# 1. Data Types

# Arrays

Arrays are declared using the syntax `type array_name[array_size];`.

When an array of a certain length is declared, the compiler allocates a contiguous block of memory for it. The size of an array cannot be changed after declaration.

In most contexts, the array name decays to a pointer to the first item of an array. As a result, passing an array to a function actually passes a pointer instead of a copy of the array.

In formal parameters, `int arr[]` and `int arr[123]` are equivalent to `int *a`. However, this syntax is only valid in formal parameters and should be avoided.

`x[y]` is defined as `*((x) + (y))`. Since `name` is a pointer, `name[index]` is equivalent to `*(name + index)`. The address of the element at index `i` of the array `array` is `array + (sizeof(array[0]) * i)`.

The size of an array is not stored. The statement `int array[4] = {0, 1, 2, 3};` declares and initialises an array containing 4 values; however, the statement `array[4] = 5;` won't cause an error and will simply access and update the next address, which could be storing another variable or a garbage value. As a result, it is possible to accidentally and unknowingly change the value of a variable, or to access garbage values.

When initialising arrays, if the initialiser list is shorter than the array, then the leftover elements will be initialised to `0`. For example, `int array[4] = {0, 1};` initialises the elements at indices `2` and `3` to `0`. The expressions in the initialiser must be constant, variables cannot be used.

When an array name is assigned to a pointer variable, the variable points to the first element of the array.

## Dynamic Allocation of Arrays

When declaring arrays, the size needs to be known at compile time and variables cannot be used. However, dynamic memory allocation of arrays is possible using the function `malloc`. The syntax is as follows `type *array = malloc(length * sizeof(type));`. Note, `sizof(array)` will return the size of the pointer.

It is useful when returning an array from a function. Since local variables are deallocated when the function terminates, but memory allocated using `malloc` isn't, functions can return pointers to memory allocated using `malloc`.

Read more about `malloc` in 3. Dynamic Memory Allocation.

## Designator

Designators are a C99 feature. In a designated initialiser, each value is labelled with the member it initialises. Since the values are labelled, they don't need to be in order.

As with regular initialisers, the leftover members will be initialised to `0`. The expressions must also be constant.

If the length of the array is not mentioned, the compiler will use the largest designator to determine it.

Designators increase the readability and make it easier to check for errors.

```
int array_1[5] = { [1] = -3, [3] = 4 };
int array_2[5][5] = { [1][2] = -3, [3][0] = 4 };
int array_3 = {[5][1] = -1};
```

## Compound Literal

Another feature introduced in C99, compound literals are used to create unnamed objects with initialised values. They are particularly useful when passing function parameters. Compound literals may contain arbitrary expressions, not just constants.

```
(int [5]) {1, 2, 3, 4, 5};
(int []) {1, 2, 3};
```

---

# Pointers

Pointers are declared using the syntax `type *pointer_name;`

Pointers are variables that store the address of another variable.

The `*` operator dereferences a pointer and the `&` operator returns the address of a variable. The `[]` operator dereferences the address at an offset from the address stored in the pointer.

## Logical and Arithmetic Operations

If `pt` is a pointer to a variable of type `type` and `n` is a variable of an integer type, then `pt + n` is equivalent to `pt + sizeof(type) * n` and `pt - n` is equivalent to `pt - sizeof(type) * n`.

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a sub-object at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space. The constant `0` compares equal to null pointers.

If two pointers are subtracted, both should point to elements of the same array object, or one past the last element of the array object. The result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header.

The operators `>`, `>=`, `<`, and `<=` convert the pointers to unsigned integers for comparison. Pointer comparison is defined for pointers pointing within the same object in memory.

## Null Pointers

A null pointer can be obtained by casting, implicitly or explicitly, the constant `0` to a pointer type. A null pointer is guaranteed to not compare equal to a pointer to any actual value. Hence, it can be used as a signal value. `NULL` can be used instead of `0` for the purpose of clarity. The `NULL` macro is defined in `locale.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`.

Note, a null pointer may not necessarily point to the address `0x0`. However, casting the constant `0` of an integer type to a pointer type is guaranteed to result in a null pointer.

All non-null pointers test true. Thus, `if(ptr == NULL)` is equivalent to `if(!ptr)` where `ptr` is a pointer variable.

## Pointers to Pointers

Since pointers are variables, there can be pointers to pointers.

Pointers to pointers are commonly used when passing pointers as arguments to functions and are required when the function has to change where the pointer points.

```c
#include <stdio.h>
#include <stdlib.h>

void badDoStuff(int *localPointer) {
        localPointer = malloc(sizeof(int));
        *localPointer = 10;
}

void goodDoStuff(int **pointerPointer) {
        *pointerPointer = malloc(sizeof(int));
        **pointerPointer = 10;
}

int main() {
        int *pointer;
        printf("Bad Function - %d\n", (badDoStuff(pointer),
*pointer));
        printf("Good Function - %d\n", (goodDoStuff(&pointer),
*pointer));
        free(pointer);
        return 0;
}
```

Here, the bad function reassigns the local variable `localPointer` to point at the pointer returned by `malloc`, hence not affecting the variable `pointer`. However, since the good function accepts a pointer to the variable `pointer`, it can dereference the local variable `pointerPointer` once and then directly manipulate the variable `pointer`.

Read more about `malloc` in 3. Dynamic Memory Allocation

## Pointers to Functions

Pointers to functions are often passed as parameters to other functions. They are useful as they allow us to write functions to be as general as possible.

When a function name isn't followed by parentheses, the C compiler produces a pointer to the function instead of generating code for a function call.

We can also declare arrays of function pointers

```c
// function headers
int sum(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);

// Array declaration
int (*func[4]) (int, int) = {sum, sub, mul, div};

// Calling functions at an index
for(int i = 0; i < 4; i++) printf("%d", (*func[i]) (4, 2));
```

---

# Structures

Structures and structure variables are declared using the syntax

```c
/* Variables of a type of struct can be declared in two ways, as
 * shown below. The list of variable names before the semicolon is
 * optional and variables can be declared using the usual syntax
 * Optionally, structs can be initialised at declaration (struct_3)
**/

struct tag {
        type member_1;
        type member_2;
} struct_1, struct_2[10], struct_3 = { value_1, value_2 };

struct tag struct_4;
```

Note, struct tags are not recognised without the keyword `struct`, and hence won't clash with variables or typedefs of the same name. Structs need not have a tag if all the variables are declared along with the struct.

When struct variables are initialised, the values in the initialiser must be in the same order as the members of the struct. The expressions in the initialiser must be constant (variables cannot be used). The initialiser does not need to contain values for all members. Leftover members will be initialised to `0`.

Structs are useful for aggregating data that is not of the same type and when the data has no relation to numeric indices.

The word `struct` is required whenever a variable of a structure type is declared.

Memory is allocated in the order that the members are declared in a struct. Padding might be added to align the data in memory. As a result, the order of declaration must be carefully determined.

The `->` (arrow) operator can be used to access the members of a struct through a pointer to it. The syntax is `structPointer->member`. The arrow operator is defined as `(*structPointer).member`. The parenthesis are necessary since the `.` operator has a higher precedence than the `*` operator, parenthesis need to be used when accessing a member through a pointer to a struct.

The `=` (assignment) operator can be used to copy the values of the members of a struct into another struct. `struct tag struct_5 = struct_3;`

Unlike arrays, a copy of the struct is passed to a function and returned by it. As such, it may introduce a large overhead, particularly with large structs. Passing a pointer to a struct should be considered instead.

Structs can also be self-referential.

```
struct node {
        int root;
        struct node *left;
        struct node *right;
};
```

## Designator

```
struct demo {
        int member_1;
        int member_2;
} struct_1 = { .member_1 = 1, .member_2 = 2 };

struct demo struct_2 = { .member_2 = 1, .member_1 = 2 };
```

## Compound Literal

`(struct demo) {1, 2};`

---

# Unions

Like structs, unions contain one or more members. However, the amount of memory allocated is equal to the size of the largest member. All the members are stored at the same address. As a result, altering a member alters all members, which could corrupt them. Hence, accessing members other than the one most recently modified results in undefined behaviour.

Unions and union variables can be declared and initialised using the syntax

```
union {
        type_1 variable_1;
        ...
} u_1, u_2 = { value };
```

However, unlike structs, the initialiser can be used to initialise only the first member. The expression in the initialiser must be constant.

The `->` (arrow) and `.` (dot) operators can be used to access the members of a union similarly to structs.

The `=` (assignment) operator copies unions.

## Designator

Designated initialisers can be used to specify which member of a union to initialise.

```
union {
        type_1 variable_1;
        type_2 variable_2;
        ...
} u_1 = { .variable_1 = value_1 }, u_2 = { .variable_2 = value_2 };
```

---

# Enumeration

Enumerated type is a type which can have limited set of values that are enumerated by the programmer. Each value must have a name. These names are called enumeration constants.

Unlike the members of a structure or union, however, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

```
enum BOOL { FALSE = 0, TRUE = 1 } bool_1, bool_2, ...;
enum BOOL bool_3;
```

When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. The first enumeration constant has the value `0` by default.

### Combining Structures, Unions, and Enumerations

```
typedef struct {
        enum { INT, DOUBLE } type;
        union {
                int i;
                double d;
        } data;
} Number;

Number number = { .type = INT, .data.i = 1 };
number.data.d = 2.0;
number.type = DOUBLE;
```

In this way, structures, unions and enumerations can be combined to keep track of which member of a union was last modified.

---

## `typedef`

`typedef` is used for creating aliases of data types.

```
typedef struct tag {
        ...
} tag;

tag struct_1;
struct tag struct_2;
```

Typedefs make the code easier to modify. Since the data types of related variables can be changed by just updating the typedef as opposed to having to carefully change

the data type everywhere the variables have been used. This is especially useful since the size of data types varies between computers.

---

## `sizeof`

`sizeof` returns the the size of its operand. `sizeof(char)` is always `1 byte` however the size of other data types is machine dependent. `sizeof` can be used to calculate the length of an array as follows `sizeof(array)/sizeof(dataType);` or `sizeof(array)/sizeof(*array);`. However, this fails if the array variable is cast to a pointer type.

The type of a `sizeof` statement is `size_t`. The type `size_t` is unsigned and its size is platform dependent.

In C89, it is safest to cast it to `unsigned long` when printing, as it is the largest of the unsigned types and hence guaranteed to be large enough.

```
printf("Size of int: %lu\n", (unsigned long) sizeof (int));
```

In C99, `size_t` can be printed directly.

```
printf ("Size of int: %zu\n", sizeof (int));
```

Note, `sizeof` is a compile time operator and hence cannot be used if the size isn't known at compile time, such as using it on an array passed as a parameter to a function.

---