This course has already ended.

Sixth round tasks as a `ZIP file.`

Structured data types (*struct*) can be used to combine variables related to each other and aggregate it into a single data type, which after structure definition can be used much like other types of data. It is common to use structured data types in simple to complex C programs.

In this section, you are going to learn how to use structured data types through pointers and array of structures. The section ends with **linked list**, which is a commonly used dynamic data structure in C programs. Implementation of a **linked list** is also a good exercise to learn structured data types.

Abstract data types¶

Often a good programming practice is to hide details of data management into abstract data types that allow the handling only through a published interface, that hides the internal implementation and is not exposed. This allows different implementations (or different algorithms) for operating the data type, while the implementation details remain hidden from the users of the interface. This also makes evolution of implementations easier, because changes on one side of the interface do not require modifications on the other.

The C language does not provide built-in mechanisms for abstract data types, but they are implemented by coding conventions and particular use of C headers, together with the **typedef** declaration that allows elegant representation of the types.

The public header of an abstract data type defines the interface (functions, data types, constants) that can be used to access the data type, but typically does not have to reveal the internal composition or operation of the data, because the user of the interface is typically not supposed to access it directly. Below is an example of a Vector type that can be used to handle euclidean vectors familiar from mathematics. The defintions would be given in a separate header file "vector.h".

```
#ifndef AALTO_VECTOR_H
#define AALTO_VECTOR_H

/* Define Vector type to be used in interfaces, but don't show
   its internal structure */
typedef struct vector_st Vector;

/* Allocates a new vector from heap, returns pointer to the vector */
Vector* vectorCreate(double x, double y, double z);

/* Releases the vector from heap */
void vectorFree(Vector *v);

/* Calculates sum of two vecors. Result is returned in new vector object
   allocated from heap */
Vector* vectorSum(const Vector *a, const Vector *b);

/* Calculates the length of the vector */
double vectorLength(const Vector *v);

/* Prints the vector to standard output */
void vectorPrint(const Vector *v);

#endif // AALTO_VECTOR_H
```

This is also an example of an include guard: the **#ifndef** and **#define** precompiler instructions are used to ensure that the same definitions are not included multiple times during single compilation process, which would lead to compile errors due to duplicate declarations. This could happen with larger programs with nested include dependencies (the precompiler functionality will be discussed in later modules). Then, a forward declaration of struct vector_st is done, with a typedef mapping to Vector data type. The header also defines function interfaces for operating the vector. Note that the public header does not tell anything about how **struct vector_st** is defined, or how functions are implemented. The usage of const qualifier is also important in function interfaces: it is used to tell the calling program that the functions are not going to modify the value of vector.

Next, the implementation of vector is given in C source file (let's call it "vector.c"). The users of vector do not need to know how operations are implemented, and they might not even have access to the source code, although an object file or library of the implementation is needed in order to build running program. Here is part of the vector implementation:

```
#include <assert.h>
#include "vector.h" // to ensure that the interface is compatible

struct vector_st {
    double x, y, z;
};

Vector* vectorCreate(double x, double y, double z)
{
    Vector *v = malloc(sizeof(struct vector_st));
    v->x = x;
    v->y = y;
    v->z = z;
    return v;
}

void vectorFree(Vector *v)
{
    assert(v);  // fails if v is NULL
    free(v);
}

Vector* vectorSum(const Vector *a, const Vector *b)
{
    assert(a); // check that value is not NULL
    assert(b != NULL); // other way to check that value is not NULL
    Vector *ret = malloc(sizeof(struct vector_st));
    ret->x = a->x + b->x;
    ret->y = a->y + b->y;
    ret->z = a->z + b->z;
    return ret;
}
/* ...continues with some other Vector management functions... */
```

The above implementation uses the **assert** macro for checking that the pointers passed to functions are not **NULL** (which would indicate some sort of error in calling code). The macro is defined in "assert.h" header, and can be used to verify given conditions (assertions) about the variables. If assert condition fails, the program aborts showing the location of failing assertion. Asserts are a good tool for verifying strong assumptions about the parameters passed to the interface, when failure should be considered an error in the program that uses the interface.

The other parts of C source (in different C source files) can use the vector interface as defined in "vector.h". It is sufficient for the other parts of the source to just see the vector.h header. After the compiler makes object files of each C source file, the linker combines them into single executable. The internal vector implementation can be modified in "vector.c", and the other parts of the code do not need to be modified, as long as the interface remains unchanged.

```
#include "vector.h"

int main(void)
{
    // create two vectors and calculate their sum into third vector
    Vector *v1 = vectorCreate(1, 3, 4);
    Vector *v2 = vectorCreate(0, -2, 2);
    Vector *v3 = vectorSum(v1, v2);

    // print the result
    vectorPrint(v3);

    // release all three vectors that were allocated
    vectorFree(v1);
    vectorFree(v2);
    vectorFree(v3);
}
```

Task 6.3: Fraction¶

We practice abstract data types by implementing a new number type, fraction, that consists of numerator and denominator. For example 2/3 and 4/6 are equivalent values, where 2 and 4 are numerators, and 3 and 6 are denominators. A new data type, Fraction is used to represent fractions. First you need to define the fraction_st struct as needed in the **fraction.h** file. Functions need to be declared in the header file (**fraction.h**) and defined in the source file (**fraction.c**).

In this task, you will implement the functions listed below.

(a) Set value¶

Implement function `Fraction* setFraction(unsigned int numerator, unsigned int  denominator)` that allocates a new Fraction from heap, sets its value as given in parameters, and returns the created object.

In addition, implement also the following simple functions:

- `void freeFraction(Fraction* f)` that releases the memory used by the Fraction.
- `unsigned int getNum(const Fraction *f)` that returns the numerator part of the fraction.
- `unsigned int getDenom(const Fraction *f)` that returns the denominator part of the fraction.

(b) Compare values¶

Implement function `int compFraction(const Fraction *a, const Fraction *b)` that returns -1 if a < b, 0 if a == b, or 1 if a > b. The function should work correctly also when denominators are different.

(c) Add values¶

Implement function `Fraction *addFraction(const Fraction *a, Fraction *b)` that adds values 'a' and 'b', and returns the result in a new object allocated from heap. Again, the denominators may be different in the two numbers. The resulting value does not need to be reduced to smallest possible denominator.

Hint: Start by modifying the two numbers such that they have the same denominator.

(d) Reduce value¶

Implement function `void reduceFraction(Fraction *val)`, that reduces the value to the smallest possible denumerator. For example, using this function 3/6 should reduce to 1/2. For doing this, you'll need to find the greatest common divisor for the numerator and denominator. The exercise template file **fraction.c** contains function `unsigned int gcd(unsigned int u, unsigned  int v)` (source: wikipedia), that you can use from your function.)

Points 25 / 25 My submissions 2 Deadline Friday, 16 July 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Fraction

Select your files for grading

fraction.h

Choose File No file chosen

fraction.c

Choose File No file chosen

Submit