Computer Programming

Modular compilation. Preprocessor
Abstract data types. Exceptions

Marius Minea

marius@cs.upt.ro

9 December 2014

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* (files) making up program

default for functions and file scope identifiers;

explicit with `extern` 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 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 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
#define MAX(a,b)          ((a)>(b)?a:b)
#define NAME(arg1,arg2,...)   replacement
```

can use VA_ARGS to refer to extra arguments

CAREFUL with macros: put args in parantheses in macro body

Don't use with side-effects if arg evaluated twice: `MAX(x++,y)`

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 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 language 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

For example, the `FILE` datatype enforces such an encapsulation

Why exceptions ?

Error handling is absolutely needed for any environment interaction

Also needed when proper result can't be returned
non-numeric string to number; 5th element of 3-element list

Error situations can happen anywhere in the "normal" control flow
end-of-file, read error, insufficient memory
or user-level errors (input does not match format)
handling complicates code, obscures the main functionality

Functions must be designed to return error conditions
complicates their interface

User code has to check for errors *at all points*
and propagate recovery up from from deep within processing

Exceptions as a programming language feature

Exceptions are a control flow mechanism
  different from function call/return, breaking from loops
  can transfer control across functions

Exceptions are *raised* and *caught* (handled)
  can be raised by a library function, or by the user

Imagine a statement that says:

`setup` *exception-name* `in` *protected-code* `with` *handler-code*

When this is executed, the runtime system sets up things so that if the named exception appears (is *raised*/*thrown*) when executing *protected-code*, control is transferred to the handling code.

If nothing happens, execution proceeds with the next statement.

Syntax varies:

Java: `try` *protected-code* `catch (` *exception* `)` *handler-code*

ML: `try` *protected-code* `with` *exception* `->` *handler-code*

Exceptions in C: setjmp/longjmp

```
#include <setjmp.h>
jmp_buf myexc;
...
if (setjmp(myexc)) {
  // nonzero: exception was thrown, handle here
} else {
  // protected code where exception is caught
}
...
// somewhere else, usually in another function
longjmp(myexc, nonzero); // throws myexc with nonzero param
```

Can handle in a `switch`, to distinguish values from `longjmp`:

```
switch (setjmp(myexc)) {
case 0: /* normal code that may throw myexc */ break;
case val1: ...; break;
case val2: ...; break;
default: /* any other value */
}
```