Lecture 10: Modules and separate compilation
23 Feb 2010

Nate Nystrom
University of Texas at Arlington
Key to building large software systems is ability to organize the program into modules with clean, small interfaces

A *module* identifies an abstraction in the program
- e.g., a logical component, an abstract data type
Advantages of modularity

aggregation
encapsulation
independence
separate compilation
code reuse
Aggregation

Modules provide a higher-level organization of the program

Easier to keep track of what different parts of the program are supposed to do
Encapsulation

Information hiding

Implementation details can be made not visible to other modules

Can reimplement module without affecting others
Independence

minimize dependencies between modules

module interface should specify dependencies

makes it possible to develop one part of the program independently of other parts
Separate compilation

Can compile a module separately from others

If one module changes, only that module needs to be recompiled

Each module can be compiled independently, checking against the interfaces of the other modules.

During linking, resolve references from one module to another, constructing a complete program.
Code reuse

Can reuse modules in other programs if they provide other general-purpose functionality

Examples:
- libraries
- frameworks
Separate *specification* and *implementation*

- **specification**
  - describes what a piece of code is supposed to do
  - provided by *interfaces*

- **implementation**
  - describes how to do it
  - provided by *modules*
No, not Java interfaces!
- more on that later

An interface is:
- Collection of types
- Data declarations
- Procedure signatures
Implementations for procedures

Additional data and procedures
- not present in the interface
- private and not available to other modules

Initialization code
Many languages provide modules or module-like structures

But they use different names:
- Pascal – units
- Ada – packages
- Java – packages, classes
- ML – structs (but usually just called modules)
- Modula-3 – modules
Modula-3

We’ll explore the essence of modules using Modula-3

Why M3?
- Not cluttered with other features (viz. objects)
- Not completely ad hoc (C)
Separating spec and implementation

Two constructs:

- Interfaces
- Modules
Interface for a Util module that provides utility functions for arrays of integers

INTERFACE Util;
  TYPE A = ARRAY OF INTEGER;
  PROCEDURE Sort(a: A);
  PROCEDURE Max(a: A): INTEGER;
END Util.

Defines a type A

Provides signatures for two procedures operating on type A.
Modula-3 modules

Implementation of the Util module:

```
MODULE Util EXPORTS Util;

PROCEDURE Sort(a: A) =
  BEGIN (* code for Sort *) END Sort;
PROCEDURE Max(a: A): INTEGER =
  BEGIN (* code for Max *) END Max;
PROCEDURE Compare(a: A, b: A): INTEGER =
  BEGIN (* code for Compare *) END Compare;
END Util.
```

EXPORTS indicates what interface this module implements.
Modula-3 modules

Implementation of the Util module:

```
MODULE Util EXPORTS Util;

PROCEDURE Sort(a: A) =
    BEGIN (* code for Sort *) END Sort;
PROCEDURE Max(a: A): INTEGER =
    BEGIN (* code for Max *) END Max;
PROCEDURE Compare(a: A, b: A): INTEGER =
    BEGIN (* code for Compare *) END Compare;
END Util.
```

Module can define more members than those in interface. Not visible to other modules.
Features

Modules and interfaces
- specify a *namespace* collecting related types, declarations, and procedures

Interface
- specifies names that are exported (i.e., visible to) other modules

Module
- defines the implementation of a given interface
- implementation may include members not exported
C supports *ad hoc modules*

Large programs are modularized by breaking the program into multiple `.c` files

Each `.c` file is compiled into a object (.o) file

Multiple `.o` files are linked together into an executable or library

Use header (.h) files to share declarations
Separate compilation in C

Can compile each .c file separately

Can refer to code in other .c files using `extern`

- `extern int strcmp(const char *, const char *);`
- `extern float BIG_ARRAY[N];`
Separating spec and implementation

*By convention*, separate specification and implementation by using header (.h) files

Put specification for a module in .h file
- type declarations (typedefs, structs)
- variable declarations
- function declarations (prototypes)

Put implementation in .c file

But not quite:
- some implementation in .h file
- inline functions, macros, C++ templates
Information hiding

By default, any declaration in file 1 can be made visible in file 2 by declaring a prototype in file 2.

One mechanism for information hiding in C: static

- static variables and functions in one file are not visible in other files.
Separate specification and implementation with:
- Specification given by *interfaces*
- Implementation given in *classes*

But, separation not enforced
- not *required* to use interfaces
- can use other classes directly
Java interfaces

interface IntRef {
    void set(int x);
    int get();
}

Specifies a set of public methods all classes implementing the interface must provide.

class C implements IntRef {
    private int v;
    public void set(int x) { v = x; }
    public int get() { return v; }
}
A package is a collection of related classes and interfaces.

Namespaces provided by packages
- but, packages don’t separate specification and implementation
- no way to say several *mutually dependent* classes implement several mutually dependent interfaces
- can declare classes, interface, methods, fields to be package-scoped — visible only to other members of the package.
Using a module
Modula-3 imports

To use the module, need to import it into the current module.

```
MODULE Main;
IMPORT Util;
VAR arr = ARRAY [1..10] OF INTEGER;
BEGIN
  (* initialize arr *)
  Util.Sort(arr);
END Main.
```
Modula-3 imports

Can also import individual members.

```
MODULE Main;
FROM Util IMPORT Sort;
VAR arr = ARRAY [1..10] OF INTEGER;
BEGIN
  (* initialize arr *)
  Sort(arr);
END Main.
```
Modula-3 imports

Can also import individual members

MODULE Main;
FROM Util IMPORT Sort;

VAR arr = ARRAY [1..10] OF INTEGER;

BEGIN
  (* initialize arr *)
  Sort(arr);
END Main.

By importing individual members, can avoid accidental conflicts between names.
#include <string.h>

Adds all the prototypes for the string library into the scope

Can “import” one function by writing its prototype:
- extern int strcmp(const char *, const char *):

Note: you can also write the prototype incorrectly:
- extern int strcmp(double);
Java

Provides import statements

```java
import p.C
```
- adds the name C to the scope, referring to p.C

```java
import p.*
```
- adds all the names in package p to the scope: C refers to p.C if p.C exists

These work at the source-file level, not at the package or class level.
Java also provides visibility annotations:
- public, private, protected

Only way for one package to access a declaration in another package is to make the declaration public
- But, now everyone can access the declaration
Initialization
Module can also specify initialization code.
INTERFACE A;
VAR x : INTEGER;
END A.

MODULE A EXPORTS A;
IMPORT B;
BEGIN
  x := B.y;
END A.

Another example with initialization code.
INTERFACE A;
VAR x : INTEGER;
END A.

MODULE A EXPORTS A;
IMPORT B;
BEGIN
  x := B.y;
END A.

Initialization code may refer to components of other modules. This could introduce a cyclic dependency.
INTERFACE A;
VAR x : INTEGER;
END A.

MODULE A EXPORTS A;
IMPORT B;
BEGIN
  x := B.y;
END A.

INTERFACE B;
VAR y : INTEGER;
END B.

MODULE B EXPORTS B;
IMPORT A;
BEGIN
  y := A.x;
END B.
Modula-3 initialization

Modula-3 specification:

"If module M depends on module N and N does not depend on M, then N’s body will be executed before M’s body, where:
- A module M depends on a module N if M uses an interface that N exports or if M depends on a module that depends on N.
- A module M uses an interface I if M import or exports I or if M uses an interface that (directly or indirectly) import I.

Except for this constraint, the order of execution is implementation-dependent."
Cyclic dependencies

To rule out non-deterministic behavior, the language can either:

- be more restrictive and rule out cyclic module dependencies via imports/exports, or
- use a more complex analysis to rule out cyclic assignments of data between different modules

The ML module system (similar to Modula-3’s) rules out cyclic dependencies
Similar problem occurs in Java. But semantics are specified.

class A {
    static int x = B.y + 1;
}

class B {
    static int y = A.x + 1;
}

Classes are initialized the first time they are used at run-time.

If A is first class used: initializer for A runs, loads B, initializer for B runs, B.y = 1 (since A.x still 0), then A.x = 2.
“static initialization order fiasco”

“a subtle way to crash your program”

Order of initialization depends on the linker and loader implementation.

Linker optimizations can break your code.
Workaround

Wrap initialization code in a function.

```c
int x = e;

-->
int x() {
    static int x = e;
    return x;
}
```

will be initialized on first use (like Java)
Type abstraction
Opaque types

Modula-3 also lets you hide the structure of types in an interface.

Referred to as *opaque types* (in M3) or *abstract types* (in other languages) or *existential types* (in still other languages)

An opaque type is defined only in the module implementation.
Opaque types

INTERFACE Rational;
TYPE T <: REFANY;
EXCEPTION Err;
PROCEDURE Create(n, d: INTEGER): T RAISES {Err};
PROCEDURE Add(r1, r2: T): T;
...
END Rational.
MODULE Rational;
REVEAL T = REF RECORD
    num, denom: INTEGER
END;
EXCEPTION Err;
PROCEDURE Create(n, d: INTEGER) =
    BEGIN
        IF d = 0 THEN RAISE Err; END;
        RETURN NEW(T, num := n, denom := d);
    END Create;
PROCEDURE Add(r1, r2: T) = ...
END Rational.
opaque types

MODULE Main;
IMPORT Rational;

VAR r: Rational.T;

BEGIN
  r = Create(3,2); ...
END

Client of Rational module can refer to type T without knowing its structure.
Abstract types in C

Yes!

/* forward declare a struct without defining its structure */
struct S;
...
/* use the struct */
struct S* x;

But: must can only use a struct without a definition as a pointer. This is because the compiler does not know the size of the struct.
Generic modules

Much like generics in Java, or templates in C++, one can parameterize a module or interface.

Modules are parameterized on interfaces.

```
GENERIC INTERFACE GUtil(Elem);
TYPE A = ARRAY OF Elem.T;
PROCEDURE Sort(a: A);
PROCEDURE Max(a: A): Elem.T;
END GUtil.
```

Generic modules are called a functor in ML.
Generic modules

GENERIC MODULE GUtil(Elem);

PROCEDURE Sort(a: A) =
   BEGIN
      ... Elem.LessThan(a[i], a[j]) ... 
   END Sort;

PROCEDURE Max(a: A) =
   VAR x = Elem.Smallest();
   BEGIN
      ... Elem.LessThan(a[i], a[j]) ... 
   END Max;

END GUtil.
Generic modules

INTERFACE IntElem;
TYPE T = INTEGER;
PROCEDURE LessThan(a, b: T): BOOLEAN;
PROCEDURE Smallest(): T;
END IntElem.

INTERFACE IntUtil = GUutil(IntElem);
END IntUtil.

MODULE IntUtil = GUutil(IntElem)
END IntUtil.
Import vs. requires

One way to think of a generic module is as a module that \textit{requires} another module.

Imports let you refer to code in another module.

“requires” let you specify that you depend on another interface but not on a particular implementation of that interface.

Other module systems (e.g., units for Scheme) let you specify requires more explicitly.
Generic classes and interfaces

Can parameterize a class or interface on a type.

Can specify bounds on the parameter:

class SortedSet<T extends Comparable<T>> { ... }
Templates

Can parameterize on types and also values.
- `template<typename T> class List { ... };`
- `template<int N> struct Factorial {`
  `  static const int value = N * Factorial<N-1>::value;`
`  };

But, need the source code of the template to use it. Separate compilation lost!
Java imports

Use import statements to use code in other packages.

import p.C; // imports class (interface) C from package p
import p.*; // imports all members of p

Scope of import is a single file, not entire package.
Visibility rules

Named entities (classes, interfaces, fields, methods, constructors, ...) have the following access modifiers:

- public
- protected
- (default, package-scoped)
- private
Visibility rules

public
- can access the name from any scope

protected
- can access the name from within the same class, from within any subclass, from within the same package

default
- can access the name from within the same package

private
- can access the name from within the same enclosing top-level class
  - a nested class can access the private members of another nested class
Separate compilation

When compiling Java source, may invoke method, access field of another class
- Need to know the types of these members

javac compiles each class or interface file into a .class file
- .class file contains this type information
- javac can read type information for a .class file to type-check an access to a member of that class
Class files contain:
- class name
- superclass name
- interface names
- member class names
- modifiers (private, public, etc)
- method names, return types, parameter types, exceptions thrown, modifiers (+ bytecode for the methods)
- field names, types, modifiers, constant values
- annotations on classes, fields, methods
Separate compilation

To support separate compilation, there needs to be some way for the compiler to resolve names in other modules without having to recompile the other modules

- C: uses header files and function prototypes and extern
- M3, ML: uses interface declarations
- Java: compiled form of code records type information
Source files compiled into object files
- foo.c -> foo.o

Each object contains symbols corresponding to names of declarations in the source

Some symbols may be unresolved -- refer to other files
These are the symbols declared extern in C

**Linker** combines several object files into an executable, resolving the unresolved references
Loader turns an executable file on disk into a process in memory

Loader may also load dynamic libraries for dynamically linked executables
Object files

Contains:
- code (aka text)
- data
- symbol table
- string table

- code and data are *relocatable*

Compiler generates object files for a single compilation unit
- foo.c --> foo.o
- .o on Unix, .obj on Windows

Linker combines object files into other object files
- a.out, .so, .dylib, .com, .exe, .dll are all “objects”
Loading an object file

A much simplified version of loading

Object file

- data
- code

loader maps object file into memory, then jumps to main

Process in memory

- stack
- heap
- data
- code

$2^{32} - 1$
Unrelocatable objects

For this simplified version of loading to work

- the offset of a piece of code/data in the file must be the same as its address in memory

- Ex:

  ```
  int main() { ... }
  ```

  ```
  pushq %rbp
  movq %rsp, %rbp
  ```

- loader must map main to address 0x1234

- Windows: .com unrelocatable, .exe relocatable
Un relocatable objects

Object file contains absolute addresses

Mapping of object files into memory cannot relocate code and data

=> Cannot support shared libraries

=> All code for a program must live in the same file
Relocatable objects

Generate code with symbolic references to locations of code/data defined within the object and to locations outside the object

- If program contains references to printf, object code will contain only symbolic reference to “printf”
- If program exports a global variable x, object file generates a symbol for “x”.

Object file contains relocation entries to allow linker/loader to patch code that’s relative to a symbol

**Linker** can combine several object files into one, moving code

**Loader** is allowed to move objects to any address in memory
- Loader decides actual address of a given symbol
Relocation entries

Linker and loader can both resolve references to symbols

Relocation entries
- location (offset) within object file to be relocated
- symbol number

Symbol entry
- the name of the symbol (e.g., main)
- value (offset in file) if a local symbol

Loader goes through relocation entries
- writes to address (offset) to be relocated the actual address of the symbol
Linking

Loader is allowed to move objects to any address

Relocatable objects contain symbols
- names of code or data defined in other files

Linker takes one or more relocatable objects
- matches up undefined symbols with their definition in another object
- generates new object file with symbols resolved
  - might be a *statically linked* executable—unrelocatable
  - might contain undefined symbols for code in shared libraries
Name mangling

Don’t want symbols to conflict with reserved names, or names in run-time libraries

C
- main -> _main

Fortran
- calc -> _calc_

C++
- need to encode type information because of method overloading
- Pair(char*, char*) -> __ct__4PairFPcPc
Module systems help to structure large programs

To support separate compilation, need to be able to find type information for code/data in other files without having the source code

- C: use extern declarations, header files
- Java: use class files
Questions?