Computer Programming

Modular compilation. Abstract data types

Marius Minea

marius@cs.upt.ro

8 December 2015

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

C preprocessor

Preprocessing is done prior to compilation: `cpp` or `gcc -E` :

header file inclusion

```
#include <file.h>
```
or
```
#include "file.h"
```

conditional compilation: e.g. to avoid multiple inclusion

```
#ifndef _MYHEADER_H
#define _MYHEADER_H
// contents of header here
#endif
```

also: **`#ifdef`**, **`#undef`** *name*, **`#else`**, **`#elif`**, **`#error`**

can test arbitrary *constant* (compile-time) expressions

```
#if __ORDER_LITTLE_ENDIAN__
// code only gets compiled if this true
#endif
```

Typical library structure

function *declarations*: in `mylibrary.h`

```
#ifndef _MYLIBRARY_H
#define _MYLIBRARY_H
// function declarations (prototpes) go here
#endif
```

library code (function *definition*) in `mylibrary.c`
has `#include "mylibrary.h"` (declaration/definition consistency)

library compiled to *object code*: `gcc -c mylibrary.c`
produces `mylibrary.o` (with *symbols* for function names)

main file has `#include "mylibrary.h"` and uses functions
compile with `gcc program.c mylibrary.o`

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

Lists as abstract data types

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 ?
$cons : E \times L \to L$	list constructor
$head : L \to E$	head of list
$tail : L \to L$	tail of list

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`

$\Rightarrow$ not knowing structure, user program cannot access fields

For example, the `FILE` datatype 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
```