhere in memory, so it has an address, and can therefore be the target of a pointer.

Pointers to functions are commonly used to implement *callbacks*: if you were interacting with a slow hardware device (such as a printer) via a driver, and wished to be notified when the state of the hardware device changed, you might pass a pointer-to-function

to the driver. The driver would then call your *callback function* when the device state changed.

#### 4.4.4.1 Declaring function pointers

The syntax for declaring a function pointer is slightly strange. It's easiest to demonstrate it with some examples:

```
void (*fp) (void);
```

Declares a pointer called `fp` that points to a function that takes no arguments and returns nothing.

```
uint32_t (*func) (void);
```

Declares a pointer called `func` that points to a function that takes no arguments and returns a `uint32_t`.

```
void (*foo) (uint16_t);
```

Declares a pointer called `foo` that points to a function that takes a single `uint16_t` argument and returns nothing.

```
double (*callback) (uint32_t, int_fast16_t *);
```

Declares a pointer called `callback` that points to a function that takes two arguments (a `uint32_t` and a pointer to an `int16_t`) and returns a `double`.

#### 4.4.4.2 Function names and function pointers

Recall how the name of an array ‘becomes’ a pointer to the first array element as soon as it's used. This means that a pointer to some data behaves in the same way as an array, and all the array access syntax applies to a pointer just the same as it applies to an array.

The same is true of function names. Whenever you use a function name, it effectively *becomes* a function pointer – and the syntax for calling a function is the same no matter whether you're using a real function name or a function pointer.

For example, the following code is valid:


```
void function(uint32_t value) {
    printf("Function called, value is %d\r\n", value);
}

int main(int argc, char[] argv) {
    void (*fptr) (uint32_t);           // Declare a function pointer
    fptr = function;                   // Assign the function pointer
    fptr(5);                           // Call the function via the pointer
}
```

#### 4.4.4.3 A generic heap


The heap implementation you have created so far works only with one type of data. It could be modified reasonably easily to work with any kind of numeric data, but for anything more complex it won't work because more complex data types cannot be compared using the `<` and `>` operators.

In a more generic heap implementation, the data could either be stored in the heap itself or pointers to the data could be stored. Either way, the `heap_up()` and `heap_down()` functions will need a way to compare elements in the heap. This can be achieved using function pointers: the heap structure can contain a pointer to a comparison function, which is called each time two elements need to be compared.

-  In order to be truly generic, the heap must be capable of holding pointers to *any* kind of data. For this purpose we will use heap elements of type `void *`.<sup>2</sup>


 Use Project→Open Project... to open `Storage\heap_generic\heap_generic.uvprojx`.

You will see that there is a structure type `example_item_t` declared, and then an array of pointers to these structures is declared. The calls to `heap_insert()` pass pointers to compound literals to populate the heap.

-  Look at the definition of the `heap_t` type in `heap.h`. You'll see that it contains a pointer to an array of `void` pointers, which will point to the heap elements. It also contains a function pointer, using the line

```
int_fast8_t (*comparator) (void *, void *);
```

This means that the `comparator` field points to a function that accepts two `void *` elements, and returns an `int_fast8_t`. The static heap initialiser also includes a parameter for the comparator callback function, and uses it to populate this field.

 To get this heap to work, your tasks are:

- Write a static comparator function in `main.c` for the `example_item_t` type. Whenever it is called by the heap, the two parameters will be pointers to `example_item_t` structures, so you may begin by casting them accordingly. The function must return a number *greater than 0* if the first parameter is “greater” than the second, a number *less than 0* if the first parameter is “less” than the second, and *zero* if they are equal. In this case, items are to be compared only using their `priority` fields, and ignoring their names.
- Copy your `heap_up()` and `heap_down()` functions from the previous exercise, but modify them to call the heap's comparator function instead of using the numeric comparison operators.

Test your code. Does it work? If not, can you debug it? Please ask for help if you need it.

---

<sup>2</sup>The introductory videos have covered void pointers, but if you want a recap, you might want to read [Appendix 4A](#) on page 57 at this point.

⚠ This kind of code is often useful, but we’re starting to see the dangers of type casting and void pointers. The compiler is unable to check, for example, that the type of the items added to the heap is consistent with the expected type.

## 4.5 Linked Lists

❗ Linked lists are useful for all sorts of real-world purposes, as you’ll discover. But this section is about linked lists in a more abstract sense.

As you know, linked lists consist of a “chain” of items, with each pointing to the one after it, and optionally also to the one before it (creating a doubly-linked list).

The next lab will contain a lot more information about memory allocation, but for the moment I’ll introduce a rule of thumb: *wherever possible, the pointers should form part of the data in the list*. If the items of data in the list don’t themselves contain the pointers, you end up needing to allocate and deallocate separate storage for the pointers whenever items are added to or removed from the list. This brings several problems with it, and is best avoided.

### 4.5.1 A circular buffer

For this task we will be using a doubly-linked list to implement a *round robin*. A round robin is a way of choosing all elements in a group equally in a rational order, and usually means running through a list repeatedly from beginning to end.

To do this we’ll create a linked list that doesn’t *have* a beginning or an end. It’ll have a *head*, which will move around the circular list as items are enumerated.

⚙ Use Project→Open Project... to open `Storage\linkedlist\roundrobin.uvprojx`.

❗ In `list.h` a couple of structure types are created. The first is `list_item_t` which is an example of a data structure suitable for storage in a doubly-linked list. It contains pointers to the previous and next items, as well as some data; for the purposes of this example, it’s just a string, but of course it could be anything. Note the use of an intermediate name for this structure; the name `list_item_t` is not available until the `typedef` is complete, so the `prev` and `next` pointers are declared using the full `struct list_item_s` name. The other structure declared here is `list_t` which contains a pointer to the head of the list. There are also prototypes for three functions here.

You will see some test code in `main.c`: an array of list items is defined, and items are added to and removed from the list, with loops to print out list entries in between. The array is not required as part of the list storage, it’s just a convenient way to declare and refer to a number of items.

⚙ The functions in `list.c` all need completing.

- `list_insert()` should insert a new item into the list. You will need to:
  - Check to see if the list is empty. If it is, the new item will need to link to itself (in both directions) and become the new list head.
  - Insert the new item by updating the pointers in the item and pointers in the two existing items that will be either side of it.
- `list_remove()` should remove an item from the list. It is acceptable to assume that the supplied item is part of the supplied list! You will need to:
  - Check to see if the item being removed is the only one in the list. If it is, you'll need to set the head pointer to zero.
  - Remove the item by updating the pointers items either side of it to point to each other.
  - Update the head pointer if the item being removed was the head of the list.
- `list_next()` should return an item from the list, and update the head pointer to point to the next one.

Build and test your code to ensure that your logic is correct. The final `printf()` call in `main.c` should print a value of zero, because the list should be empty at that point.

## 4.6 Summary and Conclusions

By now, you should:

- Know how structures work in C;
- Understand how arrays can be used to implement stacks, queues and heaps;
- How the logic of adding and removing items works in linked lists.

If you have any questions about what you have learned, please ask.

**Remember to copy your project files to a network drive before you log off.**



## Appendix 4A Void Pointers

You’ve seen the `void` keyword before – it’s used to indicate that a function takes or returns no arguments. But `void` is actually a data type. It refers to a special type of data item that has *zero size*. No bytes at all. Obviously, a `void` variable can’t take a value; even zero requires some storage. So `void` is never useful as a variable type.

A pointer-to-`void` is therefore a pointer to nothing. You can’t dereference a `void *`, because it doesn’t point to anything. You can’t even increment a `void *`, because the underlying data type has a size of zero. So what use is it?

It turns out that `void *` breaks a rule. Usually in C you can’t assign a value to a variable unless the value and the variable have the same type. This includes pointer types. So:

```
uint32_t *x;
uint32_t *y;
double *z
y = x;      // Compiles fine
z = x;      // Compile error
```

But as a special exception, the C compiler will allow assignment of a `void *` to or from *any other pointer type* without a cast. Effectively this means that a `void *` is not a ‘pointer-to-nothing’, it’s a ‘pointer-to-anything’.

```
uint32_t *x;
uint32_t **y;
void *z
z = x;      // Compiles fine
y = z;      // Compiles fine (but what does it do?!)
x = y;      // Compile error (levels of indirection differ)
```

Don’t use `void` pointers unless you need to. They break ‘type safety’, the mechanism where the compiler protects you from assignments that convert one type to another implicitly, as you can see above. Usually, you’ll know what types your pointers have, so `void` pointers will not be necessary. However, in a systems programming context, they can be very useful, as you’ll see in several lab sessions.

### 4A.1 Pointers-to-Pointers-to-Void

Although the `void` data type takes no space, a pointer-to-void (`void *`) does; it’s a pointer, so it’s the same size as any other pointer.

So if you ever have a pointer-to-pointer-to-`void` (`void **`), it will behave like a normal pointer. You can dereference it to get a pointer-to-`void`, you can increment it, and so on.

In terms of cast-free assignment, think of it as a ‘pointer-to-pointer-to-anything’. You can assign a `void **` to a `uint32_t **`, for example (‘pointer-to-pointer-to-anything’ is a valid description for ‘pointer-to-pointer-to-`uint32_t`’), but not to a `uint32_t *` because

the levels of indirection don't match ('pointer-to-pointer-to-anything' does not describe 'pointer-to-`uint32_t`').

Just to complete the picture, in case this isn't already confusing enough, you can of course assign a `void **` to a `void *` even though the levels of indirection don't match, because that's what a `void *` does.