C-Lessons

Complex data types

Lecturers: Mirko Jantschke, Pascal Scholz

19. Dezember 2018

Contents

Composite data types

Limits of primitive data types

Primitive data types are fine as long as you want to

- Store a single value that does not depend on other variables
- Store a sequence of values of the same type with a constant length
  $\rightarrow$ *arrays*

However, it is not possible to

- Compose variables of different data types to a compound structure
  $\rightarrow$ *composite data types*
- Have a variable that can only attain certain values
  $\rightarrow$ *enumerations*
- Have a sequence with an adjustable length
  $\rightarrow$ soon...

Data records

Composite data types are derived from primitive data types. You can store any number of primitive variables in one composite variable.

- The composite variable is called *structure* and has the type *struct*
- The primitive variables are called *members* of that structure

Defining a new composite type "*struct* person":

```
struct person {       /* struct <identifier> */
    int id;
    int age;          /* block for member declaration */
    char name[32];
};                    /* end declaration with ';' */
```

A *struct* variable is at least as large as all of its members.

struct variables

Our new type *struct person* can be used to declare variables any where in its scope:

```
struct person pers_alice, pers_bob;
```

You can declare a *struct* variable directly in the type definition:

```
struct person {
    /* member declaration */
} pers_alice, pers_bob;
```

If we do not need the struct type *person* for further variable declarations, its identifier can be left out.

Definition and member access

To initialize the *struct* members upon declaration, enclose the values in braces as we did it for arrays:

```
struct person pers_alice = { 1, 20, "Alice" };
```

To access the struct members, use the struct identifier followed by a '.' and the member identifier:

```
printf("%d\n", pers_alice.id);
pers_alice.age++;
```

structs as *struct* members

An address is rather complicated:

```c
struct address {
    int postcode;
    /* ... imagine much more members */
};
```

Now, let the *person* have one:

```c
struct person {
    struct address contact;
    /* ... and all the other members */
} pers_alice;
```

Access:

```c
pers_alice.contact.postcode = 15430;
```

unions

- Size of the union is equal to the size of the largest member
- All member variables share the same memory
- Everytime you access a member, you access the same memory
- If you assign a value to a member, all other members become invalid (as they share the same memory)

Let's assume the follwing union:

```c
union compound {
    int a;      /* size = 32 bit */
    float b;    /* size = 32 bit */
    char c[8];  /* size = 64 bit */
};
```

$\rightarrow$ This union will be 64bit large.

unions

Now let's see what happens, if we use the union:

```c
/* cool stuff */
    union c u;
    u.a = 0;
    u.b = 0;
    printf("C: %s\n", u.c);
    char str[8] = {'a','b','c','d','e','f','g','\0'};
    strcpy(u.c, str); /* Function from string.h */
    printf("A: %d\n", u.a);
    printf("B: %f\n", u.b);
    printf("C: %s\n", u.c);
/* more cool stuff */
```

And the Output will be:

```
    C:                          // prints '\0'
    A: 1684234849bvgvf
    B: 1677799408082104352768.000000
    C: abcdefg
```

Enumerations

Smart aliases

An enumeration consists of identifiers that behave like *constant values*.
It is declared using the keyword *enum*:

```
enum light {
    RED,
    YELLOW,
    GREEN
};
```

Now you can assign the values *red*, *yellow* and *green* to variables of the type *enum light*. Internally they are represented as numbers ($red = 0$, $yellow = 1$ etc.), but

- Using the aliases is clear and fancy
- No invalid values (like *-1*) can be assigned

Profit

You can determine the values of the constants on your own:

```
enum workday {
    MONDAY,          /* 0 */
    TUESDAY,         /* 1 */
    THURSDAY = 3,    /* 3 */
    FRIDAY           /* 4 - implicit (predecessor + 1) */
};
```

However, this can confuse people → only use it if there is a good reason.

Enumerations provide a nice way to define ”global”constants:

```
enum { WIDTH = 10, HEIGHT = 20 };
...
char tetris_board[WIDTH][HEIGHT];
```

Style

Consistency

- Since complex type definitions heavily rely on blocks, you should use the same coding conventions on them
- Let your custom type identifiers start with small letters

If you define a complex data type, you are very likely going to use it in many different parts of your program.
→ Have a global type definition, declare the variables in the local context

Name *enum* constants in CAPITAL letters to visually seperate them from variables.

typedef

Sometimes you see people writing code like that:

```
typedef struct foo {
    /* member declarations */
} bar;
```

This creates the new type *bar* which is nothing more than a *struct foo*.

However, this simple fact is hidden for other programmers working on the same project → **possible confusion**.

- Unclear, if *bar* is a composite type at all
- If so, is it a *struct* / *union* / *enum* or something really crazy?

Related Task

Cristmas Task

Solution to this task

Christmas Task: Write a program, that calculates if the reindeers of santas sledge have enough pullforce to pull the weight of the presents. Therefor we need 3 Structs (for example data see next slide):

- struct reindeer, that contains age, gender and pullforce as members
- struct present, that contains weight and price as members
- struct sledge, that contains max. 8 reindeers, max. 20 presents and a maxTransportWeight as members

For the gender of the reindeer use an enum with m or w.

If the reindeers dont have the pullforce to pull all the presents there should also be printed how many kilos are left. So santa knows how many and which of his reindeers he has to add.

Cristmas Task 2

Examples data for struct presents:

- Computer: weight = 7kg, price = 850€
- Bookshelf: weight = 55kg, price = 519€
- Fridge: weight = 150kg, price = 750€

Expample data for reindeer:

- rudolph: age = 12, gender = m, pullforce = 50kg

Bonus: Santa also has to pay a 2 percent fee for his insurance of the worth of his cargo. Calculate the total insurance he has to pay!

Circles

Task as online

A point in 2D consists of two positions: x and y (both float). A circle consists of a centre (point), a radius, a circumference and an area (all float).

Write a program that reads two positions and a radius from the command line. They should be stored in a struct circle as described above along with its remaining parameters (calculate them!). Then, print the resulting circle.

Experts: Write a function that takes two circles as arguments and returns a circle that goes through their centres.