Operating Systems

Complements of C Programming

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Object Orientation

Object oriented programming makes it easy to handle large projects by

- encapsulation
- polimorphism (late binding)
- inheritance

In fact most OS are written in C/Assembly and are LARGE projects

C is not an object oriented language for most, so in theory OOP is not used by most OSes => FALSE

OOP is a programming style. A language that supports explicit object orientation, just makes it easier from a syntactic point of view to implement the mechanisms above

A C++ program can be compiled in machine code just as well as a C program. Is machine code object oriented?

Implementing OO Concepts in C

Object Oriented Programming in C is an excellent book on the topic

- https://www.cs.rit.edu/~ats/books/ooc.pdf

In this lesson we will learn how to implement

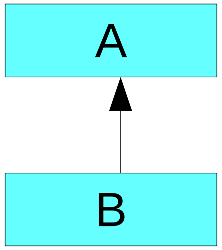
- inheritance and
- polimorphism

in C

Inheritance

An object of class B inherited from A has all the attributes of A.

An object of class B is also an object of class A



In C++ (java like style)

```
class A {
     int my_int;
     float my_float;
};

class B : public A {
     int my_other_int;
};

void printMyInt(A* a) {
     printf("my_int=%d\n", a->my_int);
};

int main(){
  B b;
  b.my_int=3;
  printMyInt(&b); // prints 3

  A a;
  a.my_int=4;
  printMyInt(&a); // prints 4

  // assigns to all members of a the
  // corresponding value in b
  // possible since a is base of b
  a=b;

  printMyInt(&a); // prints 3
}
```

Inheritance

Simple inheritance can be implemented in C by setting as first variable in the derived struct an instance of the base class

A pointer to the derived class will be also a pointer to the 1st member variable in the struct which is of type A

In C

```
typedef struct{
     int my_int;
     float my_float;
} A;

typedef struct B {
     A parent;          // the 1st member of B
                       // is the base class
     int my_other_int;
};

void printMyInt(A* a)  {
     printf("my_int=%d\n", a->my_int);
};

int main(){
  B b;
  b.my_int=3;
  printMyInt(&b); // prints 3

  A a;
  a.my_int=4;
  printMyInt(&a); // prints 4

  a=b.parent;

  printMyInt(&a); // prints 3
}
```

A generic list in C

- We want to implement once a double linked list whose elements can be extended to store arbutrary (but homogeneous) types
- Let’s start with an interface for our generic C list library:
  - A generic list is characterized by a struct (ListHead) that contains
    - pointers to the first and the last element
    - a size (int)
  - The elements of the list
  - **It has no "info" field**

```
#pragma once

// generic list item
typedef struct ListItem {
  struct ListItem* prev;
  struct ListItem* next;
} ListItem;

// head of list
typedef struct ListHead {
  ListItem* first;
  ListItem* last;
  int size;
} ListHead;

// initializes the head
void List_init(ListHead* head);

// retrieves an element in the list
// returns NULL if item not in list
ListItem* List_find(ListHead* head,
                    ListItem* item);

// inserts an element after previous in the list
// returns NULL on failure
ListItem* List_insert(ListHead* head,
                      ListItem* previous,
                      ListItem* item);

// detaches (without deleting) an element to the
// list
ListItem* List_detach(ListHead* head,
                      ListItem* item);
```

A generic list in C

```c
#include "linked_list.h"
#include <assert.h>

void List_init(ListHead* head) {
  head->first=0;
  head->last=0;
  head->size=0;
}

ListItem* List_find(ListHead* head,
                    ListItem* item) {
  // linear scanning of list
  ListItem* aux=head->first;
  while(aux){
    if (aux==item)
      return item;
    aux=aux->next;
  }
  return 0;
}
```

```c
ListItem* List_detach(ListHead* head,
                      ListItem* item) {

#ifdef _LIST_DEBUG_
  // we check that the element is in the list
  ListItem* instance=List_find(head, item);
  assert(instance);
#endif

  ListItem* prev=item->prev;
  ListItem* next=item->next;
  if (prev){
    prev->next=next;
  }
  if(next){
    next->prev=prev;
  }
  if (item==head->first)
    head->first=next;
  if (item==head->last)
    head->last=prev;
  head->size--;
  item->next=item->prev=0;
  return item;
}
```

A generic list in C

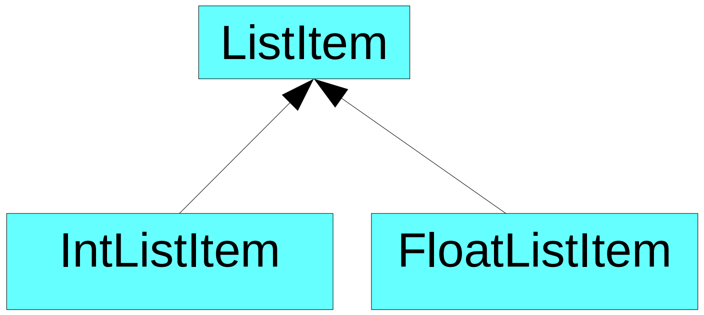
Note that in this example the memory management is delegated to the user, who needs to know what is stored in the class

```
ListItem* List_insert(ListHead* head,
                      ListItem* prev,
                      ListItem* item) {
  if (item->next || item->prev)
    return 0;

#ifdef _LIST_DEBUG_
  // we check that the element is not in the list
  ListItem* instance=List_find(head, item);
  assert(!instance);

  // we check that the previous is inthe list

  if (prev) {
    ListItem* prev_instance=List_find(head, prev);
    assert(prev_instance);
  }
  // we check that the previous is inthe list
#endif

  ListItem* next= prev ? prev->next : head->first;
  if (prev) {
    item->prev=prev;
    prev->next=item;
  }
  if (next) {
    item->next=next;
    next->prev=item;
  }
  if (!prev)head->first=item;
  if(!next) head->last=item;
  ++head->size;
  return item;
}
```

Specializing the generic

Our list, alone is of little use

We need to specialize it to store any quantity of our interest in the nodes



```
// derived class
typedef struct {
  ListItem list;
  int info;
} IntListItem;

//helper function to print a list of ints
void IntList_print(ListHead* head){
  ListItem* aux=head->first;
  printf("[");
  while(aux){
    IntListItem* element = (IntListItem*) aux;
    printf("%d ", element->info);
    aux=aux->next;
  }
  printf("]\n");
}
```

Specializing the generic

A simple main

```
#define MAX_NUM_ITEMS 64

int main(int argc, char** argv) {
  // we populate the list
  ListHead head;
  List_init(&head);
  for (int i=0; i<MAX_NUM_ITEMS; ++i){
    IntListItem* new_element= (IntListItem*)
      malloc(sizeof(IntListItem));
    if (! new_element) {
      printf("out of memory\n");
      break;
    }
    new_element->list.prev=0;
    new_element->list.next=0;
    new_element->info=i;
    ListItem* result=
      List_insert(&head,
                  head.last,
                  (ListItem*) new_element);
    assert(result);
  }
  IntList_print(&head);

  printf("removing odd elements");
  ListItem* aux=head.first;
  int k=0;
  while(aux){
    ListItem* item=aux; aux=aux->next;
    if (k%2){
      List_detach(&head, item);
      free(item);
    }
    ++k;
  }
  IntList_print(&head);

  printf("removing from the head, half of the list");
  int size=head.size; k=0;
  while(head.first && k<size/2){
    ListItem* item=List_detach(&head, head.first);
    free(item); ++k;
  }
  IntList_print(&head);

  printf("removing from the tail the rest of the list,
          until it has 1 element");
  while(head.first && head.size>1){
    ListItem* item=List_detach(&head, head.last);
    free(item);
  }
  IntList_print(&head);

  printf("removing last element");
  ListItem* item=List_detach(&head, head.last);
  assert(item);
  free((item);
  IntList_print(&head);
}
```

Function Pointers

A function pointer is a type representing the start address of a function.

Function pointers have a type that depends on the function prototype:

A function pointer to

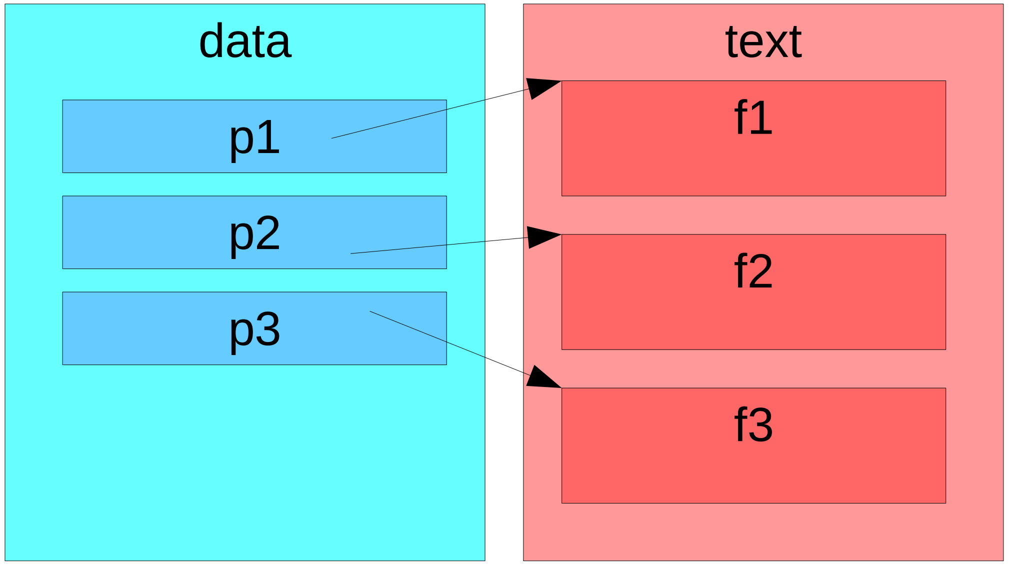
`int f1(float)`

- is different from a function pointer to

  `int f2(char)`

- is the same as a function pointer to

  `int f3(float)`



```
typedef int(*IntFloatPtrFn)(float);
typedef int(*IntCharPtrFn)(char);

int f1(float f) {...}
int f2(char c) {...}
int f3(float f) {...}

IntFloatFnPtr p1=f1;
IntCharFnPtr p2=f2;
IntFloatFnPtr p3=f3;

// the following assignent works, same type
p1=p3;

// the following doesn't
p2=p1;
```

Function Pointers

Declaration of a function pointer type that matches a function prototype

```
typedef <return_type>(*<typename>)(args)
```

Example

```
int (*IntCharFnPtr)(char)
```

Assigninment

- From a function: the function name without (…) represents the address of a function.

```
int f(char) {...}
IntCharFnPtr p1=f;
```

- From another pointer of the same type

```
IntCharFnPtr p2=p1;
```

Invoking the function pointed by a function pointer is done by dereferencing the function pointer and calling it as a normal function

```
int retval=(*p1)(‘c’);  // equivalent to retval=f1(‘c’);
```

Function Pointers in OS

Function pointers are used extensively

- Hardware
  - Interrupt vector
  - Traps

- OS
  - System call table
  - Device drivers
  - Loading/Binding libraries

- Implementation of Object Oriented Programs
  - Late binding (Polimorphism)

Function Pointers Example

Simple program that uses a single function pointer whose value changes each time it is invoked

What does it do?

```
//declare a function pointer void f(void)
typedef void (*ActionPtrFn)(void);

// variable declaration
ActionPtrFn action;

// forward decl
void actionFn0();
void actionFn1();
void actionFn2();

void actionFn0(void) {
  printf("handing  0\n");
  action=actionFn1;
}

void actionFn1(void) {
  printf("handing  1\n");
  action=actionFn2;
}

void actionFn2(void) {
  printf("handing  2\n");
  action=actionFn0;
}

int main(int argc, char** argv) {
  action=actionFn1;
  while(1) {
    int value;
    getchar();
    (*action)();
  }
}
```

Function Pointers Example

Vectors or other data structures of function pointers are rather common in the above discussed scenarios

This example installs a vector of function pointers and asks the user which one to call from the terminal

```c
#define MAX_ACTIONS 10
int run=1;
//declare a function pointer type void f(int)
typedef void (*ActionPtrFn)(int);

void actionFn0(int ev) {
  printf("Action 0, handling event %d\n", ev);
}
void actionFn1(int ev) {
  printf("Action 1, handling event %d\n", ev);
}
void actionFnDefault(int ev) {
  printf("default action, handling event %d\n", ev);
}
void actionFnQuit(int ev) {
  printf("quitting %d\n", ev); run = 0;
}
// array of pointers to functions
ActionPtrFn actions[MAX_ACTIONS];
int main(int argc, char** argv) {
  for (int i=0; i<MAX_ACTIONS; ++i)
    actions[i]=actionFnDefault;
  actions[0]=actionFn0;
  actions[1]=actionFn1;
  actions[8]=actionFn0;
  actions[9]=actionFnQuit;
  while(run) {
    int v;
    scanf("%d",&v);
    if (v<0||v>=MAX_ACTIONS) {
      printf("invalid index %d",v); continue;
    }
    (*actions[v])(v);
  }
}
```

Polimorphism (late binding)

In programming languages with an advanced support for object orientation, polimorphism is extensively used

- A base class offers a set of functionalities whose implementation can be overridden in the derived classes
- In Java all non static methods can be overridden
- In C++ only those that are declared virtual (for efficiency reasons) can be overridden

Late Binding

A pointer to a base class encodes the knowledge of the real type it represents.

Calling a virtual method results in calling the implementation of the "true" pointed variable

A class with virtual methods is called a "virtual" class in C++.

To implement this mechanism an instance of an object should "know" its type, and what function to call

In C++ (java like style)

```
class A {
public:
     virtual void print(){
          printf("A\n");
     }
};

class B : public A {
public:
     virtual void print()override {
          printf("B\n");
     }
};


int main(){
  A a; // on stack
  B b; // on stack

  A* a_ptr= &a;
  a_ptr->print(); // calls print of A

  a_ptr=&b;
  a_ptr->print(); // calls print of B
}
```

Late Binding Internals

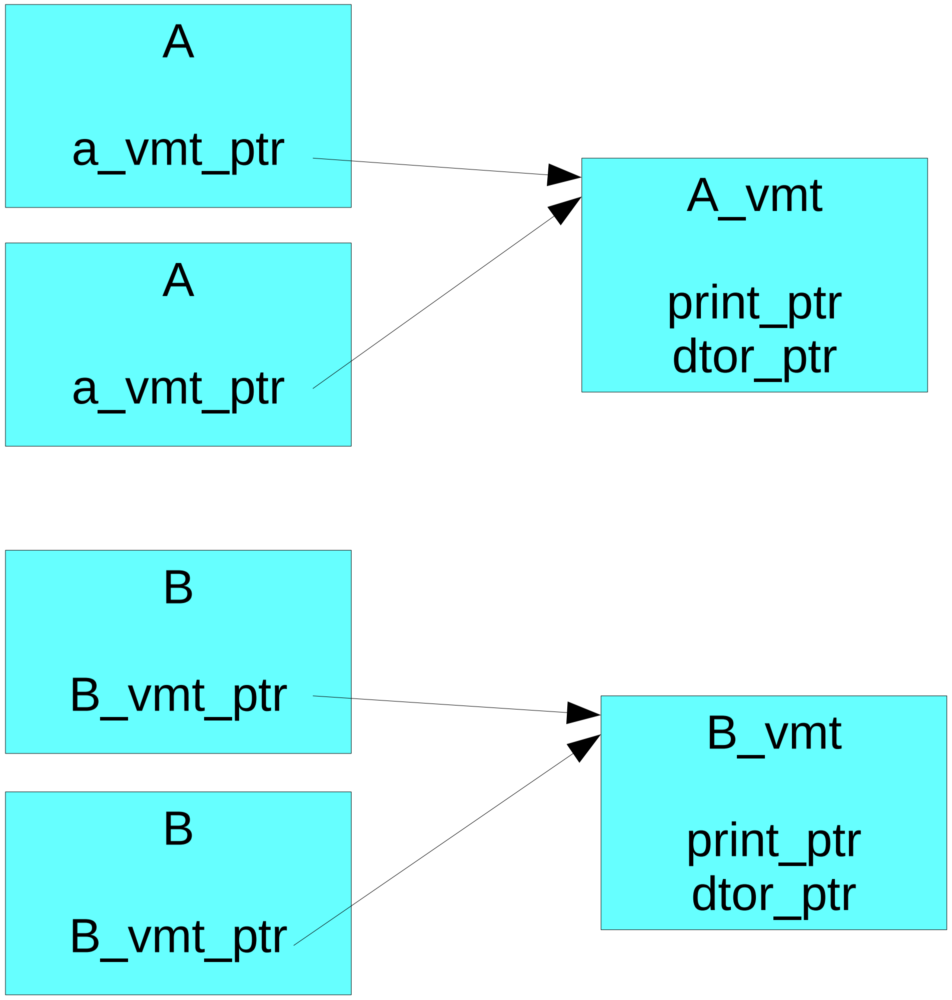
Late binding is implemented by adding to each instance of a virtual class a pointer to an instance of a structure called virtual method table.

All objects of the same class point to the same instance of the virtual method table (VMT).

The VMT contains the pointers to the overridden functions.

A call to a virtual method results in crawling to the VMT and fetching the pointer to the right method.

The destructor (dtor) is a special virtual function called when the object is destroyed, and is responsible of a proper cleanup. A default implementation is added by default by C++ compilers.



Late Binding in C

```
#include <stdio.h>

struct A; // forward declaration of A;

//pointer to print fn;
typedef void (*A_PrintFnPtr)(struct A*);

// A print function
void A_print_impl(struct A* a){printf("A\n");}
// B print function
void B_print_impl(struct A* a){printf("B\n");}

// A VMT (same type as B VMT since no methods
// are added in B).
// If B adds methods B_VMT inherits from A_VMT
// overriden methods overwrite the corresponding
// pointer

typedef struct {
  A_PrintFnPtr print_fn;
} A_ops;

// A class
typedef struct A {
  A_ops* ops;
} A;

// VMT is inherited
typedef struct B {
  A parent;
} B;
```

```
// global variables representing vmt
// the print_fn field is initialized on creation
A_ops a_ops = {.print_fn=A_print_impl};
A_ops b_ops = {.print_fn=B_print_impl};

// wrapper, crawls in VMT and calls right function
void A_print(struct A* a_ptr) {
  (*a_ptr->ops->print_fn)(a_ptr);
}

int main(){
  A a1={.ops=&a_ops}; //declare and set ptr to vmt
  A a2={.ops=&a_ops};
  B b1={.parent.ops=&b_ops};
  B b2={.parent.ops=&b_ops};

  A* a_ptr = &a1;
  A_print(a_ptr);

  a_ptr = (A*) &b1;
  A_print(a_ptr);

  a_ptr = &a2;
  A_print(a_ptr);

  a_ptr = (A*) &b2;
  A_print(a_ptr);

}
```

A Polymorphic List

Using the concepts learned so far, we can implement a polimorphic list, that is a list containing arbitrary objects.

The objects in the list will be instances of virtual classes.

The list head does not need (in this example) to be virtual.

We add an optional virtual destructor for further extensions.

```
// forward declaration shuts up the compiler
struct ListItem;

// definition of a type for a function pointer
// void function(struct ListItem*), used for print
typedef void(*ListItemPrintFn)(struct ListItem*);

// definition of a type for a function pointer
// void function(struct ListItem*), used for dtor
typedef void(*ListItemDestroyFn)(struct ListItem*);

// VMT for the linked list item
typedef struct {
  ListItemDestroyFn dtor_fn;
  ListItemPrintFn   print_fn;
} ListItemOps;

typedef struct ListItem {
  struct ListItem* prev;
  struct ListItem* next;
  ListItemOps* ops; // pointer to VMT
} ListItem;

// exposed functions that call right methods
// by vrawling in the virtual method table
void ListItem_destroy(ListItem* item);
void ListItem_print(ListItem* item);

// installs an instance of VMT in the item
void ListItem_construct(ListItem* item,
                        ListItemOps* ops);

// generic print (calls ListItem print for all)
void List_print(ListHead* head);
```

A Polymorphic List

In the implementation we don't need to provide implementations for the print or the dtor methods

This list is purely virtual (abstract)

How can I implement a print method if i don't know what data my type is storing?

We will delegate the implementation of print to the derived classes of ListItem

```
void ListItem_construct(ListItem* item,
                        ListItemOps* ops) {
  item->prev=item->next=0;
  item->ops=ops;
}

// wrapper for destroy
void ListItem_destroy(ListItem* item) {
  assert(item);
  if (item->ops && item->ops->dtor_fn)
    (*item->ops->dtor_fn)(item);
}

// wrapper for print
void ListItem_print(ListItem* item) {
  assert(item);
  assert(item->ops && item->ops->print_fn);
  (*item->ops->print_fn)(item);
}

// print function
void List_print(ListHead* head) {
  ListItem* item=head->first;
  printf("Printing polimorphic list\n");
  int k=0;
  while(item) {
    printf("l[%d]: ",k);
    ListItem_print(item);
    item=item->next;
    ++k;
  }
}
```

A Polymorphic List

We want to specialize a list item to store a float and an int.

We will then put all these heterogeneous objects in a single list and we will print the list.

The objects do not need to memory management (no mallocs inside a list item), so we ignore the dtor and we focus on the print method.

```
/*IntList*/
typedef struct {
  ListItem list;
  int info;
} IntListItem;

// print method (late binding)
void IntListItem_print(struct ListItem* item){
  printf("[int] %d\n",((IntListItem*)item)->info);
}

// vtable for int list (an INSTANCE)
// this is a global variable
// initialized on construction
ListItemOps int_list_item_ops={
   .dtor_fn=0,
   .print_fn=IntListItem_print
};

/*FloatList*/
typedef struct {
  ListItem list;
  float f;
} FloatListItem;

// print method (late binding)
void FloatListItem_print(struct ListItem* item){
  printf("[float] %f\n",((FloatListItem*)item)->f)
}

// vtable for float list (an INSTANCE)
ListItemOps float_list_item_ops={
   .dtor_fn=0,
   .print_fn=FloatListItem_print
};
```

A Polymorphic List

The main just populates the list and prints the values with a single call to List_print(...).

```
int main(int argc, char** argv) {
  // we populate the list,
  ListHead head;
  List_init(&head);
  for (int i=0; i<MAX_NUM_ITEMS; ++i){
    ListItem* item=0;
    // even elements are int, odd are float
    if (i&0x1) {
      FloatListItem* new_element=
          (FloatListItem*)
         malloc(sizeof(FloatListItem));
      ListItem_construct
       ((ListItem*) new_element,
        &float_list_item_ops);
      new_element->f=i/2.0;
      item=(ListItem*) new_element;
    } else {
      IntListItem* new_element=
          (IntListItem*)malloc(sizeof(IntListItem))
      ListItem_construct
      ((ListItem*)new_element, &int_list_item_ops)
      new_element->info=i;
      item=(ListItem*) new_element;
    }
    List_insert(&head, head.last, item);
  }
  List_print(&head);
};
```

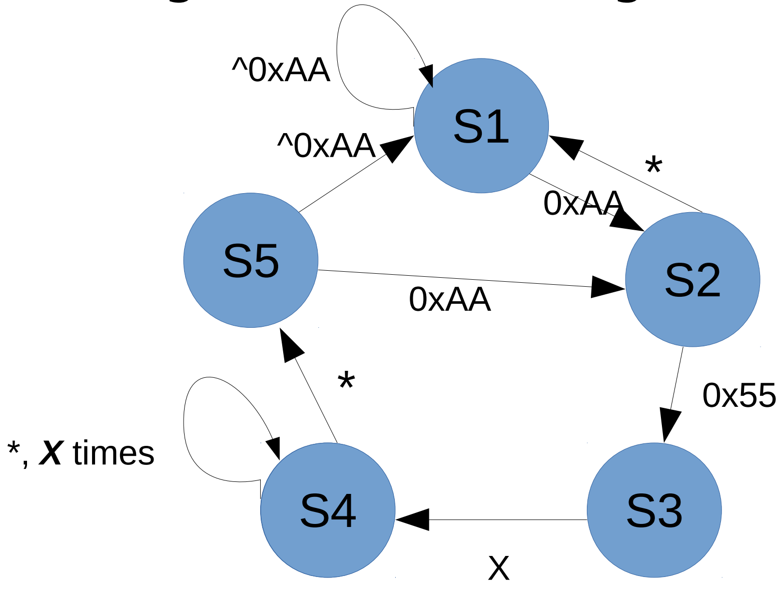
Exercise 1 - inheritance

- Extend the list class (without polimorphism) so that its elements represent lists of floats
- Use this representation to store a matrix of floats
- Write a function that computes the sum of each row and populates a new list of floats

Start from the linked_list program provided

Exercise 2 - function pointers

- Implement a program that realizes the following automata by using function pointers
- The automaton is fed one character (byte) at a time and toggles state according to the following table



*: any character, ignore value
X: any character, ***X***: value of character

- Hints
  - hint there is a function per state of the automaton
  - all states will have prototype like **`void f(char)`**, where char is the input.
  - the current state corresponds to the state of the automaton, and it is stored in a global variable

Exercise 3 - Polymorphism

- Implement a polymorphic list item that can store polymorphic lists
- Load the structure with a matrix of float/int values