Structures

Marcus Birkenkrahe

November 26, 2024

Motivation

- There are three user-defined data structures that help organize data and group them logically:

Data Structure	Explanation
`struct`	Different types. Each member has its own memory location.
`enum`	Assigns names to integer constants. All members are integers.
`union`	Groups variables, members share the same memory location.

Data Structure	Use Cases
`struct`	Complex objects like points, employees, or students.
`enum`	Fixed sets of values (days of the week, states, error codes).
`union`	Optimizing memory usage.

- Structures are by far the most important so we'll spend most of our time on them.

First examples

- Structure example: A point is defined as a pair of (x,y) values. Once the `struct` is defined, `Point` can be used like any other data type.

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

```
int main() {
  struct Point p1 = {10, 20};
  printf("Point: (%d, %d)\n", p1.x, p1.y);
  return 0;
}
```

```
Point: (10, 20)
```

- Enumeration example: The `enum Day` is an integer type whose values are named.

```
#include <stdio.h>
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };

int main() {
  enum Day today = WED;
  printf("Today is day number: %d\n", today); // Output: 2 (WED)
  return 0;
}
```

```
Today is day number: 2
```

- Union example: Similar to a structure except that its members share the same storage so that only one member can be stored at a time.

```
union Data {
  int i;
  float f;
  char str[20];
};

int main() {
  union Data data;
  data.i = 42;
  printf("Integer: %d\n", data.i);

  data.f = 3.14;
  printf("Float: %f\n", data.f);
```

```
  strcpy(data.str,"hello");
  printf("String: %s\n", data.str);

  return 0;
}
```

```
Integer: 42
Float: 3.140000
String: hello
```

Structure variables

- So far, we've only covered one data structure, the array. Array elements all must have the same type and they can be subscripted.
- Structure members can have different types, and they have names, which we use rather than their position.
- Most high level languages provide this feature:
  1. In C++, the `class` is an extension of the `struct` with the difference that its members are `private` by default.
  2. In R, the `list` is a `struct` without methods (with with `apply`), and custom methods can be defined.
  3. In SQL, the `table` schema is a `struct` (without methods). Foreign keys link tables like pointers in C.

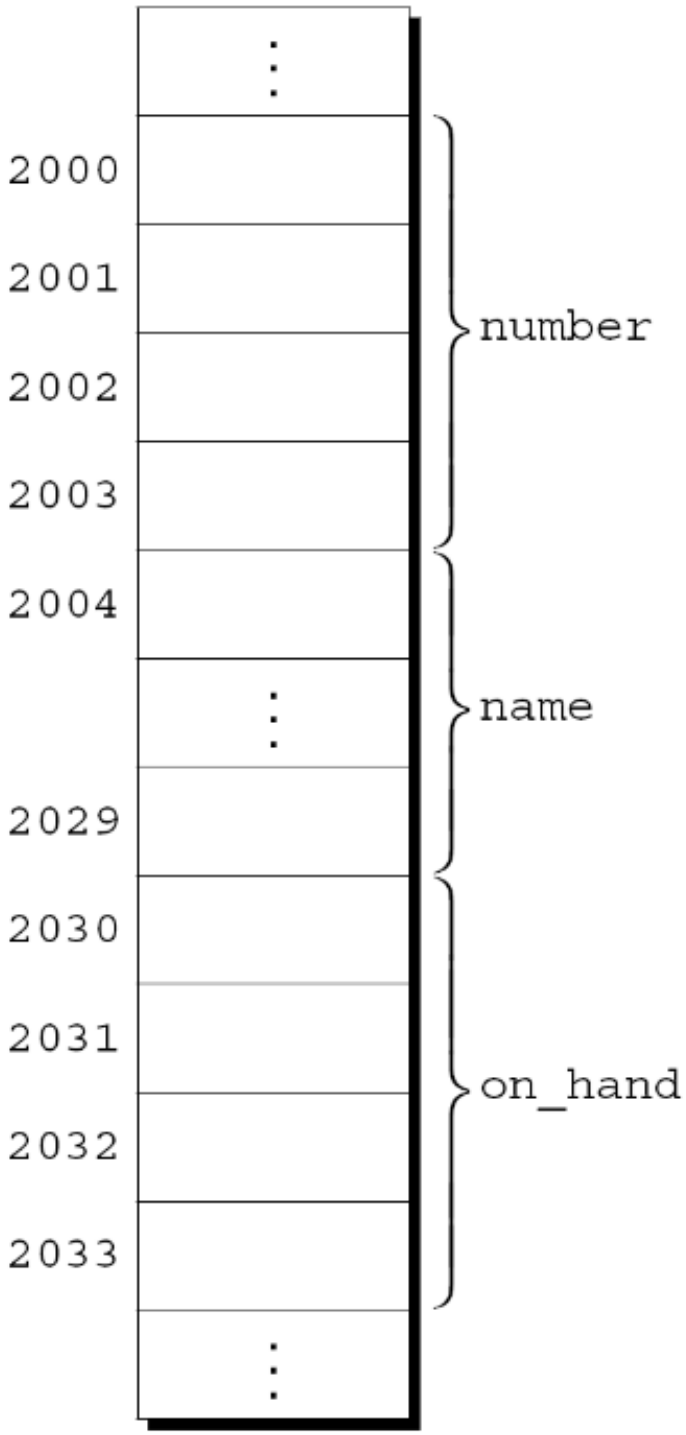
Declaring structure variables

- Structures are for storing a collection of related data items - for example parts in a warehouse, represented by:
  1. Part `number` (integer)
  2. Part `name` (string)
  3. Number of parts `on_hand` (integer)
- In code:

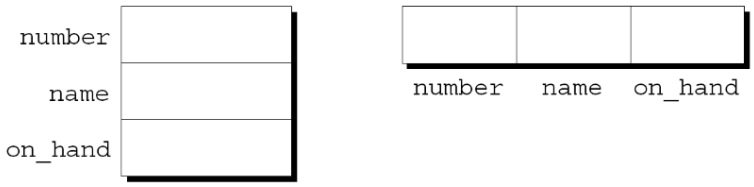
```
#define NAME_LEN 25

struct {
  int number; // parts number
  char name[NAME_LEN+1]; // parts name - string + null character
  int on_hand; // part is on hand
} part1, part2; // two part variables
```

- Members are stored in memory in the order in which they are declared:


2000
2001
2002
2003
2004
2029
2030
2031
2032
2033
number
name
on_hand

- Here, `number` and `on_hand` occupy 4 bytes (`int`), and `name` occupies 25 bytes (`char` has 1 byte x 25).
- Usually, structures are represented by adjacent vertical or horizontal boxes:



- Structure scope: Each structure represents a new block scope, and its names will not conflict with other names in a program. This is also called a `namespace`.
- You remember this concept from beginner's C++: By declaring the namespace `std` at the start, you don't have to write `std::cout` and `std::endl`.

```
using namespace std;

cout << "hello name space" << endl;
```

Using a structure

- Now let's use this parts structure:

```
#define NAME_LEN 25

// declare structure
struct {
  int number; // parts number
  char name[NAME_LEN+1]; // parts name - string + null character
  int on_hand; // how many parts are available
} part1, part2; // two parts

// main program
```

```
int main(void)
{
  // use parts structure
  strcpy(part1.name, "Printer cable"); // cannot copy string array
  part1.number = 528;
  part1.on_hand = 10;

  // print part1 data
  printf("Part: %s, number = %d, on hand = %d\n",
     part1.name, part1.number, part1.on_hand);

  return 0;
}
```

```
Part: Printer cable, number = 528, on hand = 10
```

Practice: Create and test an employee database structure

- Replicate this code for another structure that contains the following information on employees: `number`, `name`, and `sex`. Define two employees, `employee1` and `employee2`.

  When the code compiles, test the structure in a `main` program for `employee1` whose name is "Fritz Fisch", who is male and has the employee number 205482.

- Generate the output:

  ```
  Employee: Fritz Fisch, number = 285942, sex = M
  ```

- Solution: #+begin_src C #define NAME_LEN 25
  / declare structure struct { char name[NAME_LEN+1]; int number; char sex; } employee1, employee2; / two employees

  / main program int main(void) { / use employee structure strcpy(employee1.name, "Fritz Fisch"); employee1.number = 285942; employee1.sex = 'M';

  printf("Employee: %s, number = %d, sex = %c", employee1.name, employee1.number, employee1.sex);

  return 0; } #+end_srcn

  ```
  Employee: Fritz Fisch, number = 285942, sex = M
  ```

Initializing structure variables

- A structure declaration may include an initializer.
- Non-initialized members are set to 0.
- Initializers can be *positional* (same order as in declaration), or *designated* (any order as long as they're named): In the code below, `part1` is initialized with a designator, while `part2` is not.

```
#define NAME_LEN 25
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {.name = "Disk drive", .on_hand = 10, .number = 528},
  part2 = {914, "Printer cable"};

printf("Part 1: %s, number = %d, on hand = %d\n"
 "Part 2: %s, number = %d, on hand = %d\n",
 part1.name, part1.number, part1.on_hand,
 part2.name, part2.number, part2.on_hand);
```

```
Part 1: Disk drive, number = 528, on hand = 10
Part 2: Printer cable, number = 914, on hand = 0
```

Operations on structures

- The members of a structure are *lvalues*: they can appear left of an assignment, or as the operand in an increment/decremend expression:
- Like an array, a structure variable can be initialized and declared at once:

```
#define NAME_LEN 25

struct {
  int number; // parts number
  char name[NAME_LEN+1]; // parts name - string + null character
```

```
  int on_hand; // part is on hand
} part1, part2; // two part variables // declaration of parts only

printf("part no. = %d\n",part1.number = 201);

part1.number++;

printf("part no. = %d\n",part1.number);


part no. = 201
part no. = 202
```

- The period to access a structure is a C operator. It takes precedence over nearly all other operators. Other C-like languages with user-defined structures or classes have this dot-operator, too.

  Python example:

  ```
  import numpy as np
  arr = np.array([1,2,3]) # use array method of numpy library
  print(arr)
  ```

- In the following statement, the argument contains two operators: the "dot" operator takes precedence: `&` computes the address of `part.num`:

```
scanf("%d", &part.num);
```

- We can show this by print address and value before and after the `scanf` command:

  ```
  echo "1000" > input
  cat input


  1000

  // declare structure
  struct {
    int num;
  ```

```
} part = {.num = 999};

// print structure member and address before user input
printf("%p %d\n", &part.num, part.num);

// get user input
scanf("%d", &part.num);

// print user input and address of user input
printf("%p %d\n", &part.num, part.num);


0x7ffc640cd754 999
0x7ffc640cd754 1000
```

- Though arrays cannot be copied using =, structures can!

```
#define NAME_LEN 25

struct {
  int number; // parts number
  char name[NAME_LEN+1]; // parts name - string + null character
  int on_hand; // part is on hand
} part1, part2; // two part variables
part1.number=415;
strcpy(part1.name,"Keyboard");
part1.on_hand=20;

printf("Part 1: %s, number = %d, on hand = %d\n"
 "Part 2: %s, number = %d, on hand = %d\n",
 part1.name, part1.number, part1.on_hand,
 part2.name, part2.number, part2.on_hand);

part2 = part1; // copy one structure into another

printf("Part 1: %s, number = %d, on hand = %d\n"
 "Part 2: %s, number = %d, on hand = %d\n",
 part1.name, part1.number, part1.on_hand,
 part2.name, part2.number, part2.on_hand);
```

```
Part 1: Keyboard, number = 415, on hand = 20
Part 2: , number = 2, on hand = 4096
Part 1: Keyboard, number = 415, on hand = 20
Part 2: Keyboard, number = 415, on hand = 20
```

- You can use this to copy arrays with dummy structures:

```
struct { int a[10]; } a1={1}, a2; puts("a1:");
for(int *p=a1.a;p<a1.a+10;p++) printf("%d ",*p);
puts("\na2:");
for(int *p=a2.a;p<a2.a+10;p++) printf("%d ",*p);
a2 = a1; puts("\na2:");
for(int *p=a2.a;p<a2.a+10;p++) printf("%d ",*p);
```

```
a1:
1 0 0 0 0 0 0 0 0 0
a2:
2 0 -1075053569 0 -1736518487 32764 100 0 4096 0
a2:
1 0 0 0 0 0 0 0 0 0
```

- No other operations but `=` are available. In particular, there is no way to compare structures with logical operators (`==` and `!=`).
- The `=` operator only works if the structures types *compatible*, which means that they must be declared at the same time.

Structure types

- We need to define a name that represents the *type* of structure, not a particular (anonymous) structure *variable*.
- We can either define a *structure tag* or use *typedef* to define a type name for our structure:
- This example declares a structure tag named `part`:

```
#define NAME_LEN 25
struct part {
```

```
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};  // semi-colon must terminate the declaration
```

- The tag can now be used to declare variables:

```
#define NAME_LEN 25
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};  // semi-colon must terminate the declaration

struct part part1, part2; // must be prefixed by ‘struct‘
```

- Declaration of a tag and of structure variables can be combined:

```
#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {100,"keyboard",10};

struct part part2;
strcpy(part2.name,"screen");

printf("Parts: %s and %s", part1.name,part2.name);


Parts: keyboard and screen
```

- Alternatively, use `typedef` to define a genuine data type. All `Part` variables, no matter when they're declared, are compatible.

```
#define NAME_LEN 25

typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;  // define a type ‘Part‘

Part part1, part2; // declare variables
```

- Declaring a structure tag is mandatory when the structure is used in a linked list.

Practice: Creating a structure tag

1. Declare a structure `Book` that has the following members:
   - An integer `id`.
   - A string `title` (with a maximum length of 50).
   - A float `price`.
2. Declare and initialize two `Book` variables:
   - `book1` with the `id` 101, `title` "C Programming", and `price` 29.99.
   - `book2` without initialization.

   Solution:

   ```
   struct Book {
     int id;
     char title[50];
     float price;
   } book1 = {.id = 101, .title = "C Programming", .price = 29.99},
     book2;
   ```

3. Write a `main` program. In it:
   - Define `book2` with `id` 102, `title` "Data Structures", and `price` 39.99.

- Print `book1` and `book2` details.

```
struct Book {
  int id;
  char title[50];
  float price;
} book1 = {.id = 101, .title = "C Programming", .price = 29.99},
  book2;

int main()
{
  book2.id = 102;
  strcpy(book2.title,"Data Structures");
  book2.price=39.99;

  printf("Book1: id = %d, title = %s, price = %g\n",
    book1.id, book1.title, book1.price);
  printf("Book2: id = %d, title = %s, price = %g\n",
    book2.id, book2.title, book2.price);
  return 0;
}
```

```
Book1: id = 101, title = C Programming, price = 29.99
Book2: id = 102, title = Data Structures, price = 39.99
```

Structures as arguments and return values

- Functions may have structures as arguments and return values.
- Example: This function, when given a `Part` structure as its argument, prints the structure's members:

```
// Define a structure with a tag
struct Part {
  int number;
  char name[50];
  int on_hand;
};
```

```
// Function to print a Part object p
// No return value
void print_part(struct Part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}

int main(int argc, char *argv[])
{
  // Initialize part1
  struct Part part1 = {101, "Keyboard", 20};

  // Print part1 details
  print_part(part1);
  return 0;
}
```

```
Part number: 101
Part name: Keyboard
Quantity on hand: 20
```

- The second function returns a `Part` structure that it constructs from its arguments:

```
// Define a structure with a tag
struct Part {
  int number;
  char name[50];
  int on_hand;
};

struct Part build_part(int number, const char *name, int on_hand)
{
  struct Part p;  // declares part as Part

  p.number = number;
  strcpy(p.name, name);
```

```
  p.on_hand = on_hand;

  return p; // returns part
}

int main(int argc, char *argv[])
{
  // Create a Part using the build_part function
  struct Part part1 = build_part(101, "Keyboard", 20);

  // Print part details
  printf("Part number: %d\n", part1.number);
  printf("Part name: %s\n", part1.name);
  printf("Quantity on hand: %d\n", part1.on_hand);

  return 0;
}
```

```
Part number: 101
Part name: Keyboard
Quantity on hand: 20
```

- Passing a structure to a function and returning one requires making copies of all members in the structure, which imposes an overhead.
- To avoid this overhead, it can be useful to pass a pointer to a structure, or have a function return a pointer instead of the structure itself.
- The `<stdio.h>` header defines a type `FILE`, which is a structure, and stores information about the (unique) state of an open file.
- Every function in `<stdio.h>` that opens a file returns a pointer to `FILE`, and every function working on an open file requires a `FILE` pointer as an argument.

Practice: Function to print structure details

Write a program that:

1. Declares a `Book` structure with the following members: integer book ID), book title (max 100 characters), and book author (max 50 characters).

```
struct Book {
  int id;
  char title[100];
  char author[50];
};
```

2. Implements a function `print_book` to print the book's details in a formatted manner.

```
struct Book {
  int id;
  char title[100];
  char author[50];
};
  // function to print book details
  // returns: nothing
  // params: Book structure
void print_book(struct Book b)
{
  printf("Book ID: %d\n", b.id);
  printf("Book Title: %s\n", b.title);
  printf("Author: %s\n", b.author);
}
```

3. Initializes a `Book` structure in the `main` function and calls `print_book` to display its details.

   Expected output:

```
Book ID: 201
Book Title: The C Programming Language
Author: Brian W. Kernighan and Dennis M. Ritchie
```

Solution:

```
// declare Book structure
struct Book {
  int id;
  char title[100];
  char author[50];
};

// function to print book details
// returns: nothing
// params: Book structure
void print_book(struct Book b);

// main program
int main(int argc, char **argv)
{
  // initialize a book structure
  struct Book book1 = {
    .id=201,
    .title="The C Programming Language",
    .author="Brian W. Kernighan and Dennis M. Ritchie"
  };

  // print book details using print_book
  print_book(book1);

  return 0;
}

void print_book(struct Book b)
{
  printf("Book ID: %d\n", b.id);
  printf("Book Title: %s\n", b.title);
  printf("Author: %s\n", b.author);
}
```

```
Book ID: 201
Book Title: The C Programming Language
Author: Brian W. Kernighan and Dennis M. Ritchie
```

Bonus practice: Return structure from function

Write a program that:

1. Defines a `Student` structure with the following members:
   - `int roll_no` (for roll number)
   - `char name[50]` (for student name)
   - `float marks` (for marks)
2. Implements a function `struct Student create_student(int roll_no, const char *name, float marks)` that:
   - Takes roll number, name, and marks as arguments.
   - Constructs and returns a `Student` structure with the provided values.
3. In the `main` function, uses `create_student` to initialize a `Student` structure and prints its details.

Expected Output (Example):

```
Student Roll No: 101
Student Name: John Doe
Marks: 92.5
```

Nested arrays with structure elements

- Structures and arrays can be combined without restriction.
- This tagged structure can store a person's first name, middle initial, and last name.

```
#define FIRST_NAME_LEN 20
#define LAST_NAME_LEN 50

struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

- You can use `person_name` as part of a larger structure now:

```
#define FIRST_NAME_LEN 20
#define LAST_NAME_LEN 50

struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
}; // a person's first, last name and middle initial

struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- You access a `student`'s name parts with two dot operators:

```
#define FIRST_NAME_LEN 20
#define LAST_NAME_LEN 50

struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
}; // a person's first, last name and middle initial

struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2; // student's age, sex and name, two students defined

int main()
{
  // student1's first person_name.first
  strcpy(student1.name.first, "Fred");
  printf("The student is called %s.\n", student1.name.first);
  return 0;
```

```
}
```

```
The student is called Fred.
```

- Why not just add the name details to `student`?

    1. You can treat names more easily as units of data: If you write a function that displays a name, you could pass just one argument, a `person_name` structure, instead of three arguments.
    2. You can copy the information from a `person_name` structure to the `name` member of a `student` in one instead of three assignments.

- Example:

```
#define FIRST_NAME_LEN 20
#define LAST_NAME_LEN 50

struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
}; // a person's first, last name and middle initial

struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2; // student's age, sex and name, two students defined

void display_name(struct person_name name)
{
  printf("Student's name: %s %c. %s\n",
   name.first, name.middle_initial, name.last);
}

int main()
```

```
{
  // initialize student's name
  struct person_name name;
  strcpy(name.first, "Jane");
  name.middle_initial = 'D';
  strcpy(name.last, "Doe");

  // assign name to student1
  student1.name = name;

  // display student1's name
  display_name(student1.name);

  // initialize student's new name
  struct person_name new_name;
  strcpy(new_name.first, "Jane");
  new_name.middle_initial = 'D';
  strcpy(new_name.last, "Zane");

  // assign new name to student1
  student1.name = new_name;

  // display student1's new name
  display_name(student1.name);

  return 0;
}
```

```
Student's name: Jane D. Doe
Student's name: Jane D. Zane
```

Arrays of structures

- Arrays whose elements are structures are very common, and can serve as a simple database.
- Example: This array of `part` structures can store information about 100 parts.

```
struct part inventory[100];
```

- To access on of the parts you use subscripting: This prints the `part` stored in position `i`:

  ```
  print_part(inventory[i]); // array element contains i-th ‘part‘ structure
  ```

- To assign a `number` within a `part` structure, combine subscripting and member selection:

  ```
  inventory[i].number = 883; // changes number of i-th part to 883
  ```

- To assign a character in a `part` name, combine subscripting, selection, and subscripting again:

  ```
  inventory[i].name[0] = '\0'; // changes ‘name‘ of i-th part to \0
  ```