# DVA104

## Data Structures, Algorithms, and Program Design.

Gabriele Capannini

gabriele.capannini@mdh.se

Mälardalen University

July 17, 2018

# Abstract Data Type (ADT)

- An abstract data type is not a concept only in the C language, but a general programming concept.
- An ADT is designed by keeping in mind **how it is used** and not how it is implemented.
- The intention is to abstract details: an important goal in programming is to avoid thinking as much as possible.
- Example. . .

# Abstract Data Type (ADT) - Example

We need an ADT (struct) that characterizes a student.
Information we want to store are:

- Name,
- Student Id,
- Credits.

Then we can have functions to edit data for a given student, for example, to increase the number of credits.

We design the data type by choosing what information to store about each student.

# A new (abstract) approach to data types

Instead of trying to state what we want to store for a student, it is often more useful to think what we want to do with a student.

Example:

- We want to be able to create a new student.
- We want to be able to print the name of a given student.
- We want to be able to add higher education credits.
- We don´t need to be able to reduce college credits.
- What do we want more to do with a student?

# A new (abstract) approach to data types

When you design an ADT:

- Focus on what you can do with the type. . . not what it actually contains
- This is called the interface for a data type

We try to design the Student type completely without assuming anything about what it looks like: we do not even assume it's a struct! Why?

- The most important thing when designing data types is usually how it is used!
- For example, when using files, you really do not know what FILE is, you just know how to use it.

# ADT - Interface

Every action performed on an instance of a given data type is called operation. In C, operations are implemented using functions.

Our Student interface consists of three operations (functions):

- Create a new student : `createNewStudent()`
- Print the name of a student : `printName()`
- Adding a number of credits to a student : `addCredit()`

# ADT - Interface (continued)

The next step is to see what information these functions need, **without** assuming anything about the data type itself.

Create a new student:

- What should the function do?
- What information does it need (argument)?
- What should the function return?

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Create a new student:

- What should the function do? **Return a new variable of type Student.**
- What information does it need (argument)?
- What should the function return?

```
ReturnType createNewStudent(ArgumentList)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Create a new student:

- What should the function do? Return a new variable of type Student.
- What information does it need (argument)? **We need to know name and id.**
- What should the function return?

```
ReturnType createNewStudent(char *name, char *id)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Create a new student:

- What should the function do? Return a new variable of type Student.
- What information does it need (argument)? We need to know name and id.
- What should the function return? **The new Student.**

```
Student createNewStudent(char *name, char *id)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Create a new student:

- What should the function do? Return a new variable of type Student.
- What information does it need (argument)? We need to know name and id.
- What should the function return? The new Student.

**Note that we haven't implemented the function yet!**

```
Student createNewStudent(char *name, char *id)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Print the name of a student:

- What should the function do?
- What information does it need (argument)?
- What should the function return?

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Print the name of a student:

- What should the function do? **Print the student name (obviously).**
- What information does it need (argument)?
- What should the function return?

```
ReturnType printStudent(ArgumentList)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Print the name of a student:

- What should the function do? Print the student name (obviously).
- What information does it need (argument)? **A student.**
- What should the function return?

**Note that we do not send a pointer: we don't neither know if it is a struct or not, nor if the function is able to change the student (which is doable if we pass a pointer).**

```
ReturnType printStudent(Student student)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Print the name of a student:

- What should the function do? Print the student name (obviously).
- What information does it need (argument)? A student.
- What should the function return? **Nothing.**

```
void printStudent(Student student)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Add some credits of a student:

- What should the function do?
- What information does it need (argument)?
- What should the function return?

**This function should change the state of a student. Usually this is done by passing a pointer to the target variable.**

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Add some credits of a student:

- What should the function do? **Update student credits.**
- What information does it need (argument)?
- What should the function return?

```
ReturnType addCredits(ArgumentList)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Add some credits of a student:

- What should the function do? Update student credits.
- What information does it need (argument)? **The pointer to the target student and how many points to add.**
- What should the function return?

```
ReturnType addCredits(Student *student, float credits)
```

# ADT - Interface (continued)

The next step is to see what information these functions need, without assuming anything about the data type itself.

Add some credits of a student:

- What should the function do? Update student credits.
- What information does it need (argument)? The pointer to the target student and how many points to add.
- What should the function return? **Nothing. . . We just want to set some data.**

```
void addCredits(Student *student, float credits)
```

# ADT - Interface (end)

It's done! We have an **interface** for the ADT Student.

```
Student createNewStudent(char *name, char *id);
void printStudent(Student student);
void addCredits(Student *student, float credits);
```

We have designed what we **can do** with a student without worrying neither about how data is stored nor implementing the underlying code.

# Interface in C

Interface can be placed in a `.h` file.

```
#ifndef STUDENT_H
#define STUDENT_H

/* Student is not defined yet */

Student createNewStudent(char *name, char *id);
void printStudent(Student student);
void addCredits(Student *student, float credits);

#endif
```

# Interface in C (continued)

We can now assume that there is a `student.c` file containing the function definitions. Notice that we do not need to know that file to use students!

```
#include "student.h"

int main() {
   /* Create two students */
   Student studentA, studentB;
   studentA = createNewStudent("Alice", "awd16001");
   studentB = createNewStudent("Bob", "bbd15004");
   /* Modify the credits */
   addCredits(&studentB, 7.5);
   /* Print the name */
   printStudent(studentB);
}
```

# Interface in C (continued)

Note that we can easily understand how this program works even if we don´t know anything about either its functions or type of student!

- This is what you want to achieve in programming!
- We have abstracted the details!

# ADTs: recap

However, to make it working, we must both define the type of student and implement the all functions!

Therefore an abstract data type consists of:

- an interface,
- an implementation of the interface (definition of data types and functions),

in C, you can do this as it follows:

- the interface is placed in a `.h` file (visible to the person to use data type),
- implement the implementation in the `.c` file (hidden for the one who will use the data type)... However, for technical reasons, you could have parts of it in the `.h` file (e.g., a `typedef`).

# Implementation

The implementation is advantageously added in a `.c` file.

Usually, the implementation contains the definitions of:

- any used struct,
- the functions declared by the interface in the `.h` file.

# Implementation of `Student`

In order to write the functions we need to know how the real data type student looks.

Guidelines for the specific type:

- It should be as simple as possible.
- It should only contain information about exactly that needed to implement the functions.

# Implementation of `Student` (continued)

Implementation is based on the operations to perform:

```
Student createNewStudent(char *name, char *id);
```
What does `Student` need for this function?

```
void printStudent(Student student);
```
What does `Student` need for this function?

```
void addCredits(Student *student, float credits);
```
What does `Student` need for this function?

# Implementation of `Student` (continued)

Implementation is based on the operations to perform:

```
Student createNewStudent(char *name, char *id);
```

**Student needs a name (`string`) and an id (`string`)**

```
void printStudent(Student student);
```

What does `Student` need for this function?

```
void addCredits(Student *student, float credits);
```

What does `Student` need for this function?

# Implementation of `Student` (continued)

Implementation is based on the operations to perform:

```
Student createNewStudent(char *name, char *id);
```

Student needs a name (`string`) and an id (`string`).

```
void printStudent(Student student);
```

**No other information but the student.**

```
void addCredits(Student *student, float credits);
```

What does `Student` need for this function?

# Implementation of `Student` (continued)

Implementation is based on the operations to perform:

```
Student createNewStudent(char *name, char *id);
```
Student needs a name (`string`) and an id (`string`).

```
void printStudent(Student student);
```
No other information but the student.

```
void addCredits(Student *student, float credits);
```
**Student needs credits (`float`).**

# Implementation of `Student` (end)

In the end `Student` is:

```
struct Student_s
{
  char name[20];
  char id[9];
  float credits;
};
```

Note that even if information about the fields of the structure belongs to the implementation, they are usually placed in the `.h` file for practical reasons.

# Alternative one: `student.h`

Here the struct is declared in the `.h` file while its definition is in the implementation.

```
#ifndef STUDENT_H
#define STUDENT_H

struct Student_s;

typedef struct Student_s Student;

Student createNewStudent(char *name, char *id);
void addCredits(Student *student, float credits);
void printStudent(Student student);

#endif
```

# Alternative one: `student.c`

```c
#include <stdio.h>
#include <string.h>
#include "student.h"

struct Student_s {
  char name[20];
  char id[9];
  float credits;
};

Student createNewStudent(char *name, char *id) {
  ...
}
void addCredits(Student *student, float credits) {
  ...
};
void printStudent(Student student) {
  ...
};
```

# Alternative two: `student.h`

Otherwise, both declaration and definition can stay in the interface.

```
#ifndef STUDENT_H
#define STUDENT_H

struct Student_s {
  char name[20];
  char id[9];
  float credits;
};

typedef struct Student_s Student;

Student createNewStudent(char *name, char *id);
void addCredits(Student *student, float credits);
void printStudent(Student student);

#endif
```

# Alternative two: `student.c`

Now the implementation contains only the functions.

```
#include <stdio.h>
#include <string.h>
#include "student.h"

Student createNewStudent(char *name, char *id) {
  ...
};
void addCredits(Student *student, float credits) {
  ...
};
void printStudent(Student student) {
  ...
};
```

# Pre- and Post-conditions

In programming it is important to know exactly what a function accomplishes.

Therefore, it is important to define this before even writing them. This can be done by means of so-called **pre-** and **post-conditions**.

- Pre-conditions state the requirements, i.e., what must be true before the function is called.
- Post-conditions state what will be true when the function return.

Example. . .

# Pre- and Post-conditions (continued)

```
Student createNewStudent(char *name, char *id);
```

Pre-conditions?

Post-conditions?

# Pre- and Post-conditions (continued)

```
Student createNewStudent(char *name, char *id);
```

Pre-conditions?

- `name` must point to a non empty string, i.e., `name!=NULL`
- `id` must point to a non empty string, i.e., `id!=NULL`
- ...

Post-conditions?

# Pre- and Post-conditions (continued)

```
Student createNewStudent(char *name, char *id);
```

Pre-conditions?

- `name` must point to a non empty string, i.e., `name!=NULL`
- `id` must point to a non empty string, i.e., `id!=NULL`

Post-conditions?

- None, the function returns a value, but nothing has changed.
- A new student with `name` and id `id` is created.

# Pre- and Post-conditions (continued)

```
void printStudent(Student student);
```

Pre-conditions?

Post-conditions?

# Pre- and Post-conditions (continued)

```
void printStudent(Student student);
```

Pre-conditions?

- Nothing, `student` is not a pointer but a variable, so there must be something stored in it.

Post-conditions?

# Pre- and Post-conditions (continued)

```
void printStudent(Student student);
```

Pre-conditions?

- Nothing, `student` is not a pointer but a variable, so there must be something stored in it.

Post-conditions?

- Nothing except for the name shown on the screen.

# Pre- and Post-conditions (continued)

```
void addCredits(Student *student, float credits);
```

Pre-conditions?

Post-conditions?

# Pre- and Post-conditions (continued)

```
void addCredits(Student *student, float credits);
```

Pre-conditions?

- `credits` must be positive.
- `student` is a pointer and must be different from `NULL`.

Post-conditions?

# Pre- and Post-conditions (continued)

```
void addCredits(Student *student, float credits);
```

Pre-conditions?

- `credits` must be positive.
- `student` is a pointer and must be different from `NULL`.

Post-conditions?

- Credits of the student pointed by `student` must be increased by `credits`.

# Implementation

- We concretely defined `Student` as a `struct`.
- We declared all operations with their pre- and post-conditions.
- Now we can implement all of them. . . Or pass the interface to someone else who can do the work for us.

# Implementation

Consider, for example, the implementation of the function `createNewStudent(...)`:

```
#include <stdio.h>
#include <string.h>
#include "student.h"
Student createNewStudent(char *name, char *id) {
  Student student;
  strcpy(student.name, name);
  strcpy(student.id, id);
  student.credits = 0;
  return student;
}
```

# Pre- and Post-condition in practice

There is a useful function (properly it is a macro) in C which does what we expect from pre- and post-conditions.

```
#include <assert.h>
void assert(expr);
```

`expr` can be a variable or any C expression. If `expr` evaluates to `true`, `assert`) does nothing. If expression evaluates to `false`, `assert` aborts program execution and displays an error message.

You can turn off every `assert()` in your program by adding the line `#define NDEBUG` in the code! *Where?*

# assert(...) example

```c
#include <stdio.h>
#include <assert.h>

int main() {
  int n;
  printf("Enter a number greater than zero: ");
  scanf("%d", &n);
  assert(n>0);
  printf("You entered %d\n", n);
}
```

if n is not grater than 0, the program ends in this way:

```
Enter a number greater than zero: 0
assert: assert.c:8: main: Assertion ‘n>0’ failed.
Aborted (core dumped)
```

# assert(...)+NDEBUG (example)

```
#include <stdio.h>
#define NDEBUG /*HERE!*/
#include <assert.h>

int main() {
  int n;
  printf("Enter a number greater than zero: ");
  scanf("%d", &n);
  assert(n>0);
  printf("You entered %d\n", n);
}
```

No check is performed now, the program goes on even if you enter, for example, 0:

```
Please enter a number greater than zero: 0
You entered 0
```

# Implementation (continued)

For the `createNewStudent(char *name, char *id)` function, we have the following pre-conditions:

1. `name` must point to a non empty string, i.e., `name!=NULL`
2. `id` must point to a non empty string, i.e., `id!=NULL`

That is. . .

```
Student createNewStudent(char *name, char *id) {
  assert(name!=NULL); /*Pre-condition 1*/
  assert(id!=NULL); /*Pre-condition 2*/
  Student student;
  strcpy(student.name, name);
  strcpy(student.id, id);
  student.credits = 0;
  return student;
}
```

# Implementation (continued)

For the `printStudent(Student student)` function, we have that:

```
void printStudent(Student student) {
  printf("%s\n", student.name);
};
```

No pre- or post-conditions to check. If an instance of `Student` has been created by means of our interface, `name` cannot be `NULL`.

Notice that in more recent languages (e.g., $C^{++}$) this aspect is enforced by forcing the programmer to use specific functions (so-called methods) for interacting with an instance of a struct.

# Implementation (continued)

For the `addCredits(Student *student, float credits)` function, we have the following pre-conditions:

1. `student` must point to a student, i.e., `student!=NULL`
2. `credits` must be positive

and post-conditions:

- credits of `student` (i.e., `student->credits`) must be `credits` greater than before

That is. . .

```
void addCredits(Student *student, float credits) {
  assert(student!=NULL); //Pre-condition 1
  assert(credits>0); //Pre-condition 2
  float c = student->credits;
  student->credits += credits;
  assert(student->credits==credits+c); //Post-condition
};
```

# Final result

The interface (with struct definition) in `student.h`:

```
#ifndef STUDENT_H
#define STUDENT_H

struct Student_s {
  char name[20];
  char id[9];
  float credits;
};

typedef struct Student_s Student;

Student createNewStudent(char *name, char *id);
void addCredits(Student *student, float credits);
void printStudent(Student student);

#endif
```

# Final result (continued)

The implementation in `student.c`:

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "student.h"

Student createNewStudent(char *name, char *id) {
  assert(name!=NULL && id!=NULL);
  Student student;
  strcpy(student.name, name);
  strcpy(student.id, id);
  student.credits = 0;
  return student;
}
void addCredits(Student *student, float credits) {
  assert(student!=NULL && credits>0);
  float c = student->credits;
  student->credits += credits;
  assert(student->credits==credits+c);
};
void printStudent(Student student) {
  printf("%s\n", student.name);
};
```

# ADT Summary

An Abstract Data Type consists of two components:

- Interface (public, primarily `.h` file), i.e., what do we want to do with the data type interface
- Implementation (hidden, mainly `.c` file)

Associated Concepts

- Pre- and Post-conditions: belong to functions.
- Asserts: can be used to ensure Pre- and Post-conditions.
- Abstraction: implementation and interface are independent (actually the same interface could even have more than one implementation).

MDH

# Implementation and interface are independent - example

Let us consider the `printStudent(Student student)` function:

```
void printStudent(Student student) {
  printf("%s\n", student.name);
};
```

Now we want to change it in order to show all the information related to `student`, i.e., name, id, and credits.

*What do we need to change? Do we need to update the interface?*

# Implementation and interface are independent - example (continued)

Answer: **No**, interface stays the same! We only need to re-implement the function in the `.c` file.

```
void printStudent(Student student) {
  printf("%s\n", student.name);
  printf("%s\n", student.id);
  printf("%.1f\n", student.credits);
};
```

# . . . About hiding information

Let us suppose we want to change the function for creating a new student by automatically calculating the `id` from the student name and the admission year. Given a function like:

```
char *createStudentID(char *name, int year);
```

The new implementation `createNewStudent()` looks like (interface has been changed accordingly):

```
Student createNewStudent(char *name, int year) {
  assert(name!=NULL && year>=0);
  char *id =  createStudentID(name, year);
  Student result;
  strcpy(result.name, name);
  strcpy(result.studentid, id);
  result.credits = 0.0;
  return result;
}
```

*Do we need to have even `createStudentID` in the interface?*

# . . . About hiding information (continued)

Answer: **NO**, `createStudentID` is an auxiliary function. As matter of fact:

- The student type (namely `Student`) does not appear in argument or return value of the function: this function is not intended for the “final user”.
- It is only a step in constructing a new student.

# Static functions

- A static function is a function that is intended to be used only in one file and it can not be used outside of the file it is defined in. In other words: a static function "does not exist" outside of the file
- Less functions to think/keep track of.
- It **doesn't** appear in the `.h` file (because nobody will see it).
- Useful for:
  - Auxiliary functions.
  - Functions that are not part of the interface of an ADT.

# Static functions (example)

the new `.c` file looks like:

```
/* Only here */
static char *createStudentID(char *name, int year) {
  ... // Code genereting the id
  return studentId;
}

/* This is declared in the interface (student.h)*/
Student createNewStudent(char *name, int year) {
  assert(name!=NULL && year>=0);
  char *id = createStudentID(name, year);
  Student result;
  strcpy(result.name, name);
  strcpy(result.studentid, id);
  result.credits = 0.0;
  return result;
}
```