Computer Programming

# Preprocessor. Modular compilation. Abstract data types

Marius Minea

marius@cs.upt.ro

13 December 2016

# C preprocessor: Macros

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

*object-like macro*

```
#define NAME      replacement
#define LEN       20
```

*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

define a symbol witout value: used in conditional compilation

```
#define   NEEDS_MATH_H
#undef    SOME_DEFINED_NAME
```

undefine a defined macro

Macros are NOT ~~variables~~. The are like find-replace in a text, actual compiler never sees macros, just code after replacement.

*CAREFUL* with macros: put args in parantheses in macro body
Don't use with side-effects if arg evaluated twice: `MAX(x++,y)`

# Advanced macros: from tokens to strings

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);
```

# Conditional compilation

C preprocessor supports conditionals, using *constant* expressions
only the corresponding branch of the code will be compiled

```
// convert from byte buffer (least significant first) to int
#if __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
// if both symbols are #define'd and their value is equal
// compile code for big-endian architectures
uint16_t x = b[0] | b[1] << 8; // different order
#else
// code for little-endian architecures
uint16_t x = *(uint16_t)b;   // same order
#endif
```

also: `#elif` meaning else if ...

`#ifdef NAME` if `NAME` is defined `#ifndef NAME` if `NAME` is not defined

# Header file inclusion and others

*header file inclusion*

`#include <file.h>` search in system directories

`#include "file.h"` search current dir first, then system

*conditional compilation*: e.g. to avoid multiple inclusion

```
#ifndef _MYHEADER_H
#define _MYHEADER_H
// contents will not be compiled twice even if included twice
#endif
```

# How to structure complex programs?

Complex programs are written by multiple users, in multiple files.

How to share variables and functions (global identifiers) ?

How to ensure function used consistently (right parameters) ?

How to declare one's own identifiers without conflict with others?

# 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`: *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)

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

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

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
```