Structs and unions

Raul P. Pelaez

September 25, 2025

Contents

1 Introduction

Up until now, our programs have been using only the basic data types provided by C, such as integers, floating point numbers, and characters. However, real-world applications often require more complex data structures to represent entities with multiple attributes. This is where structures and unions come into play.

Both of these constructs allow us to create custom data types that can encapsulate (a.i. aggregate, group) multiple related variables, making our code more organized and easier to manage. Note that this is different from arrays, which are collections of elements of the same type. Structures and unions can contain elements of different types.

2 Structures

Info

Structure
A user-defined data type that can hold multiple variables of different types under a single name. A structure is a collection of variables (called members) of arbitrary types that are grouped together.

Info

What is an object?
In programming, an object is an instance of a class or a structure that encapsulates data and behavior. It is a self-contained entity that combines both state (data) and functionality (methods or functions) to represent a real-world concept or entity. Objects are used in object-oriented programming (OOP) to model complex systems and promote code reuse, modularity, and maintainability.
We cannot do OOP in C, since structures (the closest thing to objects in C) cannot have methods (functions) associated with them. However, we can still use structures to group related data together, which is a key aspect of OOP.

In terms of organization, structures allow to group related variables together under a single name, making it easier to manage and manipulate complex data. For instance, say that we need to represent a rectangle as part of a graphical application (e.g., a window). A rectangle can be defined by its width, height, and position (x, y coordinates). We could design a function that adds a rectangle to the screen like this:

```c
/**
  * Adds a rectangle to the screen at position (x, y) with given width, height, and color.
  *
  * @param x The x-coordinate of the rectangle's top-left corner in pixels.
  * @param y The y-coordinate of the rectangle's top-left corner in pixels.
  * @param width The width of the rectangle in pixels.
  * @param height The height of the rectangle in pixels.
  * @param color RGB color of the rectangle (represented as a long integer).
  */
void add_rectangle(int x, int y, int width, int height, long color);
```

Info

Doxygen comments
A special type of comment that can be used to generate documentation automatically. They usually start with `/**` and end with `*/`. Inside the comment, we can use special tags (like `@param`) to describe the parameters, return values, and other aspects of the function or variable being documented. Tools like Doxygen and Sphinx can parse these comments and generate HTML, PDF, or other formats of documentation automatically.
See an example in this source code, which is then rendered in the documentation site. All that was needed to generate that section of the webpage were these lines.

This function takes five parameters, which can be cumbersome and error-prone. It is hard to remember the order and meaning of each parameter, as they are completely arbitrary. It would be nice if a special type, called `Rectangle`, would exist that somehow encoded these four parameters. Then, we could write the function like this:

```c
/**
  * Adds a rectangle to the screen.
  *
  * @param rect The rectangle to add to the screen.
  * @param color RGB color of the rectangle (represented as a long integer).
  */
void add_rectangle(Rectangle rect, long color);
```

Now, the function is much easier to understand and less error-prone. Even when the order of the two parameters is still arbitrary we cannot confuse them (the program would not compile) since a value of type `long` cannot be casted to type `Rectangle` and vice versa. The documentation is also shorter, reducing the cognitive load of the reader. Granted, we still need to document what a `Rectangle` is, but that is a one-time effort and, presumably, the definition of `Rectangle` will be used in many places.

In C, we can define a structure using the `struct` keyword. Here is how we can define the `Rectangle` structure:

```c
struct Rectangle {
    int x;      ///< x-coordinate of the top-left corner
    int y;      ///< y-coordinate of the top-left corner
    int width;  ///< width of the rectangle
    int height; ///< height of the rectangle
};

int main() {
    struct Rectangle rect1; // Declare a variable of type Rectangle
    rect1.x = 10;
    rect1.y = 20;
    rect1.width = 100;
    rect1.height = 50;

    printf("Rectangle: x=%d, y=%d, width=%d, height=%d\n",
           rect1.x, rect1.y, rect1.width, rect1.height);
    return 0;
}
```

Info

The `struct` keyword

The `struct` keyword is used to define a new user-defined type (a structure). It is followed by the name of the structure (e.g., `Rectangle`) and a block of code enclosed in curly braces that defines the members of the structure. Each member has a type and a name, and can be of any valid C data type, including other structures.

Info

Accessing structure members: The . operator

To access the members of a structure, we use the dot operator ".". The syntax is `structure_variable.member_name`. For example, to access the `x` member of the `rect1` variable, we write `rect1.x`.

To avoid having to write the `struct` keyword every time we declare a variable of type `Rectangle`, we can use the `typedef` keyword to create an alias for the structure type. Here is how we can do that:

```c
typedef struct {
  int x;      ///< x-coordinate of the top-left corner
  int y;      ///< y-coordinate of the top-left corner
  int width;  ///< width of the rectangle
  int height; ///< height of the rectangle
} Rectangle; // Now we can use Rectangle as a type without the struct keyword

int main() {
  Rectangle rect1; // Declare a variable of type Rectangle without the struct keyword
  rect1.x = 10;
  rect1.y = 20;
  rect1.width = 100;
  rect1.height = 50;

  printf("Rectangle: x=%d, y=%d, width=%d, height=%d\n",
         rect1.x, rect1.y, rect1.width, rect1.height);
  return 0;
}
```

Output

```
Rectangle: x=10, y=20, width=100, height=50
```

Info

The `typedef` keyword
The `typedef` keyword is used to create an alias for a data type. The above code works by defining a new structure type (without a name) and then creating an alias for it called `Rectangle`. We can also create aliases for existing types, for instance:

```
typedef unsigned long ulong; // Now we can use ulong as an alias for unsigned long
typedef int* int_ptr;        // Now we can use int_ptr as an alias for int*
```

Advice

Should you always use typedef with struct?
Using `typedef` with `struct` is a matter of personal preference and coding style. It can make the code cleaner and easier to read by avoiding the repetitive use of the `struct` keyword. However, it can also lead to confusion if the same name is used for both the structure and the typedef. In general, it is a good practice to use `typedef` for structures that will be used frequently in the code, but it is not strictly necessary.
Other languages (Java, C++, Python...) do not require this "typedef trick" and you can just write `Rectangle rect1;` directly. C is a bit old-fashioned in this regard.

Info

Inline Doxygen comments
We can document the members of a structure using Doxygen comments too. If the explanation is short, we can use the `///<` syntax at the end of the line. If it is longer, we can use the `/** ... */` syntax before the member declaration.

A structure is not restricted to holding only basic data types. It can also hold other structures, arrays, pointers, and even functions. For instance:

```
struct AValidStruct{
  int id;                      ///< Unique identifier
  char name[50];              ///< Name of the entity
  struct Rectangle rect;       ///< A Rectangle structure as a member
  float* data;                ///< Pointer to an array of float data
  int (*compute_area)(struct AValidStruct*); ///< Function pointer to compute area
};
```

Advanced

Functions inside structures
In C, we cannot define methods (functions) inside structures as we do in object-oriented languages like C++ or Python. However, we can achieve similar functionality by using function pointers as members of the structure. Beware, though, trying to do object-oriented programming in C is usually a bad idea. If you need that, use C++.

Advice

Naming conventions
When naming structures, it is common to use PascalCase (e.g., `Rectangle`, `AValidStruct`) for the structure name and camelCase (e.g., `rect1`, `computeArea`) for variable and function names. This helps to distinguish between types and variables/functions at a glance.
Different languages, frameworks, libraries or teams may have their own conventions, you should strive to be consistent with the conventions used in the codebase you are working on. For instance, in the Linux kernel, structures are usually named using lowercase with underscores (e.g., `my_struct`), while `Python` uses `snake_case` for variable and function names (e.g., `my_variable`, `my_function()`).
The C standard library uses a mix of conventions, for instance, `FILE` is a structure type (in uppercase), while `fopen()` is a function (in lowercase without underscores). Aliases (typedefs) in the standard library are usually in lowercase with a `_t` underscore (e.g., `size_t`, `uint32_t`).

2.1 Pointers to structures

We can also create pointers to structures, which can be useful when passing structures to functions or when working with dynamic memory allocation.

In the following example, we create a pointer to a `Rectangle` structure and pass it to a function that will fill its members:

```c
#include <stdio.h>

typedef struct {
  int x;      ///< x-coordinate of the top-left corner
  int y;      ///< y-coordinate of the top-left corner
  int width;  ///< width of the rectangle
  int height; ///< height of the rectangle
} Rectangle;

void default_rectangle(Rectangle* rect) {
  rect->x = 0;        // Use the -> operator to access members through a pointer
  rect->y = 0;
  rect->width = 100;
    rect->height = 50;
}

int main() {
  Rectangle rect1; // Declare a variable of type Rectangle
  default_rectangle(&rect1); // Pass the address of rect1 to the function
  printf("Rectangle: x=%d, y=%d, width=%d, height=%d\n",
         rect1.x, rect1.y, rect1.width, rect1.height);
  return 0;
}
```

Output

```
Rectangle: x=0, y=0, width=100, height=50
```

Info

The -> operator
When we have a pointer to a structure, we use the arrow operator (`->`) to access its members. The syntax is `pointer_variable->member_name`. For example, to access the `x` member of the structure pointed to by `rect`, we write `rect->x`.

In the following example, we create a heap array of rectangles and print their properties:

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

typedef struct {
  int x;      ///< x-coordinate of the top-left corner
  int y;      ///< y-coordinate of the top-left corner
  int width;  ///< width of the rectangle
  int height; ///< height of the rectangle
} Rectangle;

void print_rectangles(Rectangle* rect, int count) {
  for (int i = 0; i < count; i++) {
    printf("Rectangle %d: x=%d, y=%d, width=%d, height=%d\n",
           i, rect[i].x, rect[i].y, rect[i].width, rect[i].height);
  }
}

int main() {
  int n = 3;
  Rectangle* rects = (Rectangle*)malloc(n * sizeof(Rectangle));
  for (int i = 0; i < n; i++) {
    rects[i].x = i * 10;
    rects[i].y = i * 20;
    rects[i].width = 100;
    rects[i].height = 50;
  }
  print_rectangles(rects, n);
  free(rects);
  return 0;
}
```

Output

```
Rectangle 0: x=0, y=0, width=100, height=50
Rectangle 1: x=10, y=20, width=100, height=50
Rectangle 2: x=20, y=40, width=100, height=50
```

2.2 Structures in the standard library

The C standard library defines several structures that are widely used in C programming. Here are some of the most common ones:

- `FILE`: Represents a file stream and is used for file I/O operations. It is defined in the `stdio.h` header file.
- `time_t`: Represents time in seconds since the epoch (January 1, 1970). It is defined in the `time.h` header file.
- `tm`: Represents a broken-down time (year, month, day, hour, minute, second). It is defined in the `time.h` header file.
- `sockaddr`: Represents a socket address. It is defined in the `sys/socket.h` header file.
- `stat`: Represents file status information. It is defined in the `sys/stat.h` header file.
- `dirent`: Represents a directory entry. It is defined in the `dirent.h` header file.

Example

Here is an example printing the current time using the `time_t` and `tm` structures:

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

  int main() {
      time_t now;               // Declare a variable of type time_t
```

```c
    struct tm* local_time;   // Declare a pointer to a struct tm
    time(&now);
    local_time = localtime(&now); // Convert to local time representation
    printf("Current date and time: %02d-%02d-%04d %02d:%02d:%02d\n",
           local_time->tm_mday,
           local_time->tm_mon + 1, // Months are 0-11 in struct tm
           local_time->tm_year + 1900, // Years since 1900
           local_time->tm_hour,
           local_time->tm_min,
           local_time->tm_sec);
    return 0;
}
```

Output

```
Current date and time: 25-09-2025 12:11:45
```

2.3 The size of structures

You might be surprised by the output of the following program:

```c
#include <stdio.h>
typedef struct{
  int a;     ///< 4 bytes
  float b;  ///< 4 bytes
} MyStruct; // Total: 8 bytes
typedef struct {
  char a;      ///< 1 byte
  int b;       ///< 4 bytes
  char c;      ///< 1 byte
} MyWeirdStruct; // Total: ?? bytes

int main() {
  printf("Size of MyStruct: %zu bytes\n", sizeof(MyStruct));
  printf("Size of MyWeirdStruct: %zu bytes\n", sizeof(MyWeirdStruct));
  return 0;
}
```

Output

```
Size of MyStruct: 8 bytes
Size of MyWeirdStruct: 12 bytes
```

In the first case, we have a structure holding two members of 4 bytes each (an `int` and a `float`), so the total size is 8 bytes, as expected. However, in the second case, we have a structure with three members: a `char` (1 byte), an `int` (4 bytes), and another `char` (1 byte). One might expect the total size to be 6 bytes, but the output shows that it is actually 12 bytes.

This is due to a concept called *padding* and *alignment*. Most computer architectures require that data types be aligned in memory according to their size. For instance, an `int` (4 bytes) should be aligned to a 4-byte boundary. In our `MyWeirdStruct`, the first member (`char a`) takes 1 byte, but the next member (`int b`) needs to be aligned to a 4-byte boundary. Therefore, the compiler adds 3 bytes of padding after `a` to ensure that `b` starts at the correct address. After `b`, we have another `char c`, which takes 1 byte, and then the compiler adds another 3 bytes of padding to make the total size a multiple of 4 bytes (the size of the largest member).

Info

Padding and alignment
Padding is the extra space added by the compiler to ensure that data types are aligned in memory according to their size. Alignment refers to the requirement that certain data types must be stored at memory addresses that are multiples of their size. For example, a 4-byte integer should be stored at an address that is a multiple of 4.

We can use this information to play dangerous games with memory, check this out:

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

typedef struct {
  char a;      ///< 1 byte
  int b;       ///< 4 bytes
  char c;      ///< 1 byte
} MyWeirdStruct; // Total: 12 bytes

int main() {
  MyWeirdStruct s;
  s.a = 'A';
  s.b = 42;
  s.c = 'C';
  printf("a: %c, b: %d, c: %c\n", s.a, s.b, s.c);
  // The address of b is 4 bytes after the address of a
  printf("Address of a: %p\n", (void*)&s.a);
  printf("Address of b: %p\n", (void*)&s.b);
  printf("Address of a + 4 bytes: %p\n", (void*)&s.a + 4);
  printf("Contents of &s+4 (%p) as int: %d\n", (void*)&s.a + 4, *(int*)((char*)&s.a + 4));
  printf("Contents of &s+4+4 as char: %c\n", *(char*)((char*)&s.a + 4 + 4));
  return 0;
}
```

Output

```
a: A, b: 42, c: C
Address of a: 0x16d21df54
Address of b: 0x16d21df58
Address of a + 4 bytes: 0x16d21df58
Contents of &s+4 (0x16d21df58) as int: 42
Contents of &s+4+4 as char: C
```

Read this code carefully to understand whats going on. I am proving to you that the memory layout of this structure is like this:

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:
a	pad	pad	pad	b_0	b_1	b_2	b_3	c	pad	pad	pad
1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b

Here, 1b means that each box is 1 byte large. The `pad` boxes are the padding bytes added by the compiler to ensure proper alignment of the `int b` member. The `b_0`, `b_1`, `b_2`, and `b_3` boxes represent the 4 bytes of the integer `b`. The padding bytes are just wasted.

Let me show you another example:

```c
#include <stdio.h>
typedef struct{
  double a;    ///< 8 bytes
  float b;  ///< 4 bytes
```

```c
  float c;  ///< 4 bytes
} MyStruct;

typedef struct {
  float c;      ///< 4 bytes
  double a;    ///< 8 bytes
  float b;      ///< 4 bytes
} MyStructReversed;

int main() {
  printf("Size of MyStruct: %zu bytes\n", sizeof(MyStruct));
  printf("Size of MyStructReversed: %zu bytes\n", sizeof(MyStructReversed));
  return 0;
}
```

Output

```
Size of MyStruct: 16 bytes
Size of MyStructReversed: 24 bytes
```

Knowing about padding and alignment. Can you explain why these two structures, which hold the exact same members and can be used interchangeably, have different sizes?

The size difference between these two identical structures might seem menial, but imagine that you are storing 1 billion of them (which is tiny in a realm such as LLMs). The 16 bytes of the first structure would add up to 16 GB, while the 24 bytes of the second structure would add up to 24 GB. That is a difference of 8 GB, which is significant, more so considering that the 8GB are being wasted in padding bytes. A simple reordering of two lines in the definition of the structure can save you 8 GB of RAM.

3 Unions

Info

Union

A `union` is a data structure that can hold, during its lifetime, objects of different types (from a predetermined list), taking up memory equal to the largest object it can hold. Managing the active type in a `union` can be error-prone.

Example

```c
#include <stdio.h>

typedef union {
  int i;        ///< Integer member
  float f;      ///< Float member
  char str[20]; ///< String member
} Data;

int main() {
  Data data; // Declare a variable of type Data (union)
  printf("Size of union: %zu bytes\n", sizeof(data));

  data.i = 10; // Store an integer
  printf("data.i: %d\n", data.i);

  data.f = 220.5; // Store a float (overwrites the integer)
  printf("data.f: %.2f\n", data.f);
  // Accessing the integer now gives the
  // representation of the bits of 220.5 as an int
  printf("data.i (after storing float): %d\n", data.i);
  // Store a string (overwrites the float)
  snprintf(data.str, sizeof(data.str), "Hello, World!");
  printf("data.str: %s\n", data.str);

  return 0;
}
```

Output

```
Size of union: 20 bytes
data.i: 10
data.f: 220.50
data.i (after storing float): 1130135552
data.str: Hello, World!
```

The size of an `union` is determined by the size of its largest member. In the above example, the `str` member is the largest, so the size of the `Data` union is 20 bytes. When we store a value in one member of the union, it overwrites the previous value stored in any other member. This means that at any given time, only one member of the union can hold a valid value. In environments where register memory is scarce (like embedded devices), unions can be used to save memory by reusing the same memory location for different purposes at different times.

Insight

I seldom use unions in C. They are more common in low-level programming, such as embedded systems or network protocols, where memory efficiency is crucial. In most high-level applications, structures are preferred due to their clarity and ease of use. One place where I have used unions in the past is when implementing high performance, low-level, CUDA kernels. GPUs have very limited register memory, a union can be used to alleviate register pressure by reusing the same memory location for different purposes at different times. This can help to fit more data in the limited register space, potentially improving performance.

4 Data-oriented programming

Info

Data-oriented programming (DOP)
A programming paradigm that focuses on the organization and structure of data, rather than on the behavior of objects or functions. It emphasizes the importance of data layout and access patterns to optimize performance, especially in high-performance computing and game development. Data-oriented programming often involves using structures and arrays to group related data together, making it easier to process and manipulate large datasets efficiently. It is commonly used in languages like C and C++, where low-level memory management and performance optimization are crucial.
DOP optimizes for efficiency of data movement. It is, essentially, to consider the CPU cache when programming.

Up until this point in your journey, you have been mostly exposed to what is known as *procedural programming*, where the focus is on writing functions (procedures) that operate on data. While it is possible to build arbitrarily complex programs just with that approach, it is easy to end up with a tangled mess of functions and variables that are hard to understand and maintain.

Info

Procedural programming
A programming paradigm that focuses on writing procedures or functions that operate on data. It emphasizes a linear top-down approach to program design, where the program is divided into smaller, manageable functions that perform specific tasks. Procedural programming is characterized by the use of control structures (like loops and conditionals) and the manipulation of global and local variables. It is commonly used in languages like C, Pascal, and Fortran.
Procedural programming optimizes for clarity of logic/steps.

Data-oriented programming is a way to organize code around the data it operates on, rather than around the functions that manipulate that data. This approach encourages a clear separation between data and behavior. Let me give you an example:

Example

Imagine you are building a game engine. You are now designing a system that deals with circles (rendering them, transforming them, checking collisions...). In a procedural approach, we might have functions like these:

```
void render_circle(float x, float y, float radius);
void transform_circle(float *x, float *y, float dx, float dy);
int check_collision(float x1, float y1, float r1, float x2, float y2, float r2);
```

We could also have a structure to hold the data of a circle:

```
typedef struct {
    float x;
    float y;
    float radius;
```

```
} Circle;
```

If we want to render all the circles, we would have to do something like this:

```
for (int i = 0; i < num_circles; i++) {
    render_circle(circles[i].x, circles[i].y, circles[i].radius);
}
```

This is pretty straightforward, but it has some drawbacks. For instance, we have to pass parts of the circle data to each function, which can be error-prone and inefficient (read about cache below). On the other hand, rendering one circle might have a large overhead (like setting up some graphics or window context). In a data-oriented approach, we would organize the code around the data and how we operate on it. We could design functions such as:

```
void render_circles(float *xs, float *ys, float *radii, int count);
void transform_circles(float *xs, float *ys, int count, float *dx, float *dy);
void check_collisions(float *xs, float *ys, float *radii, int count, int *collisions);
```

Now, instead of passing individual circle data to each function, we pass arrays of data. This allows us to process multiple circles in a single function call, which can be more efficient and easier to manage. For instance, we could define a structure to hold the arrays of circle data:

```
typedef struct {
  float *xs;      // Array of x-coordinates
  float *ys;      // Array of y-coordinates
  float *radii;   // Array of radii
  int count;      // Number of circles
  } CircleList;
```

Then, we could render all the circles like this:

```
render_circles(circle_list.xs, circle_list.ys, circle_list.radii, circle_list.count);
```

Insight

What I am describing here is related to a graphics technique called *instancing*, where we render multiple instances of the same object (in this case, circles) with a single draw call. In a videogame engine, this is used for things like rendering many trees, blades of grass, or identical enemies (like the rats in "A Plague Tale").

Insight

DOP in other languages

One of the most famous data-oriented libraries out there is `numpy`, which is used in Python for numerical computing. `numpy` provides a powerful array data structure that allows for efficient manipulation of large datasets. It encourages a data-oriented approach by providing functions that operate on entire arrays at once, rather than on individual elements (this is called *vectorization*). This leads to more concise and efficient code, as well as better performance due to improved memory access patterns.

Example

```
import numpy as np

a = np.ones(10) # Create an array of 10 ones
b = np.ones(10) * 0.5 # Create an array of 10 0.5s
c = a + b   # Add the two arrays element-wise
print(c)
```

Output

```
[1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5]
```

Advanced

Data-oriented design vs Object-oriented design
Data-oriented design (DOD) and object-oriented design (OOD) are two different programming paradigms that focus on different aspects of software development. DOD emphasizes the organization and structure of data, while OOD focuses on the behavior and interactions of objects. OOD is centered around the concept of objects, which encapsulate both data and behavior (methods) related to that data. In contrast, DOD separates data from behavior, organizing code around the data it operates on. DOD is often used in performance-critical applications, such as game development and high-performance computing, where efficient data access and manipulation are crucial. OOD is commonly used in applications that require complex interactions between objects, such as user interfaces and business applications. While both paradigms have their strengths and weaknesses, they can also be combined in a hybrid approach, leveraging the benefits of both paradigms to create more maintainable and efficient software. Both are part of a multidimensional spectrum of programming paradigms, not mutually exclusive categories.

Advice

There is no paradigm to rule them all
Different programming paradigms have their own strengths and weaknesses, and the choice of which one to use depends on the specific requirements of the project, the team's expertise, and personal preferences. It is important to be familiar with multiple paradigms and to choose the one that best fits the problem at hand. In practice, many projects use a combination of paradigms, leveraging the strengths of each to create more maintainable and efficient software.

Advanced

AoS vs SoA

When organizing data in our programs, we can choose between two main layouts: Array of Structures (AoS) and Structure of Arrays (SoA). In AoS, we create an array where each element is a structure that contains all the attributes of an entity. This layout is intuitive and easy to work with, as all the data related to an entity is grouped together. However, it can lead to inefficient memory access patterns, especially when processing large datasets, as the CPU may need to load unnecessary data into the cache. For instance, lets say you are simulating particles in a physical regime, we can describe the state of these particles in two ways:

AoS:

```
typedef struct {
    float position[3];
    float velocity[3];
    float mass;
} Particle;
```

SoA:

```
typedef struct {
    float* positions; // Array of positions
    float* velocities; // Array of velocities
    float* masses; // Array of masses
    int count; // Number of particles
} ParticleSystem;
```

The AoS layout (a structure per element), is conceptually easier to understand and work with, but it can easily lead to inefficient memory accesses. Say, for example, that part of our algorithm requires to process just the mass of each particle, we would have to do:

```
for (int i = 0; i < num_particles; i++) {
    process_mass(particles[i].mass);
}
```

Each time we request a particle (`particles[i]`), the CPU will load the entire structure (position, velocity, and mass) into the cache, even if we only need the mass. This leads to cache misses and hurts performance. On the other hand, with the SoA layout (a structure holding arrays of each attribute), we can process just the masses efficiently:

```
float *mass_array = particle_system.masses;
for (int i = 0; i < particle_system.count; i++) {
    process_mass(mass_array[i]);
}
```

In this case, the CPU fills the cache with just the mass data, leading to better cache utilization and improved performance.

Advanced

The CPU cache
Modern CPUs pull all kinds of black magic tricks to try and speed up the execution of programs. They abuse the fact that a program only needs to behave *as-is* from the point of view of the programmer. For instance, CPUs use *branch prediction*. When encountering a branch, a CPU will make a *guess* about which way the branch will go and start executing that path ahead of time. As the program continues, the CPU *learns* the patterns of branches and improves its predictions.
A less crazy trick is the CPU cache. CPUs have a small amount of very fast memory (the cache) that is used to store frequently accessed data. When the CPU needs to access data, it first checks if it is in the cache. If it is, it can access it very quickly. If it is not, it has to fetch it from the main memory, which is much slower. Roughly speaking, the CPU expects that when we access a memory location, we will soon access nearby memory locations (spatial locality) and that if we access a memory location, we will likely access it again soon (temporal locality). As such, when requesting a memory location, the CPU will fetch not just that location, but also a block of nearby memory locations (a cache line). This is why the layout of data in memory is important for performance.
When designing data structures and algorithms, it is important to consider how they will interact with the CPU cache. Data-oriented programming encourages us to organize our data in a way that maximizes cache efficiency.

5 Exercises

5.1 Your first `struct`

Goal

Create a `Rectangle` type (width, height) and write functions:
- `int area(Rectangle r)`
- `void grow(Rectangle *r, int dw, int dh)` (mutates the rectangle)

Separate declaration and definition in `.h` and `.c` files. Have a `main.c` that demonstrates usage. Orchestrate the build with CMake and add unit tests with `utest.h`.

Learning goal

I want you to practice the definition of a struct, passing it by value and by pointer, and using both `.` and `->` to access members. I also want you to practice again with CMake and unit testing.

hint

Remember that passing by value copies the struct; you need to pass a pointer to mutate it.

Milestone

Create a new folder for this exercise and go in there. Activate your usual conda environment, create a new one if you need (you may copy the `environment.yml` file from the template project into here and run `conda env update`).
Open the folder in VSCode by running `code .` in the terminal.
Start with a single `main.c` file, then refactor into multiple files. Define the `Rectangle` type and create an instance of it in `main`, fill it with some values.
Make sure the program compiles and runs as expected. You may simply compile it in the terminal with `cc main.c -o main`.

Milestone

Add a `CMakeLists.txt` to orchestrate the build. Make sure you can build the program with `cmake -B build && cmake --build build`.

hint

Besides the mandatory CMake lines, you just need to add an `add_executable` line.

Milestone

Refactor the code to separate declaration and definition:

- Create a `rectangle.h` for the `Rectangle` type and function declarations.
- Create a `rectangle.c` for the function definitions.
- `main.c` should include `rectangle.h` and use the functions.

Add the new library to the CMake build and link it to the executable. Make sure everything still builds and runs as expected.

hint

You need to add `add_library` and `target_link_libraries` lines to the CMake file.

Milestone

Add the declarations to of `area` and `grow` to `rectangle.h`, and add mock definitions to `rectangle.c` that just return zero/do nothing. Make sure everything still builds and runs as expected.

Milestone

Add unit tests using `utest.h`:

- Create a `test_rectangle.c` file for the tests.
- Add tests for `area` and `grow` functions.
- Add the test executable to the CMake build.

The tests might be very basic, like checking that `area` returns the expected value for a known rectangle, and that `grow` correctly modifies the rectangle properties for an example. Make sure the tests compile and run with `ctest -V --test-dir build`. Naturally, these tests will fail, since you have not implemented the functions yet.

hint

- You need to add another `add_executable` line for the tests and link it to the rectangle library.
- You need to place `utest.h` somewhere CMake can find it (e.g., in the project folder).
- You need to add `enable_testing()` at the top of the CMake file.
- You need to add an `add_test` with the name of the test executable.
- Explore the `CMakeLists.txt` of the template project for guidance.

Learning goal

I want you to do it in this order (mocks first, then tests) to show you that you can write tests before the actual implementation. This is a key idea in Test-Driven Development (TDD).

Milestone

Implement the `area` and `grow` functions in `rectangle.c`. Make sure the tests pass.

Advanced milestone

Add more tests to cover edge cases, like area receiving zero or negative dimensions, and grow with negative deltas. Fix your implementation if needed.

Learning goal

In writing these tests, you are being forced by design to implement error-handling into your functions.

hint

You cannot check for assertions failing with `utest.h`, since assertions abort the program. GTest, the big library `utest.h` is based on, can do this with `EXPECT_DEATH`. If you feel adventurous, you can switch to GTest here, it has a CMake integration module via `FetchContent`, see here.

Advanced milestone

Your functions might have a lot of error handling boilerplate. Reduce it as much as you can by "designing errors out of existence".

5.2 Are you aligned?

Goal

Answer to the question at the end of the section "The size of structures". Why do these two structures have different sizes?