



# DVA104 VT19

Stefan Bygde



# ADT: Abstrakt Datatyp

- En abstrakt datatyp är inte ett begrepp i språket C, utan ett generellt programmeringskoncept.
- Vi kommer alltså inte att lära oss något nytt om just språket C här.
- En abstrakt datatyp är en datatyp designad med grund i *hur den används*, inte hur den är implementerad/kodad.
- Avsikten med en abstrakt datatyp är att *abstrahera bort detaljer*
  - Ett viktigt mål i programmering är att **slippa tänka så mycket som möjligt**
- Vi visar det enklast med ett exempel...

# ADT: Abstract Datatyp

Ett bra exempel på en abstrakt datatyp är typen FILE i C. FILE är egentligen en struct, men är det någon som vet hur denna struct ser ut eller hur funktionerna är implementerade?

Varför inte? För att vi inte *behöver* veta det! Allt vi behöver veta är hur man *använder* en fil. Dvs, vilka funktioner vi kan anropa när vi har en filpekare.

Vi vet t.ex att vi kan anropa funktionerna: `fopen()`, `fclose()`, `fwrite()`, `fprintf()`, `fscanf()` osv. Alla dessa funktioner tar filpekare som argument och "gör någonting" med den.

# ADT: Abstract Datatyp

Vi vet t.ex att vi kan anropa funktionerna: `fopen()`, `fclose()`, `fwrite()`, `fprintf()`, `fscanf()` osv. Alla dessa funktioner tar filpekare som argument och "gör någonting" med den.

Vi säger att funktionerna utgör gränssnittet eller *interfacet* till filer. Vi ser också att alla dessa funktioner har prefixet *f*. Så vi kan se att de hör ihop.

Detta är syftet med en abstrakt datatyp: att "gömma" innehållet och implementationen och fokusera enbart på hur man *använder* datastrukturen. Via funktioner som tar pekare till en datatyp och modifierar den internt.

# Att designa en ADT

- När vi designar en ADT så utgår vi alltså enbart ifrån hur vi ska använda datatypen. Alltså **vad** datatypen ska göra.
- Implementation, dvs **hur** det ska implementeras är ej relevant i en ADT.
- Detta görs genom att specificera ett antal *operationer* på vår datatyp. Operationerna kommer att motsvara C-funktioner.

# Att designa en ADT

- Låt oss som exempel tänka oss att vi ska implementera ett LADOK-liknande system som håller reda på information om studenter och deras kurser.
- Vi vill nu konstruera en ADT: Student. Vi vill alltså, utan att veta hur den här datatypen kommer att implementeras, specificera vilka *operationer* vi vill kunna göra med en student.

# Ett nytt (abstrakt) förhållningssätt till datatyper

- Några exempel på operationer vi kan tänka oss att vilja göra i ett LADOK-program.
- Exempel på operationer:
  - Vi vill kunna skapa en ny student med namn och student-id
  - Vi vill kunna skriva ut en students namn på skärmen
  - Vi vill kunna lägga till högskolepoäng till en student
    - Vi behöver t.ex aldrig *minska* högskolepoäng till en student (eller hur!)
  - Fler exempel?
- Notera här hur vi pratar om dessa saker (nästan) helt utan att diskutera vilken information vi ska lagra om studenter. Vi utgår enbart ifrån vad vi vill kunna göra.

# Ett nytt (abstrakt) förhållningssätt till datatyper

- Vi vill ge varje operation ett bra namn som tydligt beskriver vad den gör.
- Exempel på operationer:
  - Vi vill kunna skapa en ny student med namn och student-id (`createNewStudent`)
  - Vi vill kunna skriva ut en students namn på skärmen (`printStudentName`)
  - Vi vill kunna lägga till högskolepoäng till en student (`addStudentCredits`)

Var och en av dessa tre operationer kommer att bli C-funktioner.

Notera att vi har med ordet “student” i alla tre operationer för att visa att funktionerna hör ihop (precis som prefixet *f* i filhanteringsfunktionerna).



# Abstrakta datatyper, Interface

- Varje sak vi vill göra kallas för en operation på en datatyp. I C så implementeras operationer med hjälp av funktioner.

Vårt interface består av tre operationer (funktioner):

1. Skapa en ny student: `createNewStudent()`
2. Skriva ut namnet på en student: `printStudentName()`
3. Lägga till ett antal högskolepoäng till en student: `addStudentCredits()`

# Interface forts

- Nästa steg är att se vilken information dessa funktioner behöver *utan* att förutsätta något om datatypen student.
1. Skapa ny student.
    - a. Vad ska funktionen göra?
    - b. Vilken information behöver den (argument)?
    - c. Vad ska funktionen returnera?

# Interface forts

- Nästa steg är att se vilken information dessa funktioner behöver *utan* att förutsätta något om datatypen student.
1. Skapa ny student.
    - a. Funktionen ska returnera en ny variabel av typen Student
    - b. För att skapa en student behöver vi veta dess namn och student-id

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

Notera att vi fortfarande inte gör något antagande om datatypen Student och vi bryr oss heller inte om hur funktionskroppen för `createNewStudent()` ser ut.

# Interface forts

## 2. Skriva ut namnet på en student

- a. Vad ska funktionen göra?
- b. Vilken information behöver den (argument)?
- c. Vad ska funktionen returnera?

# Interface forts

## 2. Skriva ut namnet på en student

- Behöver en student (den vi ska skriva ut) som argument
- Behöver inte returnera något (ska bara skriva ut något på skärmen)

```
void printStudentName(Student studentToPrint);
```

Tänk på att vi designar på ett sådant sätt vi inte vet alls vad "Student" är för typ. Vi skickar därför inte en pekare till en student. En pekare indikerar att vi vill ändra på argumentet (vilket den här funktionen inte ska), eller att datatypen tar upp mycket plats (vilket vi inte vet!).

När vi designar en abstrakt datatyp ska vi bara skicka en pekare om vi vill att operationen ska ändra på något.

# Interface forts

## 3. Lägga till ett antal högskolepoäng till en student

- Denna funktion ska alltså *ändra* på en student
  - Det vanliga att göra är då att skicka in en *pekare* till variabeln man vill ändra på.
- 
- Vad ska funktionen göra?
  - Vilken information behöver den (argument)?
  - Vad ska funktionen returnera?

# Interface forts

## 3. Lägga till ett antal högskolepoäng till en student

- Funktionen behöver *pekare* till en student, eftersom den ska *ändra* på studenten.
- Funktionen behöver också veta hur många högskolepoäng som studenten ska få.

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

# ADTn Student

Vi har nu ett interface till vår abstrakta datatyp student. Det ser ut som följer:

```
Student createNewStudent(char* name, char* studentid);  
void printStudentName(Student student);  
void addStudentCredits(Student* student, float credits);
```

Vi har alltså designat vad vi kan göra med en student utan att ha brytt oss om implementationen (dvs den underliggande koden), eller ens vad typen "Student" faktiskt är.



# Interface i C

- När vi har ett interface kan vi med fördel lägga det i en h-fil:

student.h

```
#ifndef STUDENT_H  
#define STUDENT_H
```

```
/* Typen Student är ännu odefinierad, vi kommer dit senare */
```

```
Student createNewStudent(char* name, char* studentid);  
void printStudentName(Student student);  
void addStudentCredits(Student* student, float credits);
```

```
#endif
```

# Poängen med abstrakta datatyper

Vi kan nu anta att det finns en fil `student.c` som innehåller funktionsdefinitionerna. Vi behöver dock inte se den filen för att kunna använda studenter!

```
#include "student.h"
int main(void)
{
    Student studentA, studentB; // Skapa två studenter
    studentA = createNewStudent("Alice", "awd16001");
    studentB = createNewStudent("Bob", "bbd15004");

    // Bob klarade just av D0An!
    addStudentCredits(&studentB, 7.5); // Bob har nu 7.5 poäng mer än tidigare!

    // Skriv ut namnet på studentB på skärmen
    printStudentName(studentB);
}
```

Notera att vi enkelt kan förstå hur detta program fungerar fast vi inte vet något om varken funktionerna eller typen `Student`! Detta vill man uppnå i programmering! Vi har **abstraherat** bort detaljerna!

# ADTer

- För att studenter ska fungera måste vi naturligtvis både definiera typen `Student` och implementera alla funktioner.
- En abstrakt datatyp består därför av:
  - Ett interface
  - En implementation av interfacet (definition av datatypen och funktionerna)
- I C kan man göra det på följande sätt för en ADT:
  - Interfacet lägger man i `ADT.h` (synligt för den som ska använda datatypen)
  - Implementationen lägger man i `ADT.c` (dolt för den som ska använda datatypen)
- Pga. tekniska skäl behöver man ibland ha delar av typ definitioner i h-filen. T.ex en `typedef`.

# Implementationen

Implementationen läggs med fördel i en C-fil som innehåller:

- Eventuella struct-definitioner
- Implementationen (definitionen) av funktionerna i interfacet.

Notera att normalt är det inte den som implementerar funktionerna som använder dem. I denna kurs kommer ni dock att agera båda sidor.

# Implementationen (Student)

För att kunna skriva funktionerna behöver vi verkligen veta hur den konkreta datatypen Student ser ut.

Guidelines för den konkreta typen:

- Den bör vara så enkel som möjligt.
- Den bör bara innehålla information om exakt det som behövs för att implementera funktionerna.

# Implementation (Student)

En design som utgår från operationerna:

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

- Vad behöver Student för denna operation?

```
void printStudentName(Student student);
```

- Vad behöver Student för denna operation?

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

- Vad behöver Student för denna operation?

# Implementation (Student)

En design som utgår från operationerna:

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

- Student behöver ett namn (sträng) och ett studentid (sträng)

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

- Ingen övrig information

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

- Student behöver credits (float)

# Implementation (Student)

Slutresultatet blir:

```
struct student
{
    char name[20];
    char studentid[9];
    float credits;
};
```

Notera att informationen om structens medlemmar **ej behövs** för den som ska använda ADTn!

För att andra filer ska få tillgång till den behöver den dock ändå ligga i h-filen.



# Den nya h-filen:

student.h

```
#ifndef STUDENT_H
#define STUDENT_H

struct student /* Hör egentligen inte till interfacet */
{
    char name[20];
    char studentid[9];
    float credits;
};
typedef struct student Student;

/* Interface */
Student createNewStudent(char* name, char* studentid);
void printStudent(Student student);
void addCredits(Student* student, float credits);
#endif
```

# Pre- och Postconditions

- I programmering är det viktigt att man vet exakt vad sina funktioner ska göra. Därför är det viktigt att man definierar detta **innan** man ens börjar skriva dem!
- Detta kan göras med hjälp av såkallade pre- och postconditions
  - Pre-condition: vad **måste vara sant** när funktionen anropas för att det ska bli rätt?
  - Post-condition: vad **är** sant när funktionen har anropats.
- Visas enklast mha exempel...

# Pre- och postconditions

Student createNewStudent(char\* name, char\* studentid);

- Preconditions (vad måste vara sant innan):
  - ?
- Postconditions (vad är sant efter):
  - ?

# Pre- och postconditions

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

- Preconditions (vad måste vara sant innan):
  - `name` måste peka till en sträng (dvs, inte vara NULL)
  - `studentid` måste peka till en sträng (dvs, inte vara NULL)
- Postconditions (vad är sant efter):
  - Inget (funktionen returnerar ett värde, men inget är förändrat)

# Pre- och postconditions

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

- Preconditions (vad måste vara sant innan):
  - ?
- Postconditions (vad är sant efter):
  - ?

# Pre- och postconditions

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

- Preconditions (vad måste vara sant innan):
  - Inget
- Postconditions (vad är sant efter):
  - Namnet på *student* står på skärmen

# Pre- och postconditions

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

- Preconditions (vad måste vara sant innan):
  - **credits** måste vara positivt
  - **student** får inte vara NULL
- Postconditions (vad är sant efter):
  - **\*student** har nu **credits** fler högskolepoäng än innan anropet

# Implementation

- Vi har definierat den konkreta datatypen `Student` som en struct
- Vi har definierat alla våra operatorer samt pre- och postconditions för dem
- Vi är klara att implementera våra funktioner.

Alternativt:

- Skicka vår definition till en annan programmerare som ska kunna implementera dem efter vår design.
- Igen: det viktiga att inse att typen redan är “klar att användas” oavsett hur implementationen kommer att se ut.



# Implementation

## student.c

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

```
Student createNewStudent(char* name, char* studentid)  
{  
    Student result;  
    strcpy(result.name, name);  
    strcpy(result.studentid, studentid);  
    Result.credits = 0.0;  
    return result;  
}
```

Här är ett exempel på hur en implementation av operationen `createNewStudent()` kan se ut.

# Pre- och Postconditions blir konkreta

- Det finns en trevlig funktion i C som gör att vi kan skriva in våra pre- och postconditions i koden.

```
assert(int condition);
```

- Denna funktion avbryter programmet om `condition` evalueras till 0 (dvs falskt).
- Denna funktion är en slags debugfunktion. Den “stängs av” i slutversionen av programmet.
- För att använda funktionen behöver man inkludera `<assert.h>`

# Pre- och postconditions

Student createNewStudent(char\* name, char\* studentid);

- Preconditions (vad måste vara sant innan):
  - name måste peka till en sträng (dvs, inte vara NULL)
  - studentid måste peka till en sträng (dvs, inte vara NULL)

```
Student createNewStudent(char* name, char* studentid)
{
    assert(name != NULL); /* Precondition: name måste peka till en sträng (dvs, inte vara NULL) */
    assert(studentid != NULL); /* Precondition: studentid måste peka till en sträng (dvs, inte vara NULL) */
    Student result;
    strcpy(result.name, name);
    strcpy(result.studentid, studentid);
    result.credits = 0.0;
    /* Här skulle vi lägga post-conditions om vi hade några */
    return result;
}
```

# Assert

- Asserts ser till att vi använder vår ADT som det är tänkt
- Om en assert aktiveras så betyder det att vi inte har respekterat våra pre- och postconditions. Alltså fungerar det inte som det är tänkt.
- Asserts kan även användas för att kontrollera värden på variabler när som helst för att säkerställa att det fungerar som man tänkt (våldigt användbart!)
- När programmet fungerar och inga asserts aktiveras kan vi stänga av asserts genom: `#define NDEBUG`

# Implementation forts:

- Inga pre- och postconditions (iaf inte som går att kontrollera)

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

# Implementation forts

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

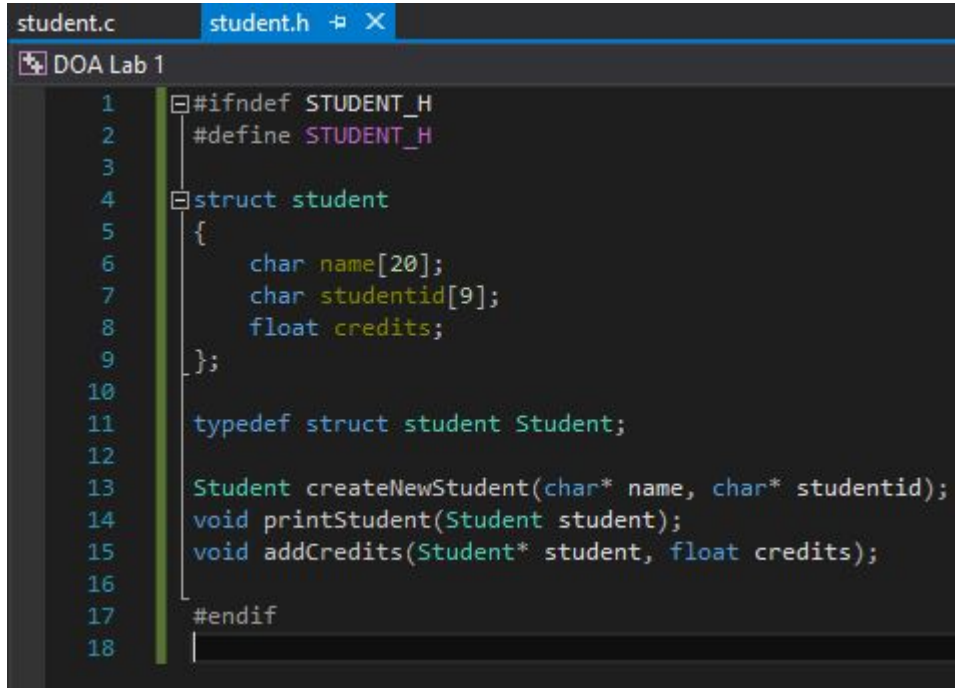
- Preconditions (vad måste vara sant innan):
  - `credits` måste vara positivt
  - `student` får inte vara NULL
- Postconditions (vad är sant efter):
  - `*student` har nu `credits` fler högskolepoäng än innan anropet

```
void addCredits(Student* student, float credits)
{
    assert(credits > 0); /* precondition: credits måste vara positivt */
    assert(student != NULL); /* precondition: student får inte vara NULL */

    float creditsBefore = student->credits; /* Just nu bara för kontroll */

    student->credits = student->credits + credits;
    /* Post condition: *student har nu credits fler högskolepoäng än tidigare */
    assert(student->credits == creditsBefore + credits);
}
```

# Slutresultatet



```
student.c  student.h  X
DOA Lab 1
1  #ifndef STUDENT_H
2  #define STUDENT_H
3
4  struct student
5  {
6      char name[20];
7      char studentid[9];
8      float credits;
9  };
10
11  typedef struct student Student;
12
13  Student createNewStudent(char* name, char* studentid);
14  void printStudent(Student student);
15  void addCredits(Student* student, float credits);
16
17  #endif
18
```

# Slutresultatet

```
student.c  X student.h
DOA Lab 1
1  #define _CRT_SECURE_NO_WARNINGS
2  #include "student.h"
3  #include <assert.h>
4  #include <string.h>
5  #include <stdio.h>
6
7  Student createNewStudent(char* name, char* studentid)
8  {
9      assert(name != NULL);
10     assert(studentid != NULL);
11
12     Student result;
13     strcpy(result.name, name);
14     strcpy(result.studentid, studentid);
15     return result;
16 }
17
18 void printStudent(Student student)
19 {
20     printf("%s", student.name);
21 }
22
23 void addCredits(Student* student, float credits)
24 {
25     assert(credits > 0);
26     assert(student != NULL);
27
28     float prevCredits = student->credits;
29     student->credits = student->credits + credits;
30     assert(student->credits == prevCredits + credits);
31 }
```



# ADTer sammanfattning

- En abstrakt datatyp består av två komponenter (som kan göras av helt olika personer/programmerare)
  - Ett interface (publikt, främst i h-filen)
  - En implementation (dolt, främst i c-filen)
- Associerade koncept:
  - Pre- och postconditions: hör till operationerna
  - Asserts: kan användas för att säkerställa pre- och postconditions
  - Abstraktion: implementationen är oberoende av interfacet