Structures

Learning Outcomes

After reading this section, you will be able to:

- Design data collections using structures to manage information efficiently

Introduction

The most commonly used data structure in C language programs aside from the array is the struct or structure. A structure type is a collection of not necessarily identical types. We use the structure type to define a group of variables as a single object.

This chapter reviews the primitive types and presents the syntax for declaring a structure type, defining an object of structure type, and accessing the data values within that object. This chapter includes an example of how to walkthrough a program that includes structure types.

Types

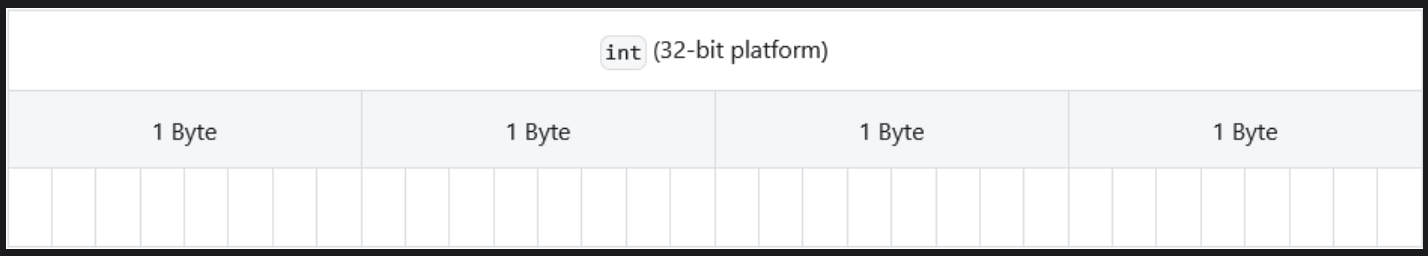
A type describes how to interpret the information stored in a region of memory. In the C language, a type may be a primitive type or a derived type. A derived type is a collection of other types.

Primitive Types

The core language defines the primitive types. We cannot redefine these types or introduce new primitive types. The C language primitive types include:

- `char`
- `int`
- `float`
- `double`

Each type defines how a value of that type is stored in a region of memory. Consider the `int` type. A value of `int` type is stored in equivalent binary representation in 4 bytes on a 32-bit platform:



To define an object of `int` type called `noSubjects`, we write:

```
int noSubjects;
```

Derived Types

The declaration of a derived type in the C language takes the form

```
struct Tag
{
    //... declarations here
};
```

where the keyword `struct` identifies a derived type or structure. `Tag` is the name by which we call the structure (just like `int` above). The declaration concludes with a semicolon.

We list the types that belong to the structure along with their identifiers within the curly braces.

```
struct Tag
{
    [type] [identifier];
    // ... other types
};
```

`type` is the member's type. `identifier` is the name by which we access the member's value.

Example

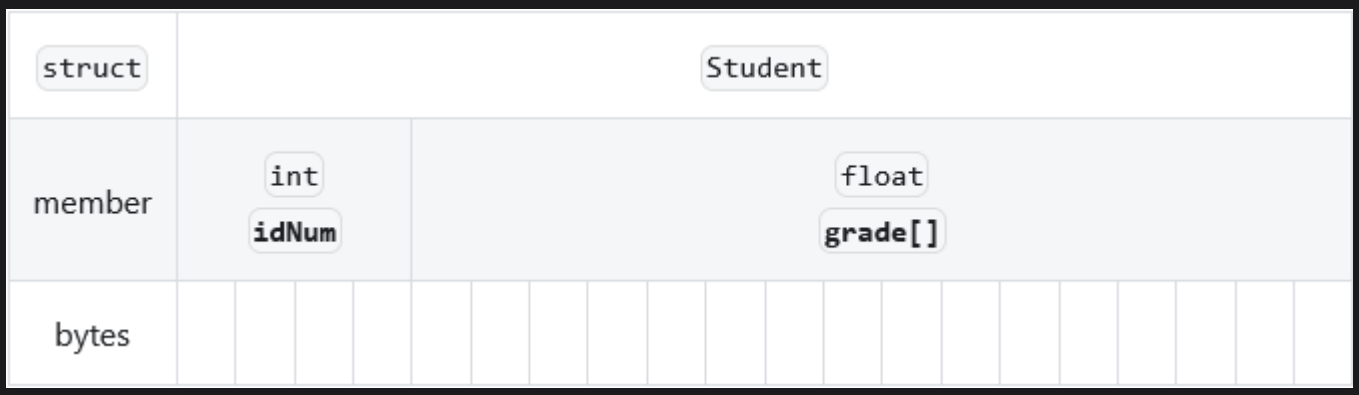
Consider a structure type that consists of two pieces of information:

- the student's ID number
- the student's grades (up to 4 individual grades)

Let us call this structure type `Student`. To declare the type, we write:

```
struct Student
{
    int idNum;       // student number
    float grade[4];  // grades
};
```

The members occupy memory in the order in which we have listed them in the declaration of our structure:



> **NOTE**
>
> This declaration does **NOT allocate any memory** for any object; it only **defines the structure** and the rules for objects of that type (in other words, this is NOT a variable declaration).

Declaration

We declare our structure globally and may store its declaration in a separate file called a header file (say, with the name `Student.h`):

```
// Student.h

struct Student
```

```
{
    int idNum;       // student number
    float grade[4];  // grades
};
```

When we place source code in a header (`.h`) file, we insert that header file's code into the source file that requires that information, as shown below. In such cases, our complete source code is stored in more than one file. When compiling multi-file source code, we only pass the `.c` file(s) to the compiler. The code in a header file is duplicated inside each C file in which it is included, which allows us to write code, like a `struct`, in one spot and edit it in that one spot alone.

> **DID YOU KNOW?**
>
> Header files play an important role in modularity and in the organization of code. Header files often contain additional things like macros and function prototypes which will be discussed in the functions topic.

Allocating Memory

When we define an object of a structure, we allocate memory for that object. Our definition takes the form:

```
struct Tag identifier;
```

where `Tag` is the name of the structure and `identifier` is the name of the object.

Example

To allocate memory for a `Student` named `harry`, we write:

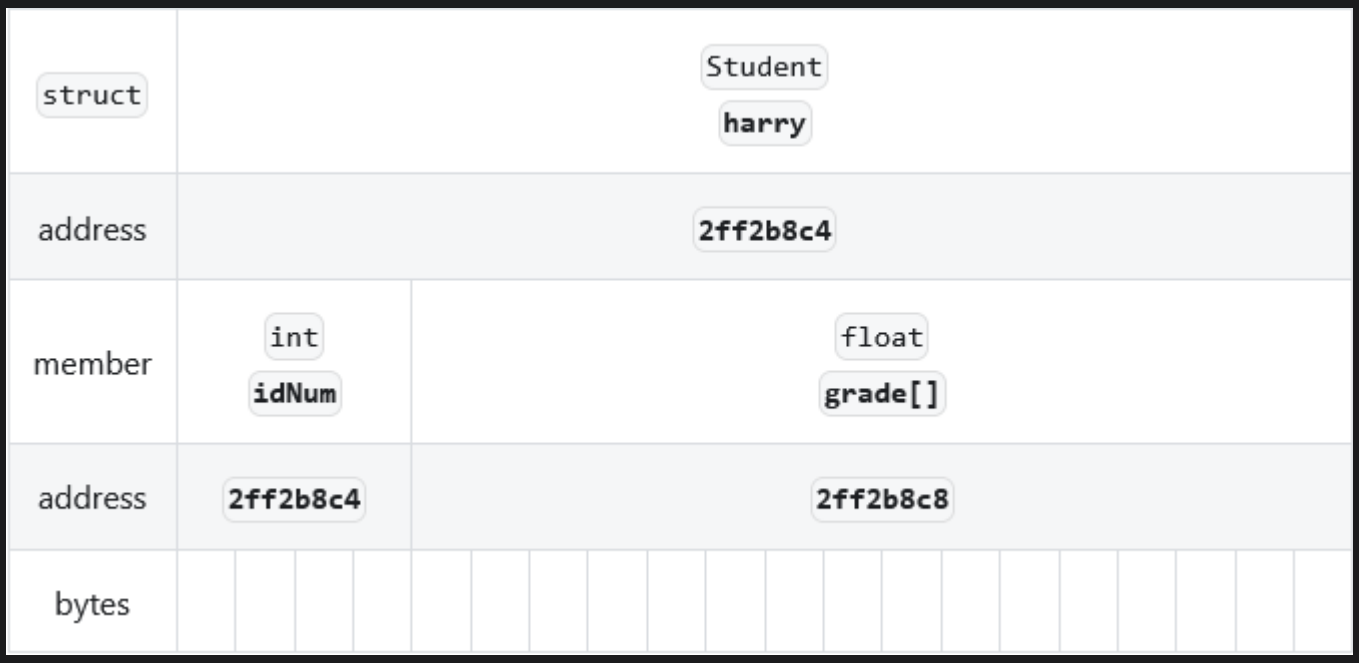
```
// main.c

#include "Student.h" // includes the description of a Student

int main(void)
{
    struct Student harry; // allocates memory for harry
```

```
    // ...

    return 0;
}
```



The object name `harry` refers to the collection of members in `Student harry` taken together.

Initialization

To initialize an object of a structure we add a braces-enclosed, comma-separated list of values. We organize the initial values in the same order as the member listing in the declaration of the structure. The initialization takes the form:

```
struct Tag identifier = { value, ... , value };
```

> **NOTE**
>
> Structure initialization is similar to one of an array.

Example

To initialize `harry` with student number `975` and grades of `75.6`, `82.3` and `68.9`, we write:

```
struct Student harry = { 975, { 75.6f, 82.3f, 68.9f } };
```

`struct`	`Student` **`harry`**				
address	**`2ff2b8c4`**				
member	`int` **`dNum`**	`float` **`grade[]`**			
address	**`2ff2b8c4`**	**`2ff2b8c8`**			
value	`975`	`75.6f`	`82.3f`	`68.9f`	`0.0f`

Member Access

To access a member of an object of a structure, we use the dot operator (`.`). Dot notation takes the form:

```
object.member
```

To access `harry`'s student number, we write:

```
harry.idNum
```

To retrieve the address of a non-array member of an object, we use the address of operator (`&`):

```
&instance.member
```

To access the address of `harry`'s student number, we write:

```
&harry.idNum
```

> **NOTE**
>
> We may omit the parentheses here - `&(harry.idNum)` - they are unnecessary because the dot (`.`) operator binds tighter than the address-of operator (see the precedence table).

To access an array member, we refer to its name without brackets. For example, to access the address of `harry`'s grades, we write:

```
harry.grade
```

To access an element of an array member, we use subscript notation

```
object.member[index]
```

To access `harry`'s ***third*** grade, we write:

```
harry.grade[2]
```

To retrieve the address of an element of an array member, we use the address of operator (`&`):

```
&object.member[index]
```

To access the address of `harry`'s ***third*** grade, we write:

```
&harry.grade[2]
```

`struct`	`Student` `harry`				
address	`&harry`				
member	`int` `idNum`	`float` `grade[]`			
address	`&harry.idNum`	`harry.grade`			
value	`975`	`75.6f`	`82.3f`	`68.9f`	`0.0f`

Example

A convenient alternative to parallel arrays for storing tabular information is an array of structures. One member holds the key, while the other member holds the data.

In the following example, the `sku` member holds the stock keeping unit (sku) for a product, while `price` holds its unit price. The header file with the declaration of the `Product` structure contains:

```
// Structure Example
// product.h

struct Product
{
    int sku;
    double price;
};
```

The program that uses the `Product` structure is listed below.

```
// Structure Example
// structure.c

#include <stdio.h>
#include "product.h"
```

```c
int main(void)
{
    int i;
    struct Product product[] = { {2156, 2.34}, {4633, 7.89},
                                 {3122, 6.56}, {5611, 9.32} };
    const int n = 4;

    printf("  SKU Price\n");
    for (i = 0; i < n; i++)
    {
        printf("%5d $%.2lf\n", product[i].sku, product[i].price);
    }

    return 0;
}
```

The output produced from the above sample is shown below:

```
  SKU Price
 2156 $2.34
 4633 $7.89
 3122 $6.56
 5611 $9.32
```

Walkthrough

A walkthrough table for a program with structure types includes lists of the member types below the object identifiers. The table for the example above is shown below.

The table includes:

- the structure type of each object
- the identifier of each object
- the type of each member
- the identifier of each member

> **NOTE**

> Each object is broken down into its members in the head of the table. We reserve a separate line for the addresses of the different objects:

		struct Product		struct Product		struct Product		struct Product	
		product[0]		product[1]		product[2]		product[3]	
		1000		100C		1018		1024	
int	int	int	double	int	double	int	double	int	double
i	n	sku	price	sku	price	sku	price	sku	price
	4								
	4								
	4								
	4								
	4								

Output: