Computer Programming

Modular compilation. Preprocessor. Abstract data types

Marius Minea

marius@cs.upt.ro

9 December 2013

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; *function* scope (`goto` labels)

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* comprising a program
  - default for functions and file scope identifiers;
  - explicit with `extern` in declaration
- *internal*: same within one translation unit; `static` keyword
- *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 without initializer, and with no storage class specifier or with `static` is a *tentative definition*
if several, must match, becomes definition by end of translation unit

How to use in practice

functions: define in one file, declare in all others
variariables: 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`**

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

```
#if sizeof(int) == 2
// code only gets compiled if this true
#endif
```

Preprocessor macros

object-like macro

```
#define NAME replacement
```

function-like macro

```
#define NAME(arg1,...,argn)      replacement
```

replacement can refer to arg1, ... argn

```
#define NAME(arg1,arg2,...)   replacement
```

can use `VA_ARGS` to refer to extra arguments

In macro replacements:

`# arg` produces string literal for tokens represented by `arg`

`x ## y` produces string concatenation of tokens for `x` and `y`

```
#define STR(s) #s
#define STRSUB(s) STR(s)
#define JOIN(x,y)      x ## y
#define SFMT(m)        STRSUB(JOIN(%m,s))
#define MAX            32
scanf(SFMT(MAX), s);  // scanf("%32s", s);
```

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 a class of datastructures

defined by the operations that can be performed on 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 without affecting the client program, which only relies on the interface.

Lists as ADT

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

$nil : () \to L$	empty list constructor
can also be a constant rather than function	
$cons : E \times L \to L$	list constructor
$head : L \to E$	head of list
$tail : L \to L$	tail of list
$isempty : L \to Bool$	is empty ?

and the *axioms*

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

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