day-6-procedural-programming-code-organisation

September 13, 2023

1 Day 6 Procedural programming and code organisation

1.0.1 Topics

- Scalar product
- Recursion
- Libraries
- Encapsulation **(challenging)**

1. Scalar product A vector is an ordered collection of values and its dimension is the number of values it contains. In a file `scalar.c`, put a global variable with file scope `static int dim` that will keep track of what dimension vectors the program is currently using. Add a function `set_dim(int d)` to `scalar.c`, which sets `dim` to the value `d` and a function `get_dim(void)`, which returns the current value of `dim`.

The scalar product of two vectors is obtained by multiplying the elements entry by entry and then summing them up e.g. `(1,2,3)*(4,5,6)=1*4+2*5+3*6=4+10+18=32`. Write a function `int scalar_product(int *v1, int *v2)` that finds the scalar product of the vectors `v1` and `v2` and returns the result.

Add a `main()` function, which:

1. Takes one command-line argument and sets the dimension to this value.
2. Asks the user to type in the values of two vectors of this dimension.
3. Uses `scalar_product()` to calculate the scalar product of the two vectors and prints this value out.

The output when run should look like this:

```
$ gcc -Wall -Wextra scalar.c -o scalar
$ ./scalar 3
Input first vector: 1 2 3
Input second vector: 4 5 6
The scalar product is 32
```

[ ]:
```
//// scalar.c

int main(void) {

}
```

2. Recursion Write a **forward recursive** function that calculates the factorial of an input n.

What does the call-stack of this forward recursive function look like?

[ ]:
```c
//// forward.c

int main(void) {
    return 0;
}
```

Exercise: Write a **backward recursive** function that calculates the factorial of an input n.

What does the call-stack of this backward recursive function look like?

[ ]:
```c
//// backward.c

int main(void) {
    return 0;
}
```

3. Libraries Take your code from `scalar.c` and move the `main()` function to a new file `main.c`. Move the implementations of `set_dim(int d)`, `get_dim(void)` and `scalar_product(int *v1, int *v2)` to `scalar-lib.c`. Create a `scalar.h` file and add suitable `#include`s so that `main()` can call the functions in `scalar-lib.c`. If you compile with:

```
gcc -Wall -Wextra main.c scalar-lib.c -o main
```

then running `./main` should work as before.

[ ]:
```c
//// scalar.h
```

[ ]:
```c
//// scalar-lib.c
```

[ ]:
```c
//// main.c
//% DEPENDS scalar-lib.c
#include "scalar.h"

int main(void) {
    return 0;
}
```

Next:

1. Compile `scalar-lib.c` as a static library `libscalar.a` and compile your executable by linking to this library statically. Check that the program still works.
2. Compile `scalar-lib.c` as a dynamic library `libscalar.so` and compile your executable by linking to this library dynamically. You should find that your program only runs if `LD_LIBRARY_PATH` has been set appropriately, or you specify `LD_PRELOAD=libscalar.so` when running your program:

   ```
   LD_PRELOAD=libscalar.so ./main
   ```

4. Encapsulation (challenging) The next example demonstrates the concept of *encapsulation* across code boundaries (i.e. between source files or libraries).

A common method of encapsulation in C and C++ is to use an opaque type. This is a type which is *declared* in an interface to some library code (i.e. the library's header file) but is not *defined* in the interface. Instead it is *defined* in the library's source code. This means that the internal details of the type are not visible to the calling code. The calling code may only use the opaque type by declaring and handling pointers to the opaque type.

An example of an opaque type from the standard libary is the `FILE` type declared in `stdio.h` - users of the `FILE` type can only declare pointers to it, and must interact with it using library functions like so:

```
// 'f' is an opaque type whose implementation I can't see
FILE *f = NULL;

// code inside the standard libary knows about the implementation
f = fopen("a_file.txt", "r");

// this means the user doesn't need to worry about the internal details
fclose(f);
```

One drawback of using opaque types is that they **must** be dynamically allocated, since the size of the type is not known to the compiler when compiling code which usus the opaque type.

Write a 2nd version of your scalar product program so that the header file `vector.h` declares `struct Vec` as an opaque type. Declare the following functions in `vector.h` and define them in `vector.c`:

- `struct Vec *vec_alloc(void)` to dynamically allocate and return an empty `struct Vec`,
- `void vec_dealloc(struct Vec *v)` to de-allocate a `struct Vec` instance,
- `void vec_init(struct Vec *v, int dim)` to initialise `v` to have dimension `dim` and populate it with `dim` values input by the user,
- `void vec_fprint(struct Vec *v, FILE *stream)` to print a `struct Vec` to the given stream e.g. `stdout`,
- `int vec_scalar_product(struct Vec *v1, struct Vec *v2)` to return the scalar product of the vectors `v1` and `v2`.

The definition of the `Vec` struct should only appear in `vector.c`. Putting this definition in the source file instead of the header means that the internal details of the type are hidden from client code which uses `vector.h`.

Use these functions in `test-vector.c` to allocate, populate and de-allocate `struct Vec` instances using user input.

[ ]:
```
//// vector.h

struct Vec; // declare an opaque type i.e. this type exists but is not defined in this file.
```

```
struct Vec *vec_alloc(void);        // dynamically allocate and return an empty `struct Vec`
```

[ ]:

```
//// vector.c
#include <stdlib.h>
#include "vector.h"

struct Vec {
    // this is where you actually define struct Vec.
};

struct Vec *vec_alloc(void) {
    // dynamically allocate and return an empty `struct Vec`
    return NULL;
}
```

[ ]:

```
//// test-vector.c
//% DEPENDS vector.o
#include "vector.h"

int main(void) {
    struct Vec *v1 = vec_alloc();
    return 0;
}
```