8. INTERACTION+INTERFACE DESIGN

Last Mod: 2024-06-16

This slide file is for Section 8b, and generally focus on the second of these two bullets:

Section 8 of the course will look at:

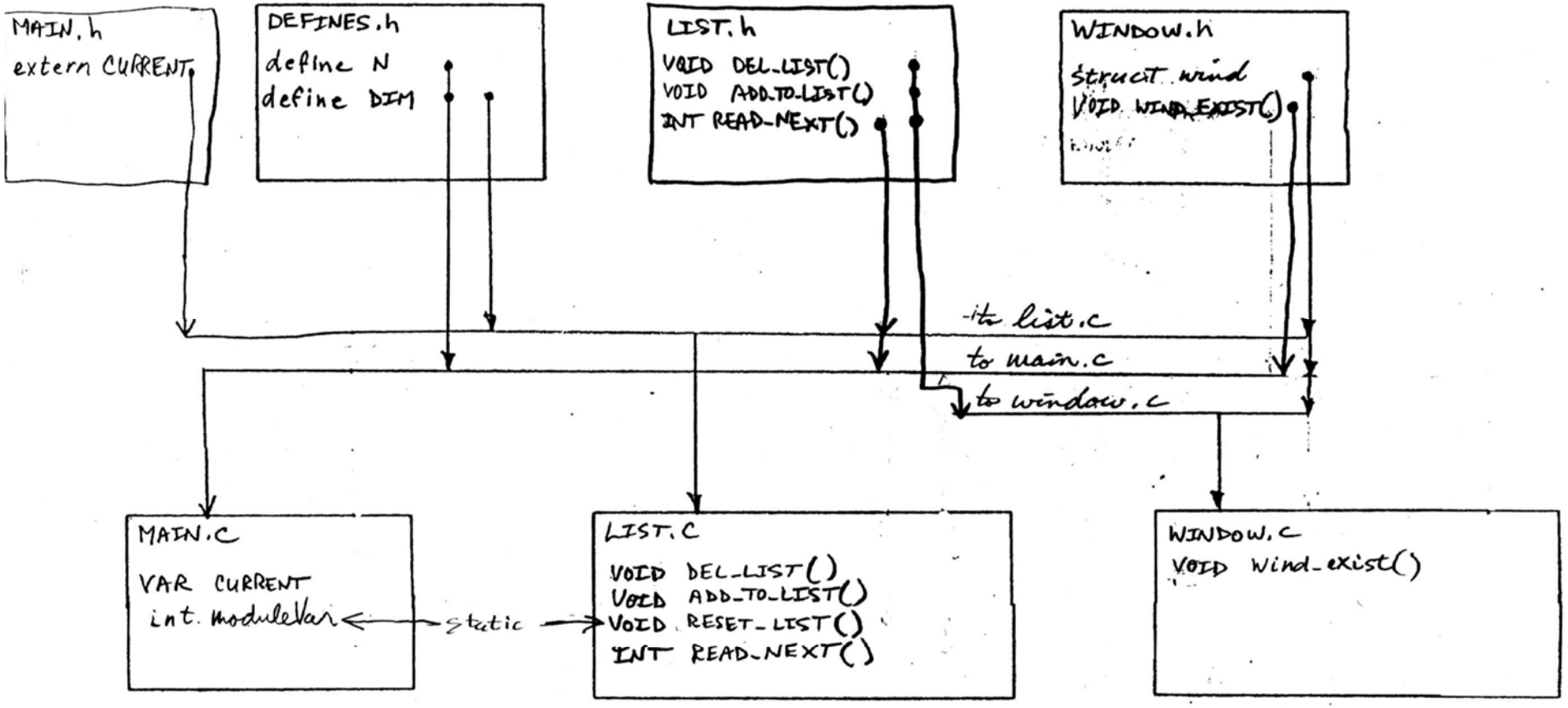
- Planning the **interaction** of the software modules and objects in the system.
  - I.e. Which object/module calls which other, when, and in what order.
- Actual function prototype parameter and return type details and strategies, that make up the interfaces that modules and classes advertise to each other.

Required Readings: None

Table of Contents

8.6 Modular Design Chart

In addition to the “overall OCD” discussed previously, there’s another slick way to show calls (and other dependencies) using a Modular Design Chart:



NOTES:
1) Header files go on top. If using ANSI C, procedures can be exported from function prototypes in the header files rather than the .c files.
2) Generally, there is one horizontal line in the software bus for each importing module. Vertical line arrows show export direction.
3) Diagram can be extended on right by taping sheets together and then accordian folding them.

- Header files go at the top.
- Horizontal lines are one per destination.
  - The collection of horizontal lines is kind of a software dependency bus.
- Vertical lines show source and destination
- Easier to portray many elements with many dependencies, without a tangled mess.
- Diagram can be extended to the right very easy.

8.7 Software Interfaces

Let's discuss designing the internal interfaces between subsystems, modules, and functions.

- It is very important to export a nice set of functions from a module or class that will allow a programmer using that module or class to get done what needs to get done.
- In addition, the documentation of an interface is extremely important for other programmers to be able to successfully call your module.

Russell's definition of a stress test:
one where the system is passed invalid input

- You should NOT assume that what a module does and how you get it to do it, is as obvious to those who did not write the module, as it is to the author.
  - This is a major failing of programmers.
  - Stepping back, away from the knowledge they know about a module, and thinking about how hard it would be to understand this module if they hadn't written it.
  - Document an interface as if your best friend was going to have to call it without being able to consult you about how the various functions and parameters (and types and global variables and constants) work.

8.7.1 Software Interfaces

Internal interfaces are specifications of how one subsystem, module, class, or function can interact with another. Methods of interaction are:

- function call
- Remote procedure call (e.g. call a function in a different running program in another (or this) computer).
- Unix inter-process pipe
- shared/global variable (possibly protected by mutex)
- Network packet communicaitions

It's important in teams, or in a company, with teams of teams, that software interfaces be written first, and passed to other team members or other teams.

- This allows work by multiple programmers to proceed in parallel.
- You will find in the Cmpt-276 project, that before authoring calls to each other's modules, their authors will need to tell you the:
  - names of their functions, and
  - number and type of each parameter of each function, and return type.
    - The latter might the might be complex objects whose outward aspects need to be pre-declared.

- And to comment on special things to know about their functions (e.g. “call init() before any other procedures in this module”).

In other words, they must write a well-commented C++ header file for their module and drop/give/email it to you.

- You can’t wait until their module is written and debugged, because you want to write code right now that calls their functions.

- When dealing with languages that have separate header and implementation files (e.g. C, C++, Modula-2, Modula-3, Ada), the comments describing each parameter should be in the header file, and NOT IN THE IMPLEMENTATION MODULES. The reasons for this are:
  - You often write C++ header files first so other team members can *test compile* their modules which call yours.
  - They need to understand how to call your objects/modules, and to provide their compiler with prototypes and any globals.

The comments should NOT *also* be in the implementation module, because if you need to change a comment, you will surely forget to change it in one of the two places!

- If this happens too much, then the code will become unmaintainable because the comments will not be right.

A software interface is just the documentation (for humans or compilers) regarding how one module can interact with another.

If you're writing system software that is called by application programmers, then these interfaces specs are called Application Programmer Interfaces (APIs).

In Java, there are no header files.

- But you can define classes with public member functions and data members.
  - The classes initially don't need private member function and private data members which will come later when the class is fully implemented.'
  - These private parts are just implement details anyway, and need not concern you: a calling programmer.
  - Note: Later when fully written, there's even a Java tool that will read such files and extract the basic public stuff in the source file and leave out, the useless (from an interface documentation point of view) function bodies.

- There are also special comment formats in Java that allow a programmer to write specially-formatted comments about functions. The special Java documentation extractor will extract the public class information, member function signatures, and those special comments that document what a function does (vs. non-special comments describing what some loop does).
- The same exists for C++. E.g. doxygen

There are several good books on good software coding and interface styles.

- E.g. [McConnell93].

8.7.2 What's Exported /Advertised

Modules and classes are more than just functions. They include exported:

- Constants
- Named Types (including classes, enums, unions).
- Variables or data members
- Exceptions that functions might throw.
- Functions.
- And certain interaction restrictions that an importing programmer must abide by.
  - E.g. Must call **`init_stack()`** before any other stack functions.

Interaction restrictions must be documented in programmer-written comments in the interface definition, or class .h file.

- They are just as essential as the definitions themselves.

“Function signatures” include:

- name
- parameter names and types
- if applicable, return type
- if applicable, the exception raises set.

The parameter list must also show which parameters are passed by reference/pointer.

[I will henceforth call these generically “reference parameters”].

- For reference parameters, you should document whether the parameter is an
  - ‘in’ parameter, the parameter is not changed
  - ‘out’ parameter, or parameter is modified
  - ‘in/out’. Parameter is modified and returned

A reference parameter can be used to:

- pass a large parameter in (without wastefully copy constructing a duplicate), or
- pass something in that needs to be change by the called function and returned not through the functions “return value”, or
- pass something in that is empty or points to no where, that the called function will fill out or point to correctly for the caller.

This parameter direction commenting is required in some Vancouver SW companies.

And is required for all reference parameters in Cmpt-276 assignments.

Some companies I've worked at also required reference parameters to indicate whether "ownership" was being passed in or out.

- When a pointed-to element is passed in, is the function being called responsible for deleting it, or will the caller later delete it.
- When a pointed-to element is in/out or out in nature, is the caller responsible for eventually deleting it, or not.

The above documentation helps prevent "memory leaks" where objects that are no longer used unfortunately take up space in RAM forever.

8.7.3 Scope

There is a difference between scope of a:

- Type's name
- Variable of that type (which is a storage location).
- Variable's contents (or the content pointed to by a reference type)

E.g. Tree type, Tree variable, Tree contents.

- In what module is the Tree type defined or hidden?
- In what possibly-different module is a tree variable defined in?
- What are the actual values in the tree?

These are called the scope of the type, and of the variable.

8.7.4 Scope and Persistence

Do not get an object or variable's persistence mixed up with its scope.

Scope can be:

- Block wide (e.g. C++ function or loop)
- Class wide
- Module wide
- C++ namespace or Java package wide (though technically these are not scopes)
- Global to the entire program
- A file can be accessed by a whole program, several programs, or network.

Persistence can be:

- Function or block lifetime
- Dynamic (in heap from new() to delete().)
- Program lifetime (like static/shepherd members, static local variables, global variable).
- File duration.

Local automatic variables: alive only for the scope it is a part of
Local static variables: alive for the entire program

Almost any combination of scope and persistence is possible.

- E.g. If labelled **`static`**, local variables do not lose value and disappear.

Generally, use minimum scope to reduce coupling.

- Of course, must have enough scope to be reachable from wherever needed.

8.7.5 Exceptions

Requests from service from a module can result in a normal functioning and return, or an exceptional situation. E.g.

- divide by zero,
- file not present,
- no more space in file,
- stack empty,
- item not found in list.

Exceptional situations arise from:

- Inherently illegal requests
  - E.g. stack empty
- Resource limitations
  - E.g. thumbdrive full
- Operational limitations of many types
  - E.g. call init() before other functions in a module.

Some exceptional cases are detectable with only minor effort.

- Others are not detectable (e.g. passing wrong address).
- Or cost and performance restrictions may suggest not bothering.

Recommendation:

- Publish restrictions in the interface comments on how to prevent exceptional conditions.
- Detect as many as are feasible and notify calling ("client") code.

Client code:
Code that programmers write to call other people's code

Three ways to notify client code:

- Return data flag/code/bool indicating exceptional situation has arisen.
  - Function return value
  - Parameter return value
  - Network message
  - Set global variable.

  This instructor has seen all 4 in real code.

  However, client programmers are notoriously lazy at bothering to check return codes.

```
filePointer = fopen(fname,”readonly”);
if (filePointer == NULL){
    printf(stderr, “Can’t open file”);
    exit(-1);
}
```

- Passing of a function to be called in an exceptional situation.
  - This is one of key reasons to allow function pointers.

It's easiest to first declare a type that defines a function pointer:

```
typedef double Ftype(int);

//The following defines space for a
//function pointer, and
//sets it to point at the address of
//the function called myErrorFunc().

Ftype * pFunc = &myErrorFunc;

x = someFunction(Ftype pfunc);
```

`someFunction()` now has capability to call **`myErrorFunc()`** if something goes wrong.

One tiny advantage of this is that the caller of **`someFunction()`** MUST pass the address of some function rather than lazily ignoring the possibility of a problem.

- Thirdly, modern programming languages have a convenient exception feature.
  - E.g. Java, C++.

  Allows low level code to “throw” and exception that must be caught and handled by some outer scope or calling function.

  Else program ends.