Computer Programming

Modular compilation. Abstract data types

Marius Minea

marius@cs.upt.ro

18 December 2017

How to structure complex programs?

Large programs are written by many users, in *many source files*.
can be *compiled separately* ("translation units"),
then *linked* into a single executable

Need to:

control sharing of variables and functions:
allow use of functions / variables *defined elsewhere*
allow declarations which are *not shared* (no name conflicts)

ensure functions are *used correctly* (with right parameters)

This is controlled through *scope* and *linkage* of identifiers

Properties of identifiers

Scope of identifiers: where is identifier *visible* ?

block scope: from declaration to end of enclosing }

file scope: if declared outside any block

also: *function prototype* scope (ID in function header)

function scope (`goto` labels: can't jump out)

if redeclared, *outer* scope *hidden* while *inner* scope in effect

Linkage of identifiers: do they refer to the same object ?

external: same in all *translation units* (files) making up program

default for functions and file scope identifiers;

explicit with `extern` declaration

internal: same within one translation unit; if declared `static`

none: each declaration denotes distinct object (for block scope)

Storage duration of objects (variables)

automatic, for variables declared with block scope
lifetime: from block entry to exit; re-initialized every time

static: lifetime is program execution; initialized once

allocated: with `malloc`

thread: for `_Thread_local` objects (since C11)

Declarations and definitions

An identifier can be *declared* multiple times, only *defined once*

A declaration with initializer is a definition.

A file scope declaration with no initializer and no storage class specifier or with `static` is a *tentative definition*
several tentative definitions for same object must match
become definition by end of translation unit

How to use in practice

functions: define in one file, declare in all others
variables: define in one file, declare `extern` in all others

Can put declarations in a *header file*, and include where needed

Typical library structure

`mylibrary.h`: *header file*, has *declarations* made *visible* for *use*:
typedefs, function *declarations* (NOT definitions/bodies), macros,
declarations of global variables (like `errno`), etc.
NO ~~definitions~~ (would duplicate if header included in many .c files)

```
#ifndef _MYLIBRARY_H
#define _MYLIBRARY_H
// any declarations available to use
#endif
```

`mylibrary.c` : *code* / *definitions* for declarations from `.h`
(function/variable definition; struct definition if only pointer in `.h`)
+ all implementation details that should be hidden from user
`#include "mylibrary.h"` (declaration/definition consistency)

`gcc -c mylibrary.c` compiles to *object file*: `mylibrary.o`
contains *object code* for functions, *symbols* for function names

main file has `#include "mylibrary.h"` , uses functions, types, ...
`gcc program.c mylibrary.o` *compiles* and *links* with library

Abstract datatypes

An abstract datatype is a mathematical model for datastructures
defined by the operations applicable to them (*functions*)
and the constraints among them (*axioms*)
without exposing details about the implementation.

ADTs *separate interface from implementation*
the interface provides the *abstraction*
the implementation is *encapsulated* (hidden)

ADTs allow changeable and interchangeable implementations
client program relies only on interface, is not affected

An example: `FILE *`

C provides the `FILE *` type to work with files.

A `FILE *` can only be used with the functions from `stdio.h`:

a value of type `FILE *` can only be obtained from `fopen`

we can't dereference a `FILE *`, not knowing the `FILE` type

the declaration is not accessible, it's not in `stdio.h`

it's some structure, declared only in the source of the library

can't index, no pointer arithmetic, etc., only standard functions

Lists as abstract data types

Def: A *list* is empty, or an element followed by a list.

An ADT list L with elementtype E is usually defined by:

$nil : () \to L$	empty list constructor
	can also be constant rather than function
$isempty : L \to Bool$	is empty ?
$cons : E \times L \to L$	constructor: new list from element and rest
$head : L \to E$	first element
$tail : L \to L$	*list* with all elements after head

and the *axioms*

$$head(cons(e, l)) = e \quad \text{and} \quad tail(cons(e, l)) = l$$

Some languages have lists as *algebraic* data type:
a *sum type* (alternative) between (1) the value for empty list, and
(2) a *product type* of an element and a list (constructor *cons*).

How to declare an ADT with structures

For structure types, encapsulation is enforced if:
  header file only contains *declaration* of *pointer type*

```
typedef struct mytype *mytype_t;
```

  C file for *implementation* contains *structure definition*

```
struct mytype {
 // declare fields here
};
// functions can access structure fields
```

Exported functions only work with *pointer type* `mytype_t`
  ⇒ not knowing structure, user program cannot access fields

The `FILE` datatype also enforces such an encapsulation

Example ADT for integer list

```
#ifndef _INTLIST_H
#define _INTLIST_H

typedef struct ilst *intlist_t;

intlist_t empty(void);
int isempty(intlist_t lst);
int head(intlist_t lst);
intlist_t tail(intlist_t lst);
intlist_t cons(int el, intlist_t tl);

// for freeing memory only: splits first element from tail,
// if elp non-NULL, store value of head there
intlist_t decons(intlist_t lst, int *elp);

#endif
```

Implementing an abstract datatype.

Linked lists and queues

Implementing the list ADT: file intlist.c

```
#include <stdlib.h>   // for NULL and malloc
#include "intlist.h"  // ensures .h and .c consistent

struct ilst {
  intlist_t nxt;
  int el;
};

intlist_t empty(void) { return NULL; }

int isempty(intlist_t lst) { return lst == NULL; }

int head(intlist_t lst) { return lst->el; }

intlist_t tail(intlist_t lst) { return lst->nxt; }
```

Implementing the list ADT (cont'd)

```
intlist_t cons(int el, intlist_t tl)
{
  intlist_t p = malloc(sizeof(struct ilst));
  if (!p) return NULL; // could report some error
  p->el = el;
  p->nxt = tl;
  return p;
}

 // returns tail, assigns *elp with head, deletes cell
intlist_t decons(intlist_t lst, int *elp)
{
  if (elp) *elp = lst->el;
  intlist_t tl = lst->nxt;
  free(lst);  // just first cell, keeps rest
  return tl;
}
```

Hiding / exposing the representation

If header file declares (exposes) only a *pointer* type to the data, implementation is *hidden*

incomplete structure type: `typedef struct ilst *intlist_t`

or a `void *` (but dangerous: no type safety)

Declaration of structure should be hidden in `.c` file
not exposed in `.h` file (which is included by all clients)

```
struct ilst {
  intlist_t nxt;
  int el;
};
```

If library client has this structure, datatype is no longer *abstract*
can use internal representation, change the structure in-place, etc.

Can we do lists of arbitrary types?

C does not have polymorphism or parametric types
⇒ cannot declare, e.g., list of *arbitrary type*

Could do: `typedef int elemtype;` (or even a `#define`)
and have everything else use `elemtype`

But need to *recompile* everything when changing `elemtype`
binary code differs even for assignment/parameter passing
due to varying element size; even more so for addition, etc.

If instead of values we store *pointers* to values,
we can have just one implementation (list of `void *`)
must separately allocate memory for elements
program logic must know element type (info not in the list)

Example: list reversal in-place

To modify the list in-place, we need access to the representation:

```
struct ilst {
  intlist_t nxt;
  int el;
};
```

Two pointers, splitting list:

one to part of list already reversed (initially `NULL`)
one to rest of list to be reversed (initially full list)

```
intlist_t rev2(intlist_t rest, intlist_t done) {
  if (isempty(rest)) return done;
  intlist_t nxt = rest->nxt; // directly change pointers
  rest->nxt = done;          // link first cell to done part
  return rev2(nxt, rest);    // tail-recursive, becomes loop
}
intlist_t rev(intlist_t lst) { return rev2(lst, empty()); }
```

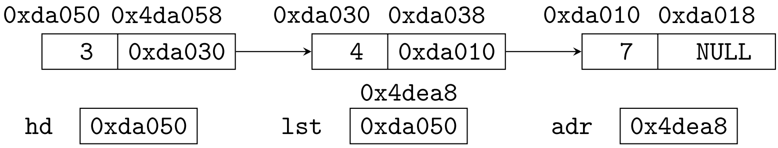
Traversing linked list with address of pointer

When inserting/deleting into a linked list (e.g. *ordered* list),
must change link in cell *prior* to the one inserted/deleted

- keep *address* of pointer to be changed (address of link field)
- better than with address of previous element (may not exist)

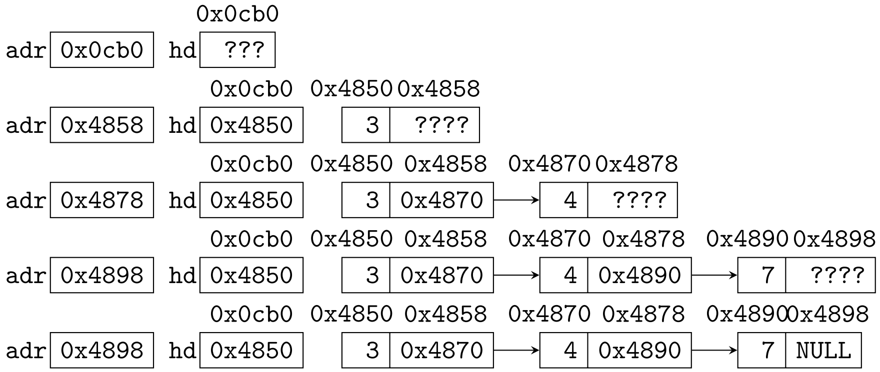
```
intlist_t hd = cons(3, cons(4, cons(7, NULL))); // in main
void trav_addr(intlist_t lst) {
  for (intlist_t *adr = &lst; *adr; adr = &(*adr)->nxt)
    printf("adr: %p, *adr: %p\n", adr, *adr);
} // might print:
adr: 0x4dea8, *adr: 0xda050
adr: 0xda058, *adr: 0xda030
adr: 0xda038, *adr: 0xda010
```

In picture, top row denotes *addresses* of individual fields



Creating a list using addresses of pointers

```
intlist_t rdlist(void) {      // read ints and place in list
  intlist_t hd, *adr = &hd;  // address where t<o link next cell
  for (int n; scanf("%d", &n) == 1; adr = &(*adr)->nxt)
    (*adr = malloc(sizeof(*hd)))->el = n; // malloc and set elem
  *adr = NULL; // done, set link to next cell to NULL
  return hd;   // value from first cycle or NULL above if empty
}
```



Implementing a queue ADT

Queue: first-in, first-out (FIFO): insert/remove at different ends

```
#ifndef _QUEUE_H
#define _QUEUE_H

typedef struct q *queue_t;

queue_t q_new(void);
int q_isempty(queue_t q);
int q_get(queue_t q);
queue_t q_put(queue_t q, int el);
void q_del(queue_t q);
void q_print(queue_t q);

#endif
```

Implementing a queue

Use a *dummy* cell before actual first element; each `get` deletes it, next cell becomes dummy. Invariant: empty queue has `hd==last`.

```
typedef struct e {    // cell for element, with pointer to next
  struct e *nxt;
  int el;
} elem_t;

struct q {
  elem_t *hd;         // dummy; actual first cell is next
  elem_t *last;       // last cell (or dummy if empty)
};

queue_t q_new(void) {
  queue_t q = malloc(sizeof(struct q));
  q->hd = q->last = malloc(sizeof(elem_t)); // both dummy cell
  q->hd->nxt = NULL;                        // no actual element
  return q;
}
```