Section 5: Modularization & ADTs

5.1: Modularization

Modularization

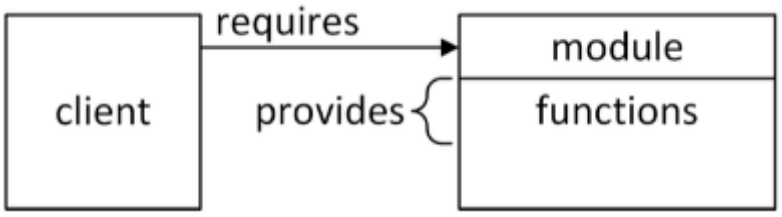
So far we have been designing programs with all of our definitions in a single source (.c) file. However, for larger programs, keeping all of the code in one file is unwieldy. Additionally, teamwork on a single file is awkward, and it is difficult to share or re-use code between programs. A better strategy is to use ***modularization*** to divide programs into well defined **modules**.

We have already seen an elementary type of modularization in the form of *helper functions* that can "help" many other functions. We will now extend and formalize this notion of modularization.

A ***module*** *provides* a collection of functions† that share a common aspect or purpose.

In this course, and in much of the "real world", it is considered **good style** to store **modules in separate files**. While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

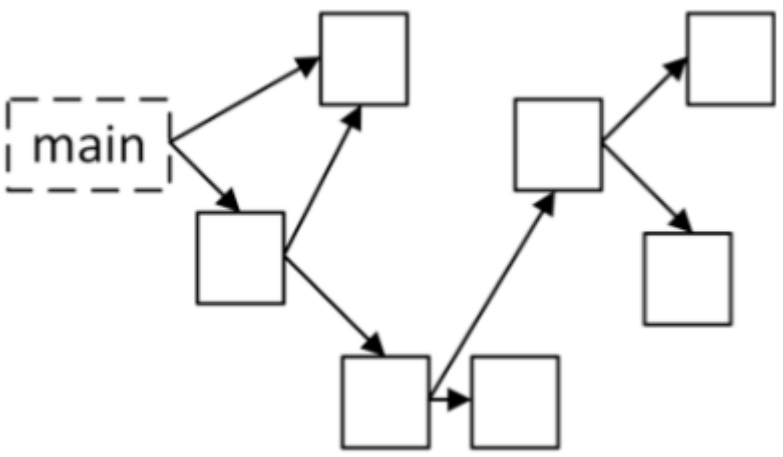
It is helpful to think of a "client" that requires the functions that a module provides.



In practice, the client is a file that may be written by yourself, a co-worker or even a stranger.

Conceptually, it is helpful to imagine the client as a stranger.

Large programs can be built from many modules. A module can be a client itself and require functions from other modules.



The module dependency graph cannot have any cycles, which means no path can start and ends at the same vertex. There must be a "root" (or main file) that acts only as a client. This is the program file that defines the main function and that is "run".

In C we convert ("compile") a source file (.c) into machine code (.o or .ll) before it can be "run". This is fine if we only have to worry about a single file, but once we have modules, we will need to connect them when we do this compilation.

When building a program with modules, we can combine ("link") multiple machine code files together to form a single program. In practice, that means we can "build" a program from multiple source code files and/or machine code files. For example, we may need to combine these files into a single machine code program: main.c, module1.c , module2.c , module3.ll

Modules do not contain a main function. Only one main function can be defined in a program.

`Modules do not contain a` main `function. Only` **one** main `function can be defined in a program.`

There are three key advantages to modularization: re-usability, maintainability and abstraction.

- **Re-usability:** A good module can be re-used by many clients. Once we have a "repository" of re-usable modules, we can construct large programs more easily.
- **Maintainability:** It is much easier to test and debug a single module instead of a large program. If a bug is found, only the module that contains the bug needs to be fixed. We can even replace an entire module with a more efficient or more robust implementation.
- **Abstraction:** To use a module, the client needs to understand **what** functionality it provides, but it does not need to understand **how** it is implemented. In other words, the client only needs an *"abstraction"* of how it works. This allows us to write large programs without having to understand how every piece works.

Declarations

In C, a function or variable must be `declared` before (*"above"*) it can be "accessed" (or referenced).

A declaration introduces an identifier (“name”) into a program and specifies its type.

In C, there is a subtle difference between a **definition** and a **declaration**.

Declaration vs. Definition

- A **declaration** only specifies the *type* of an identifier.
- A **definition** instructs C to *“create”* the identifier.

However, a definition *also* specifies the type of the identifier, so a definition also includes a declaration.

An identifier can be declared multiple times, but only defined once.

A **function declaration** is simply the function header followed by a semicolon (;) instead of a code block. It specifies the function type (the return and parameter types).

A **variable declaration** starts with the **extern** keyword, followed by the type and then the variable name. There is **no initialization.**

5.2: Modules: Scope and Interface

Scope vs. memory

Be careful not to confuse the concepts of *scope* with *memory* (or storage). Remember, **all** global variables (from all files) are in memory **before** the program is “run”, even if they are declared later in the program.

A declaration can bring an identifier **from another file** into scope.

By **default**, C global identifiers are “accessible” to *every* file in the program **if they are declared**. We will refer to this as ***program scope***.

You may want to **“hide”** a global identifier from other files if, for example, it is something required for your module and you do not want to risk the client changing it. To do this, you would prefix the definition with the `static` keyword.

In other words, the static keyword **restricts the scope** of a global identifier to the file (module) it is defined in. We will refer to this as ***module scope***.

Types of scopes:

- **local** (block) identifiers: only available inside of the function (or *block*)
- **global** identifiers:
    - **program scope** identifiers (default): available to any file in the program (if declared)
    - **module scope** identifiers (defined as static): only available in the file they are defined in

We continue to use *global scope* to refer to identifiers that have *either* program or module scope.

Use static to give module functions and variables *module scope* if they are not meant to be **provided**.

Module Interface

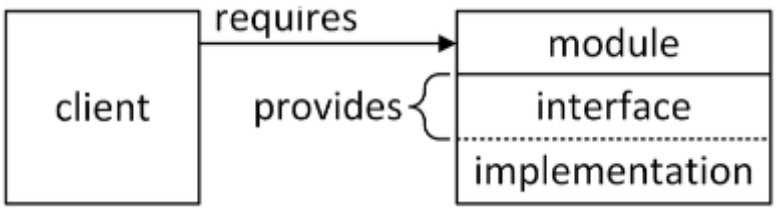
The module ***interface*** is the list of the functions that the module provides (including the documentation).

The *interface* is separate from the module ***implementation***, which is the code of the module (*i.e.*, function **definitions**).

The interface is everything that a client would need to use the module.

The client does not need to see the implementation.

The **interface** is what is provided to the client, whereas the **implementation** is hidden from the client.



The contents of the interface include:

- an **overall description** of the module
- a **function declaration** for each provided function
- **documentation** (*e.g.*, a **purpose**) for each provided function

For C modules, the **interface** is placed in a separate file with a .h file extension.

Clients can read the documentation in the interface (.h) file to understand how to use the provided functions. The client can also “copy & paste” the function declarations from the interface file to make the module functions available if they wish, although you will soon see that there is a much more elegant solution.

A ***preprocessor directive*** **temporarily** "modifies" a source file *just before* it is run (but it does not save the modifications).

The *directive* **#include** "*cuts & pastes*" or "*inserts*" the contents of another file directly into the current file.

Always put any #include *directives* at the top of the file.

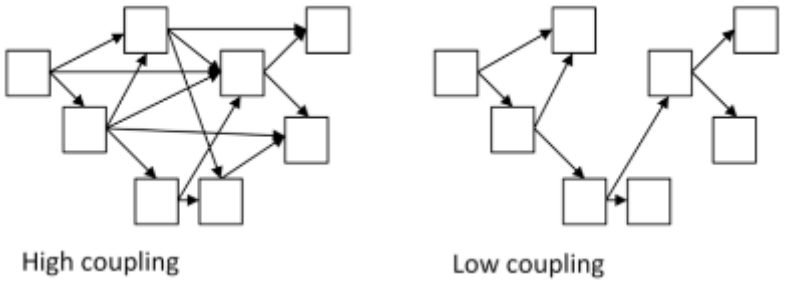
5.3: Module Design

Cohesion and Coupling

When designing module interfaces, we want to achieve ***high cohesion*** and ***low coupling***.

High cohesion means that all of the interface functions are related and working toward a "common goal". A module with many unrelated interface functions is poorly designed.

Low coupling means that there is little interaction *between* modules. It is impossible to completely eliminate module interaction, but it should be minimized.



Interface vs. implementation

We emphasized the distinction between the module **interface** and the module **implementation**. Another important aspect of interface design is ***information hiding***, where the interface is designed to hide any implementation details from the client.

Information hiding

The two key advantages of information hiding are ***safety*** and ***flexibility***.

Safety is important because we may want to prevent the client from tampering with data used by the module. Even if the tampering is not malicious, we may want to ensure that the only way the client can interact with the module is through the interface. We may need to protect the client from themselves.

Additionally, by hiding the implementation details from the client, we gain the **flexibility** to change the implementation in the future.

Because we have used **information hiding** to design our **interface**, our implementation is not only *maintainable*, but also **flexible**.

With C modules, it is easy to hide the implementation details from the client. Instead of providing the client with the implementation source code (.c file) you can provide a machine code (*e.g.*, a .ll file). This is what we did with our cs136 module.

While C is good at hiding the implementation code, it is not very good at hiding **data**. If a **malicious** client obtains (or "guesses") the memory address of some data they can access it directly.

Opaque structures in C

Fortunately, C supports ***opaque structures***, which are "good enough" to hide data from a *friendly* client. An *opaque structure* is like a "black box" that the client cannot "see" inside of. They are implemented in C using incomplete declarations, where a structure is *declared* without any fields.

If a module only provides an *incomplete declaration* in the **interface**, the client cannot create an instance of the struct or access any of the fields within it. Therefore, the definition in the commented out line 11 is invalid.

Of course, if we want a **transparent** structure that the client can access and use without a pointer, we simply put the **complete definition** of the struct **in the interface** file (.h file).

5.4: Introduction to Abstract Data Types (ADTs)

Data structures & abstract data types

In the stopwatch example, we demonstrated two **implementations** with different ***data structures***:

- a struct with two fields (sec and min)
- a struct with one field (seconds)

For each *data structure* we knew how the data was "structured", however, the client doesn't need to know how the data is structured. The client only requires an **abstract** understanding that a stopwatch stores time information.

The stopwatch module is an implementation of a stopwatch ***Abstract Data Type (ADT)***. Formally, an ADT is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

However, in practice (and in this course) ADTs are **implemented** as data storage **modules** that only allow access to the data through interface functions (ADT *operations*). The underlying data structure and implementation of an ADT are **hidden** from the client (which provides *flexibility* and *security*).

The difference between a *data structure* and an *ADT* is subtle and worth reinforcing.

As the **client**, if you have a **data structure**, you know how the data is "structured" and you can access the data directly in any manner you desire.

With an **ADT**, however, the client does not know how the data is structured and can only access the data through the interface functions (operations) provided by the ADT.

The terminology is especially confusing because ADTs are **implemented** with a data structure.

The stopwatch ADT is not a "typical" ADT because it stores a fixed amount of data and it has limited use. A ***Collection ADT*** is an ADT designed to store an arbitrary number of items. Collection ADTs have well-defined operations and are useful in many applications.

The dictionary ADT (also called a *map*, *associative array*, *key-value store* or *symbol table*), is a collection of **pairs** of ***keys*** and ***values***. Each *key* is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations are as follows:

- **lookup**: for a given key, retrieve the corresponding value or "not found"
- **insert**: adds a new key/value pair (or replaces the value of an existing key)
- **remove**: deletes a key and its value

Example: A school database collects student information in key-value pairs

To *implement* a dictionary, we have a choice: use an association list, a BST or perhaps something else?

This is a **design decision** that requires us to know the advantages and disadvantages of each choice. Regardless, the client would never know which implementation was being used.

Stack ADT

We have already been exposed to the idea of a stack when we learned about the **call stack**. The stack ADT is a collection of items that are "stacked" on top of each other. Items are *pushed* onto the stack and *popped* off of the stack. A stack is known as a LIFO (last in, first out) system. Only the "top" item is accessible. Stacks are often used in browser histories ("back") and text editor histories ("undo").

Typical stack ADT operations:

- `push`**:** adds an item to the top of the stack
- `pop`**:** removes the top item from the stack
- `top`**:** returns the top item on the stack
- `is_empty`**:** determines if the stack is empty

Example: A module that tracks browser history (LIFO)

Queue ADT

A queue is like a "lineup", where new items go to the "back" of the line, and the items are removed from the "front" of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- `add_back`**:** adds an item to the end of the queue
- `remove_front`**:** removes the item at the front of the queue
- `front`**:** returns the item at the front
- `is_empty`**:** determines if the queue is empty

Example: A module that tracks orders for pick up (FIFO)

Sequence ADT

The sequence ADT is useful when you want to be able to retrieve, insert or delete an item at any position in a sequence of items.

Typical sequence ADT operations:

- `item_at`**:** returns the item at a given position
- `insert_at`**:** inserts a new item at a given position
- `remove_at`**:** removes an item at a given position
- `length`**:** return the number of items in the sequence

The `insert_at` and `remove_at` operations change the position of items after the insertion/removal point.

Sequence ADT

A set is a group of unordered and unindexed items. The set ADT is useful when you want to be able store unique items without worrying about their order.

Typical set ADT operations:

- `add`: adds a new item to a set
- `remove`: removes an item from a set
- `length`: return the number of items in the set
- `member`: returns whether or not an item is a member of a set

Sometimes operations that allow interactions between two sets are also implemented.