



---

# INFO1910 S2 2023

# Week 7 Tutorial

---

## Structs, Unions and Function Pointers

### Type Qualifiers

We will briefly define a few useful keywords that modify the behaviour of variables in C.

```
// This variable cannot be modified after assignment  
const int x = 5;
```

```
// This variable's value may be modified by another thread or process  
volatile int y = 5;
```

We will set volatile aside for now, and focus on `const`. Const-ness of a variable does not change where it is stored, or what the value is - it only qualifies that this variable may not be modified.

```
// Constant pointer to an integer  
const int* x = NULL;  
  
// Pointer to a constant integer  
int* const y = NULL;  
  
// Constant pointer to a constant integer  
const int* const z = NULL;
```

### Question 1: Casting Constness

Can you cast into and out of constness?

Can a pointer to a const value point to a non-const value?

Can a pointer to a non-const value point to a const value?

## Storage Class Specifiers

Other keywords that act on variables are storage class specifiers. These don't modify the behaviour of the variable but change where it is stored in memory.

```
// Global variable
int global_x = 5;

// extern variable, linked from another object file
extern int z;

void fn()
{
    // Stack variable, stored on the stack
    int a = 5;

    // Register variable, should be on a register
    register int x = 5;

    // Static variable, bound to function but persists
    static int y = 5;

    // extern variable, stored by a declaration outside the stack frame
    extern int z;
}
```

Global variables should be avoided wherever possible. These variables are stored off the stack and hence persist between function calls. They are also accessible from all functions that can view the label of the variable during compilation and before linking. This may be extended using the `extern` keyword. Global variables then conflict with the names of variables in all functions and make it difficult to ensure that functions are pure and easily debugged.

Static variables are also stored in the GOT, but may only be accessed from within the function that they are defined in. The effect of this is that the value of the variable will persist between calls to the same function. The assignment expression for the static variable is only performed the first time that the function is called.

Register variables are to be stored directly on registers rather than the stack where possible. They are still bound to the stack frame within which they are defined.

## Structs and Unions

We've seen a pattern a few times now where we intentionally allocate a buffer in memory, and can then use sections of the buffer for different purposes, or re-interpret the buffer for different purposes. Most prominently was re-using a `size_t` as an eight character string back in week three.

We will now attempt to generalise this concept and provide a more useful method of declaring and handling these objects. A struct is a collection of types that are treated as a single object in memory. The fields within the structs can be accessed using the `.` operator.

```
struct my_struct
{
    int x;
    char y;
    double z;
};

int main()
{
    struct my_struct example;
    example.x = 5;
    example.y = 'c';
    example.z = 0.8;
    return 0;
}
```

Here we can see the declaration of the struct 'type', the instantiation of a variable of that type and then accessing the fields of the struct.

You may think that when it comes to pointers to structs, accessing elements would be a pain. As a result we also introduce the `->` operator to resolve this problem.

```
struct my_struct* example_ptr;

// These are equivalent
*(example).x = 6;
example->x = 7;
```

Unions act in a similar fashion, however where a struct stores all its members, a union shares its memory with multiple representation defined by its members.

```
union my_union
{
    int x;
    double d;
    float f;
    char c;
}
```

Each of these are different representations of the same block of data stored by a union variable, however as the same position in memory is used for all representations and as the types themselves have different binary representations, it is generally advised to only use one representation at a time.

Alternatively, this can be a very easy method of storing data with multiple possible representations and then deciding which representation is appropriate after the fact.

## Question 2: Chess

The first and simplest use of these structs is to now return multiple variables from a function. Simply wrap the variables as a struct and let the function return the struct.

Write a function that returns a random position on a chess board. Positions on chess boards are defined by a character from A to H and a number from 1 to 8.

You may want to make use of the random function from `stdlib`. Before using your random function for the first time you should seed it with `srandom`. You can produce deterministic output by using a fixed seed, or you can generate pseudo random output using a seed derived from the current time.

```
srandom(time(NULL));  
int x = random();
```

## Question 3: Unionism

What is the purpose of the following code?

```
union int_char  
{  
    int i;  
    char c;  
}  
// ...  
scanf("%c", &(u.c));  
printf("%d\n", u.i);
```

Suggest how having multiple representations of the same data can be very useful when reading binary streams.

## Question 4: Size of a Function

What is the size in bytes of the following functions?

```
int func_a()  
{  
    int x = 5;  
    int y = 7;  
    return x + y;  
}
```

```
int func_b(int a, int b)  
{  
    int x = 5;  
    int y = 7;  
    return a * x + b * y;  
}
```

```
int func_c(int* a)  
{  
    int x = 5;  
    if (*a)  
    {  
        int y = 7;  
        x += y;  
    }  
    return x;  
}
```

```
int func_d(int* a)  
{  
    int x = 5;  
    if (*a)  
    {  
        int y = 7;  
        x += y;  
    }  
    else  
    {  
        double z = 8;  
        x -= z;  
    }  
    return x;  
}
```

```
int func_e(char* str)
{
    int x = 5;
    while (str++)
    {
        int z = 3;
        x += z;
    }

    while (x > 5)
    {
        double q = 3;
        x -= q;
    }

    if (x == 5)
    {
        int a = 4;
        x += a;
    }
    else
    {
        double q = 9.0;
        x -= q;
    }

    return x;
}
```

Consider the compiler pipeline, why is it important to be able to calculate the size of a function?

How is the size of a function associated with the size of a struct or a union?

## Function Pointers Revisited

We have previously discussed how functions exist in a physical memory location and can be accessed by setting the program counter to this location.

Similarly, we can store these addresses in stack memory as pointers and dynamically redirect the program.

```
void hello()
{
    printf("Hello!\n");
}

void goodbye()
{
    printf("Goodbye\n");
}

int main(int argc, int argv)
{
    // An array of two function pointers
    void (*function_ptr_arr)()[2] = {hello, goodbye};

    if (argc < 2)
    {
        return 1;
    }

    int function_to_call = atoi(argv[1]);

    function_ptr_arr[function_to_call]();

    return 0;
}
```

## Question 5: Implementation

In the context of the compilation pipeline, why are only the arguments and return types required to define a function pointer?

## Question 6: Reverse Polish Revisited

Return for a moment to our implementation of the reverse polish calculator from the previous tutorial. Modify your solution to make use of function pointers.

## Question 7: qsort

The C standard libraries have an inbuilt quick sort function that takes an array and a function pointer as an argument. This function should compare two elements in the array and determine whether they are in order.

Write a comparison function for a pair of strings and for a pair of integers, then see if you can use qsort to sort an array of strings and an array of integers.

## Typedef

We can promote our unions, structs and function pointers to ‘real’ types within the syntax of the language using the typedef operator.

```
struct my_struct {
    int x;
};

typedef struct my_struct my_struct_t;

my_struct_t x;
x.x = 5;

union my_union {
    int x;
    double y;
};
typedef union my_union my_union_u;

my_union_u y;
y.x = 5;

typedef void* (*my_fp_f) (void*);

my_fp_f func;
func(NULL);
```

By convention we postpend `_t` or `_f` to indicate what the new type is (in general terms). However this is inconsistently followed.

You might consider the file pointers and descriptors from last week’s tutorial have themselves been typedefed. It’s worth emphasising that typedef does not create new types, it merely gives a new name to an existing type.

Typedefs primarily provide a layer of indirection for portability (for instance, `size_t` will vary between different architectures), and in addition to improve the readability of the code.



## Classism

Let's now add function pointers to our structs. From this we can construct very rudimentary classes with dynamic dispatch methods.

Consider the following:

```
struct animal
{
    void (*exclaim)();
}

void exclaim(struct animal* a)
{
    a->exclaim();
}

void set_exclaim(struct animal* a, void (*new_exclaim)())
{
    a->exclaim = new_exclaim;
}
```

These dispatch methods take a pointer to an instance of the struct as the first argument, which permits updates on members of the struct, alongside any other needed arguments. These are then re-ordered and the appropriate function pointer called.

We can have different instances of different animals calling different exclaim functions using a common interface. This begins to turn C down the path of object orientation, classes and inheritance. At this point we may notice some common patterns for our functions. We have constructors that build the object along getters and setters that access and modify properties of the object. These will be joined by destructors once dynamic memory becomes involved.

## Question 8: Animal Farm

You are given the following structs for a range of animals on animal farm.

```
struct resources
{
    size_t pork;
    size_t veal;
    size_t lamb;
    size_t milk;
    size_t eggs;
};

struct animal
```

```
{
    size_t legs;
    char* sound;
    int efficiency;
    void (*exclaim)(struct animal*);
    void (*work)(struct animal*, struct resources*);
};

void exclaim(struct animal*);
void work(struct animal*, struct resources*);

void work_day(struct animal*, const size_t n_animals, struct resources*);
```

Animals have different types, each animal produces different resources when they work, with the amount resources depending on the animal's efficiency. So a hen might produce an egg each day, while a cow might produce two milk.

Each type of animal should have its own 'constructor function' that returns a struct for that animal appropriately initialised.

Animal Farm can hold up to 128 animals, these animals may be of the same type. You should make an array of constructors and call random elements in the constructor array to fill the farm. When the `work_day` function is called, each animal on the farm should be made to work.

## Inheritance

Consider the following code:

```
struct my_struct {
    int x;
};

struct my_struct st;
st.x = 5;
printf("%d\n", *((int*)&st));
```

What value would you expect to be printed here and why?

We can extend this idea:

```
struct inner {
    int x;
    int y;
};

struct outer {
    struct inner parent;
```

```
    int another_x;
};

struct outer st;
((struct inner*)&st)->x = 5;
((struct inner*)&st)->y = 7;

printf("%d %d\n", st.parent.x, st.parent.y);
```

We can extend this idea to create arbitrary hierarchies of types that contain the elements of each other. This is the basis of object oriented programming.

## The Virtual Method Table

Defining each function pointer for each animal from scratch every time uses quite a lot of memory, and also takes far too much time. We can shorten this by creating a static struct or array of function pointers (which is not on the stack) and then simply point to the table from our struct. This concept is typically called a virtual method table or `vtable`.

```
struct my_struct {
    void* vtable;
    int x;
};

struct my_struct my_struct_constructor(int x)
{
    // Class methods go in here
    static void (*vtable[])() = {get_x, set_x};
    struct my_struct st;
    st.x = 5;
    printf("%d\n", *((int*)&st));
}
```

Now all of our methods are defined once, and all instances of the same struct are constructed such that they point to the same table of function pointers. These function pointers can all then be resolved using dynamic dispatch and the associated dispatch methods.

## Question 9: Farm

Convert your farm to a struct, storing your animals, resources and appropriate functions as function pointers in the farm struct. Write dispatch methods for each function in the farm struct.

Using the nested structs from the section on inheritance, create a class hierarchy for your farm.

## Question 10: Polymorphism

There has been a revolution on Animal Farm. No longer will the animals toil for the benefit of the capitalistic human class.

You are to write a function `uprising_of_the_oppressed_masses`. This function will compel an animal to do no further work. If the animal is a pig, it will also stand on two legs, not four. You may wish to add an uprising function pointer to the animal struct along with the associated dispatch method.

Consider how you could implement this by simply modifying the entries in the `vtable`.

## Question 11: Four Legs Bad

We want a special sort of pig. This new pig, upon revolting does not merely stand upon two legs but has a new work function; instead of refusing to perform work, it consumes one of each resource.

However, this pig's constructor should begin by first calling the regular pig constructor before adding its own new methods.

## Question 12: Extension: An Interesting Pattern of Computation

Consider the following; I have an array of function pointers, and perform computation by determining the index of the array for the computation that I wish to perform. I am not permitted to use any switches or loops.

Implement a Fibonacci number calculator using recursion within this pattern.

## Question 13: Extension: Finite Memory Turing Machine

Generalise your solution to the question above such that all your function pointers and variables exist within a single 'tape' variable. You may control a single 'head' pointer that is your only method of accessing data within the tape.