**Introduction to Abstract Data Types**

Abstract Data Type (ADT)

- A collection of data and operations that
  - have precisely described behaviour (we know what they do)
  - but no precise implementation (we don't know how they do it)

- ADTBookStore (from the first lecture)
  - insert(title)
  - remove(title)
  - find(title)

- Do we know any such type?

**Native types**

- How is int implemented?
- What does int a = -2; store in memory?
  - 10...10 (sign-magnitude)
  - 1111101 (1-complement)
  - 1111110 (2-complement)
  - bit order? (little vs big endian)
  - size? (16, 32, 64 bits)
  - at least $3 \times 2 \times 3 = 18$ possibilities! The choice depends on the CPU.
- How is a++ implemented?

- Even simple native types and operations are in reality abstract
- We know what they do but not how
- int a = 1 stores some representation of 1 in a
- a++ stores the representation of $a + 1$ in a
  - where $a$ is the number represented in a
- printf("%d", a) prints the number $a$ represented in a
Why?

1. We can write programs without thinking (or even knowing) about how these operations are implemented
   - use complicated algorithms easily
2. We can change the implementation of int (eg change the CPU) without changing the code
   - easy maintenance

It would be impossible to write complex programs without these features!

Writing our own ADTs

- ADTFoo will be represented by the module ADTFoo.h
  - Declare a list of functions, constants, typedefs, etc
  - Describe what the module does, with documentation!
- To use ADTFoo
  - #include "ADTFoo.h"
  - Call its methods, eg foo_create()
  - Link with foo.o (or some library containing it)
- To implement ADTFoo
  - Create foo.c, implementing all functions
  - The implementation should match the advertised behaviour

Containers

- The ADTs we learn in this class are containers
  - They allow to insert data (stored in the container)
  - Then retrieve it in different ways
  - And remove it
- Store values of any type: void*
- They have similar interfaces
  - Differ in the way data is inserted/removed/retrieved

ADT Overview

<table>
<thead>
<tr>
<th>ADT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADTVector</td>
<td>An abstract growable “array”</td>
</tr>
<tr>
<td>ADTList</td>
<td>Insert at any position, no “random access”</td>
</tr>
<tr>
<td>ADTQueue</td>
<td>First-in, First-out</td>
</tr>
<tr>
<td>ADTStack</td>
<td>Last-in, First-out</td>
</tr>
<tr>
<td>ADTPriorityQueue</td>
<td>Fast-access of the maximum element</td>
</tr>
<tr>
<td>ADTMap</td>
<td>Associate key =&gt; value (array with any type of index)</td>
</tr>
<tr>
<td>ADTSet</td>
<td>Ordered collection of unique items</td>
</tr>
</tbody>
</table>
Naming

- We use different names for ADTs and Data Structures
  - eg. ADTVector implemented by a Dynamic Array
- Loosely following the naming of the C++ standard library
- Be careful: each ADT/DS is known under many different names
  - also: the same name is often used for ADTs and DSs
- Remember the substance, not just the names!

A typical container ADTFoo

```c
#include "ADTFoo.h"
int main() {
    Foo foo = foo_create();
    // ... foo
    // Αφαίρεση στοιχείου 
    foo_remove(foo, ...);
    // Εκκαθάριση μνήμης 
    foo_destroy(foo);
}
```

A typical use of ADTFoo

```c
// Δημιουργεί και επιστρέφει ένα νέο foo 
Foo foo_create();
// Επιστρέφει τον αριθμό στοιχείων που περιέχει το foo 
int foo_size(Foo foo);
// Προσθέτει την τιμή value στο foo 
void foo_insert(Foo foo, Pointer value, ...);
// Αφαίρεται και επιστρέφει μια τιμή από το foo 
Pointer foo_remove(Foo foo, ...);
// Βρίσκεται και επιστρέφει ένα στοιχείο από το foo 
Pointer foo_find(Foo foo, ...);
// Ελευθερώνει όλη τη μνήμη που δεσμεύει το foo 
void foo_destroy(Foo foo);
```
Many containers allow iterating

Using the concept of `node`.

```c
Foo foo = foo_create();
// ...insert...

for (FooNode node = foo_first(foo); node != FOO_EOF; node = foo_next(foo, node)) {
    int* value = foo_node_value(foo, node); // η τιμή του συγκεκριμέν
    printf("value: %d\n", *value);
}
```