Modularization & ADTs

Optional Textbook Readings: CP:AMA 19.1, 10.2 – 10.5

The primary goal of this section is to be able to write a module.

Modularization

So far we have been designing programs with all of our definitions in a single source (`.c`) file.

For larger programs, keeping all of the code in one file is unwieldy.

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**.

The concept of modularization extends far beyond computer science. There are examples of modularization in construction, automobiles, furniture, nature, etc.

A practical example of a good modular design is an “AA battery”.

We have already seen an elementary type of modularization in the form of *helper functions* that can “help” many other functions.

We will extend and formalize this notion of modularization.

When designing larger programs, we move from writing “helper functions” to writing “helper modules”.

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

† Modules can provide elements that are not functions (*e.g.*, data structures and variables) but their primary purpose is to provide functions.

For convenience in these notes, we describe modules as providing only functions.

Many modern languages have strong support for *modules*.

For example, recently (2020) modules were added to C++, with the new `import` and `export` keywords;

```
export module linear_algebra;
import matrix;
export void row_reduce(...) {...}
```

The C language does not fully support modules as they are understood in modern programming.

However, we can still achieve module-like behaviour and learn the core concepts of *modularization*.

C “modules” are also commonly known as **libraries**.

Modules *vs.* files

In this course, and in much of the "real world", it is considered **good style** to store **modules in separate files**.

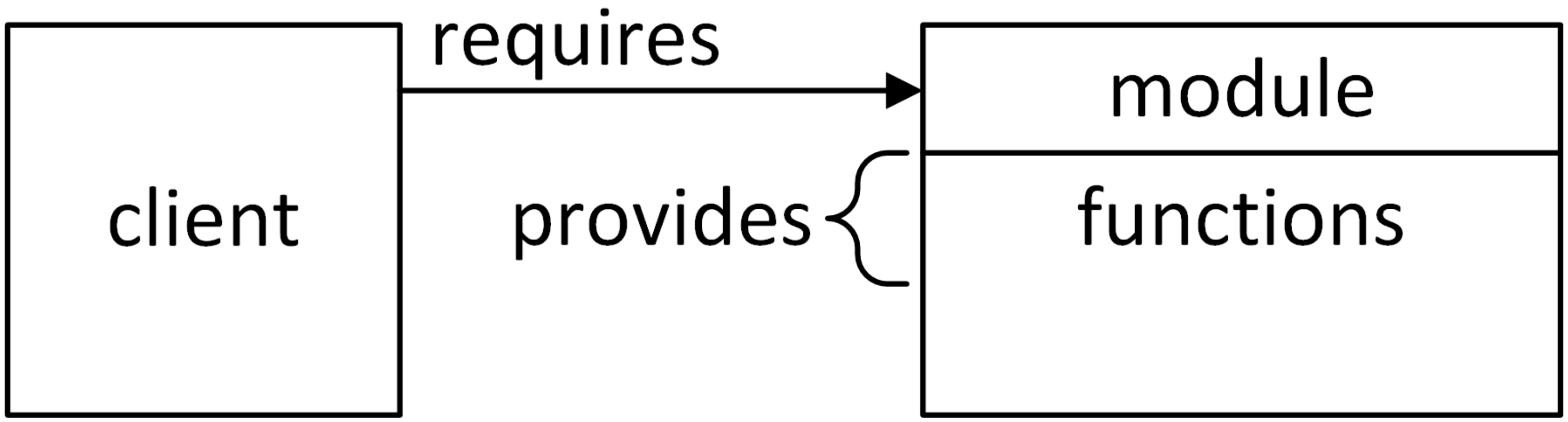
While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

Some computer languages enforce this relationship (one file per module), while in others it is only a popular *convention*.

There are advanced situations (beyond the scope of this course) where it may be more appropriate to store multiple modules in one file, or to split a module across multiple files.

Terminology

It is helpful to think of a “**client**” that **requires** the functions that a module **provides**.

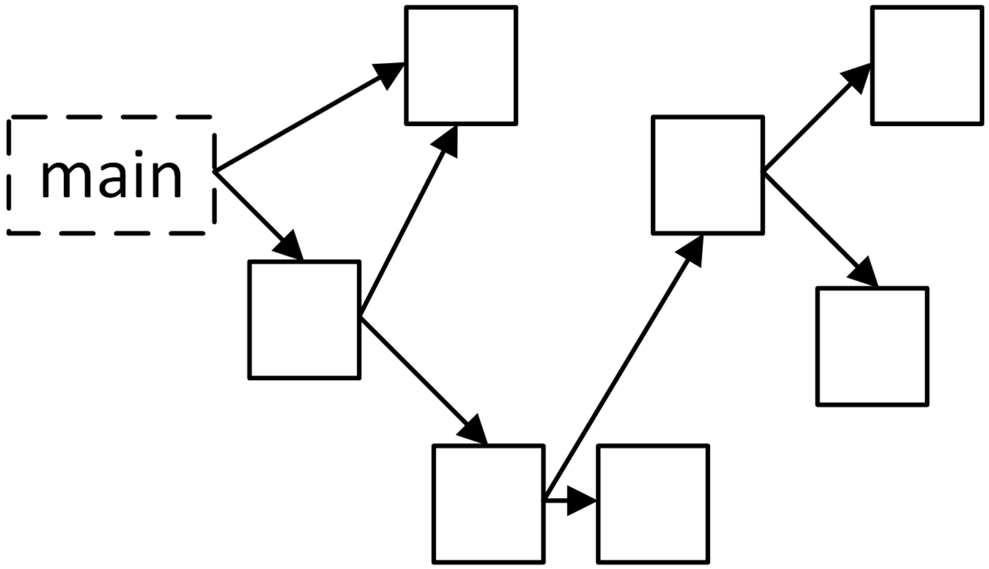


In practice, the client is a file that may be written by yourself, a co-worker or even a stranger.

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.

There must be a "root" (or ***main file***) that acts only as a client.
This is the program file that defines `main` and is "run".

Building a program

In Section 04 we briefly discussed how we convert ("*compile*") a *source file* (`.c`) into machine code (`.o` or `.ll`) before it can be "run".

When building a program, 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.

```
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.

Motivation

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.

Modularization is also known in computer science as the *Separation of Concerns (SoC)*.

example: fun number module

Imagine that some integers are more "fun" than others, and we want to create a **fun** module that **provides** an `is_fun` function.

```
// fun.c [MODULE]

// is_fun(n) determines if n is fun or not

bool is_fun(int n) {
  return (n == -3   || n == 42   || n == 136 ||
          n == 225  || n == 1337 || n == 4010 ||
          n == 8675309);
}
```

We have to learn a few more concepts before we can complete our module.

Our (yet to be completed) **fun** module illustrates the three key advantages of modularization.

re-usability: multiple programs can use the fun module.

maintainability: When new integers become fun (or become less fun), only the fun module needs to be changed.

abstraction: The client does not need to understand what makes an integer fun.

Calling module functions

`is_fun` is defined in the **module** (`fun.c`) file.

How can we call `is_fun` from our **client** (*e.g.,* `main.c`)?

```c
// main.c [CLIENT]

int main(void) {
  //...
  b = is_fun(k);  // ERROR
  //...
}
```

```c
// fun.c [MODULE]

bool is_fun(int n) {
  //...
}
```

The `is_fun` function is not *in scope*.

Also, because C is *statically typed*, it needs to know the return and parameter types of `is_fun`...

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**.

Here, when we use “identifiers” it does *not* include structures (`struct`s) – they are declared differently (more on them later).

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.

Unfortunately, not all computer languages and reference manuals use these terms consistently.

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).

example: function declaration

```
int my_add(int a, int b);         // function DECLARATION

int main(void) {
  trace_int(my_add(1, 2));        // this is now ok
}

int my_add(int a, int b) {        // function DEFINITION
  return a + b;
}
```

By *declaring* `my_add` *above* `main`, it is now available in `main`.

C ignores the parameter names in a function declaration (it is only interested in the parameter *types*).

The parameter names can be different from the definition or not present at all.

```
// These are all equivalent:

int my_add(int a, int b);
int my_add(int, int);
int my_add(int these_are, int ignored);
```

It is good style to include the correct (and meaningful) parameter names in the declaration to aid communication.

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

example: variable declaration

```
extern int g;              // variable DECLARATION

int main(void) {
  printf("%d\n", g);   // this is now ok
}

int g = 7;                 // variable DEFINITION
```

Variable declarations are uncommon.

Revisiting Scope

One way to think about declarations is that they **extend** the *scope* of an identifier.

```
                            // g OUT of scope

extern int g;               // declaration:
                            // g is now IN scope

int main(void) {
  printf("%d\n", g);        // ok
}

int g = 7;                  // definition
```

Scope vs. memory

Remember, don't confuse the concepts of *scope* with *memory* (or storage).

```
int main(void) {
  printf("%d\n", z);    // INVALID: z not in scope
}

int z = 2;
```

Remember, **all** global variables (from all files) are in memory **before** the program is "run".

In the above example, `z` is in memory (and initialized) before `main` is called but `z` is not in *scope* within the `main` function because it has not yet been *declared*.

Program scope

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

```
// main.c [CLIENT]

extern int g;
// g IN scope

int main(void) {
  printf("%d\n", g);  // ok
}
```

```
// module.c [MODULE]

int g = 7;
```

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***.

Revisiting module functions

We have solved our previous problem. We can now call our module function `is_fun` from our **client** (`main.c`) by declaring it:

```
// main.c [CLIENT]

// *** NEW declaration ***
bool is_fun(int n);


int main(void) {
  //...
  b = is_fun(k);   // ok
  //...
}
```

```
// fun.c [MODULE]

bool is_fun(int n) {
  //...
}
```

Module scope

To “**hide**” a global identifier from other files, prefix the definition with the `static` keyword.

```c
// main.c [CLIENT]

// this will not work
extern int g;

int main(void) {
  printf("%d\n", g);  // ERR
}
```

```c
// module.c [MODULE]

static int g = 7;
```

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 scope (revisited)

- **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**.

```
// fun.c [MODULE]

static const int module_scope_variable = 42;

static int is_fun_helper(int n) {
  // ...
}

bool is_fun(int n) {
  //...
}
```

The `static` keyword is **not** related to *static typing*.

`private` would have been a better choice than `static`.

Module interface

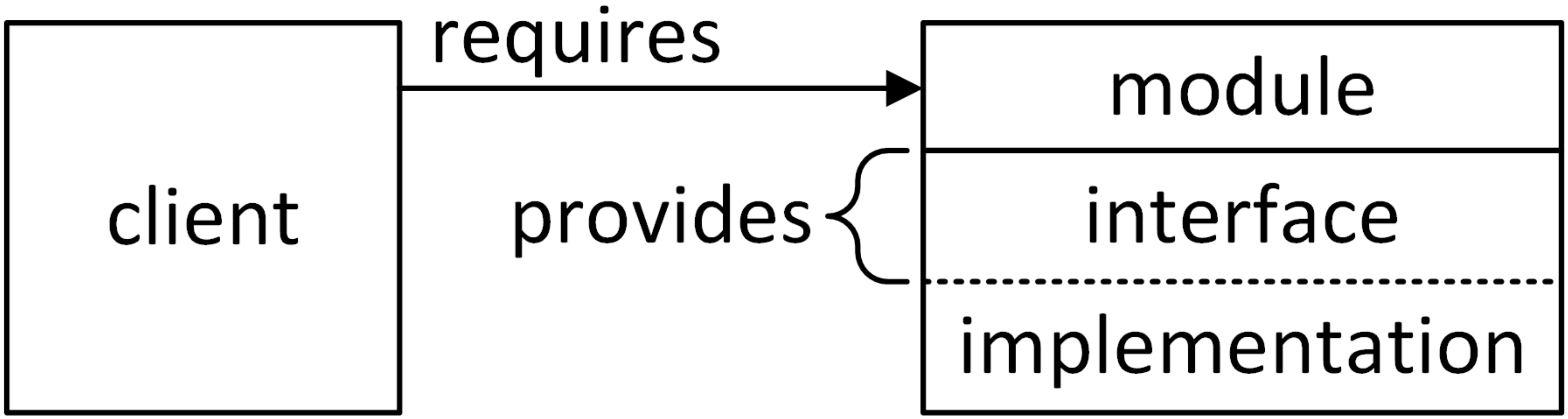
The module ***interface*** is the list of the functions that the module provides (including the documentation).

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.

Terminology (revisited)



The **interface** is what is provided to the client.

The **implementation** is hidden from the client.

Interface contents

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

Ideally, the interface would also provide *examples* to illustrate how the module is used and how the interface functions interact.

Examples are not required in this course.

Interface (.h) files

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

```
// fun.h [INTERFACE]

// This module is all about having fun

// is_fun(n) determines if n is a fun number
bool is_fun(int n);
```

The interface (`.h`) file has everything the client needs.

Interface files are also known as **h**eader files.

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.

```
// main.c [CLIENT]

// *** copied from .h ***
bool is_fun(int n);


int main(void) {
  //...
  b = is_fun(k);  // ok
  //...
}
```

But there is a much more elegant solution.

#include

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

The *directive* `#include` *“cut & pastes”* or *“inserts”* the contents of another file directly into the current file.

```
  #include "fun.h"      // insert the contents of fun.h
```

For clients, this is perfect: it inserts the interface of the module containing all of the function **declarations**, making all of the provided functions available to the client.

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

example: fun module

```
// fun.h [INTERFACE]

// This module is all
// about having fun

// is_fun(n) determines if
//   n is a fun number
bool is_fun(int n);
```

```
// fun.c [IMPLEMENTATION]

#include "fun.h"

// see fun.h for details
bool is_fun(int n) {
  //...
}
```

```
//////////////////////////////////////////////////////////////
// main.c [CLIENT]

#include "fun.h"

int main(void) {
  //...
  b = is_fun(k);
  //...
}
```

Implementation notes

In the previous example, the fun *implementation* file (`fun.c`) included *its own interface* (`fun.h`).

This is good style as it ensures there are no discrepancies between the interface (declarations) and the implementation (definitions).

The function `is_fun` is fully documented in the *interface* file for the client, so in the *implementation* a simple comment referring the reader to the interface file is sufficient.

```
// see fun.h for details
bool is_fun(int n) {
  //...
}
```

For simple (non-pointer) parameters, `const` is meaningless in a function **declaration**.

The caller (client) does not need to know if the function mutates the **copy** of the argument value.

```
int my_function(int x);                      // DECLARATION
                                             // (no const)

int my_function(const int x) {               // DEFINITION
  // mutation of x here is invalid           // (with const)
  // ...
}
```

It is good style to use `const`ant parameters in **definitions** to improve communication.

In addition to #include, The CP:AMA textbook frequently uses the #define directive. In its simplest form it performs a *search & replace*.

```
// replace every occurrence of MY_NUMBER with 42
#define MY_NUMBER 42

int my_add(int n) {
  return n + MY_NUMBER;
}
```

In C99, it is usually better style to define a variable (constant), but you will still see #define in the “real world”.

In this course, we will not use #define.

cs136.h

Since the beginning of this course, we have seen

```
#include "cs136.h"
```

in our programs. We now understand that we have been using support tools and functions (*e.g.*, `trace_int` and `read_int`) provided by a cs136 module.

The `cs136.h` interface file contains:

```
#include <assert.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

So our cs136 module was requiring additional modules.

C standard modules

Unlike Racket, there are no “built-in” functions in C.

Fortunately, C provides several ***standard modules*** (also known as *libraries*) with many useful functions.

For example, the `stdio` module provides `printf` and `scanf`.

When using `#include` we use angle brackets (<>) to specify that the module is one of the *standard modules* and quotes (`""`) for “regular” modules (*i.e.*, ones we have written).

```
#include <stdio.h>
#include "mymodule.h"
```

Here are some of the other modules that we have been using:

- `<assert.h>` provides the function `assert`
- `<limits.h>` provides the constants `INT_MAX` and `INT_MIN`
- `<stdbool.h>` provides the `bool` data type and the constants `true` and `false`
- `<stdlib.h>` provides the constant `NULL`

example: test client

For each module you design, it is good practice to create a **test client** that ensures the provided functions are correct.

```
// test-fun.c: testing client for the fun module

#include <assert.h>
#include "fun.h"

int main(void) {
  assert(is_fun(42));
  assert(!is_fun(13));
  //...
}
```

Do NOT add a `main` function to your implementation (*e.g.*,`fun.c`). Create a new test *client* with a `main` function.

There may be “white box” tests that cannot be tested by a client. These may include implementation-specific tests and tests for module-scope functions.

In these circumstances, you can provide a `test_module_name` function that asserts your tests are successful.

Designing modules

The ability to break a big programming project into smaller modules, and to define the interfaces between modules, is an important skill that will be explored in later courses.

Unfortunately, due to the nature of the assignments in this course, there are very few opportunities for you to **design** any module interfaces.

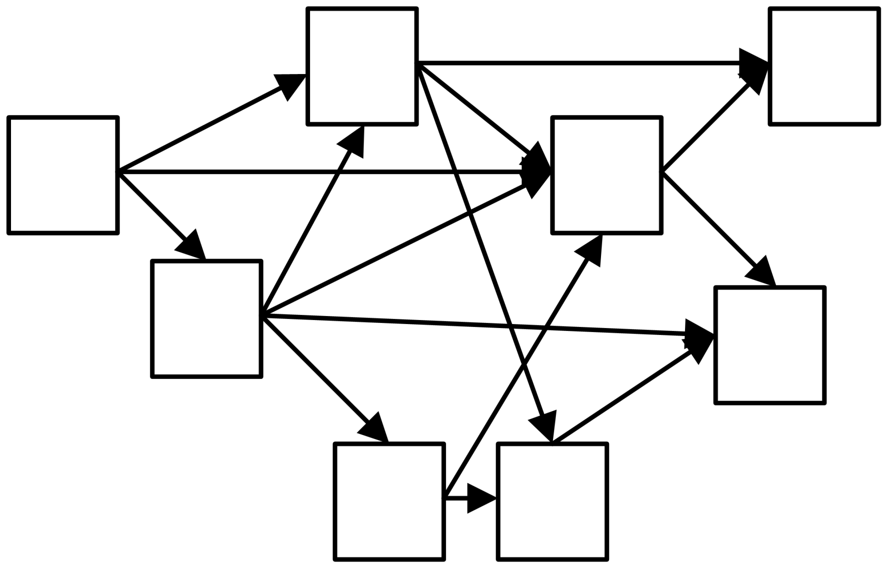
For now, we will have a brief discussion on what constitutes a **good** interface design.

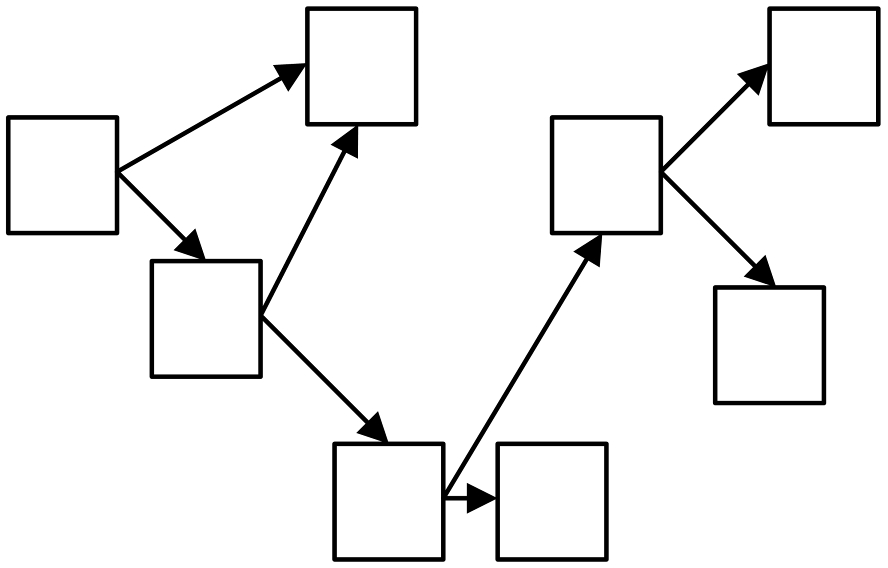
Cohesion and coupling

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.


High coupling


Low coupling

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 ***security*** and ***flexibility***.

Security 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.

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

Information hiding in C

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.

```
struct box;                    // INCOMPLETE DECLARATION
```

With an *incomplete declaration* **only pointers to the structure can be defined**.

```
struct box my_box;             // INVALID
struct box *box_ptr;           // VALID
```

If a module only provides an *incomplete declaration* in the **interface**, the client can not create an instance of the `struct` or access any of the fields.

The module must provide a function to *create* (and *destroy*) an instance of the structure. This is explored more in Section 10.

Of course, if we want a **transparent** structure the client can use, we simply put the **complete definition** of the `struct` **in the interface** file (`.h` file).

example: transparent and opaque structures

```
// module.h [INTERFACE]

struct transparent {
  int t_field;
};

struct opaque;
```

```
// module.c [IMPLEMENTATION]

#include "module.h"

struct opaque {
  int o_field;
};
```

```
//////////////////////////////////////////////////////////////
// main.c [CLIENT]

#include "module.h"

int main(void) {
  struct transparent t = {0};     // VALID
  struct opaque o = {0};          // INVALID
  struct opaque *ptr;             // VALID
}
```

example: stopwatch module

To illustrate the principles of information hiding, we will create a small `stopwatch` module.

For narrative purposes, we can imagine this module is for keeping track of race times and we do not want clients to be able to tamper with the times (this is an example of *security*).

In the **interface**, we have an incomplete declaration.

The client cannot define a `struct stopwatch`, so we need to provide *create* and *destroy* functions.

```
// stopwatch.h [INTERFACE]

struct stopwatch;

// stopwatch_create() creates a new stopwatch at time 0:00
// effects: allocates memory (client must call stopwatch_destroy)
struct stopwatch *stopwatch_create(void);

// stopwatch_destroy(sw) removes memory for sw
// effects: sw is no longer valid
void stopwatch_destroy(struct stopwatch *sw);
```

We learn how to write create/destroy functions in Section 10.

The rest of the interface provides some simple stopwatch functions.

```c
// stopwatch.h [INTERFACE]

// stopwatch_get_seconds(sw) returns the number of seconds in sw
int stopwatch_get_seconds(const struct stopwatch *sw);

// stopwatch_get_minutes(sw) returns the number of minutes in sw
int stopwatch_get_minutes(const struct stopwatch *sw);

// stopwatch_add_time(sw, min, sec) adds min[utes] and
//   sec[onds] to sw
// requires: 0 <= minutes, seconds
// effects: modifies sw
void stopwatch_add_time(struct stopwatch *sw, int min, int sec);
```

The **implementation** fully defines a `struct stopwatch` and simple functions to access the fields of the structure.

```
// stopwatch.c [IMPLEMENTATION]

struct stopwatch {
  int min;
  int sec;
};
// requires: 0 <= min
//           0 <= sec <= 59

int stopwatch_get_seconds(const struct stopwatch *sw) {
  assert(sw);
  return sw->sec;
}

int stopwatch_get_minutes(const struct stopwatch *sw) {
  assert(sw);
  return sw->min;
}
```

The only non-trivial function is the `stopwatch_add_time` function.

```
// stopwatch.c [IMPLEMENTATION]

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  assert(sw);
  assert(sec >= 0);
  assert(min >= 0);
  sw->min += min;
  sw->sec += sec;
  while (sw->sec >= 60) {
    sw->min += 1;
    sw->sec -= 60;
  }
}
```

This simple client illustrates the use of the `stopwatch` module.

```
// client.c

#include "cs136.h"
#include "stopwatch.h"

int main(void) {
  struct stopwatch *sw = stopwatch_create();

  stopwatch_add_time(sw, 1, 59);
  stopwatch_add_time(sw, 3, 30);

  trace_int(stopwatch_get_minutes(sw));
  trace_int(stopwatch_get_seconds(sw));

  stopwatch_destroy(sw);
}
```

```
stopwatch_get_minutes(sw) => 5
stopwatch_get_seconds(sw) => 29
```

Maintainability

We can improve our `stopwatch` module without changing the client. This is an example of *maintainability*.

```c
// stopwatch.c [IMPLEMENTATION]

static const int seconds_per_minute = 60;

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  assert(sw);
  assert(sec >= 0);
  assert(min >= 0);
  const int total_time = (min + sw->min) * seconds_per_minute +
                         sec + sw->sec;
  sw->min = total_time / seconds_per_minute;
  sw->sec = total_time % seconds_per_minute;
}
```

Flexibility

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

We can change the design of our `struct stopwatch` and the client will be completely unaware.

The client only accesses our structure through the provided functions. As long as we do not change the function behaviour, we have the *flexibility* to change our design.

For example, we can change our `struct stopwatch` to contain a single field (`seconds`) instead of storing the information in two fields (`sec` and `min`).

```c
// stopwatch.c [IMPLEMENTATION] (asserts removed for conciseness)

struct stopwatch {
  int seconds;
};
// requires: 0 <= seconds

static const int seconds_per_minute = 60;

int stopwatch_get_seconds(const struct stopwatch *sw) {
  return sw->seconds % seconds_per_minute;
}

int stopwatch_get_minutes(const struct stopwatch *sw) {
  return sw->seconds / seconds_per_minute;
}

void stopwatch_add_time(struct stopwatch *sw, int min, int sec) {
  sw->seconds += min * seconds_per_minute + sec;
}
```

Data structures & abstract data types

In the previous `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*).

Data structures *vs.* ADTs

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**, 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.

Collection ADTs

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.

In CS 135 we were introduced to our first *collection ADT*: a **dictionary**.

In most contexts, when someone refers to an ADT they *implicitly* mean a “collection ADT”.

By some definitions, collection ADTs are the *only* type of ADT.

Dictionary (revisited)

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:

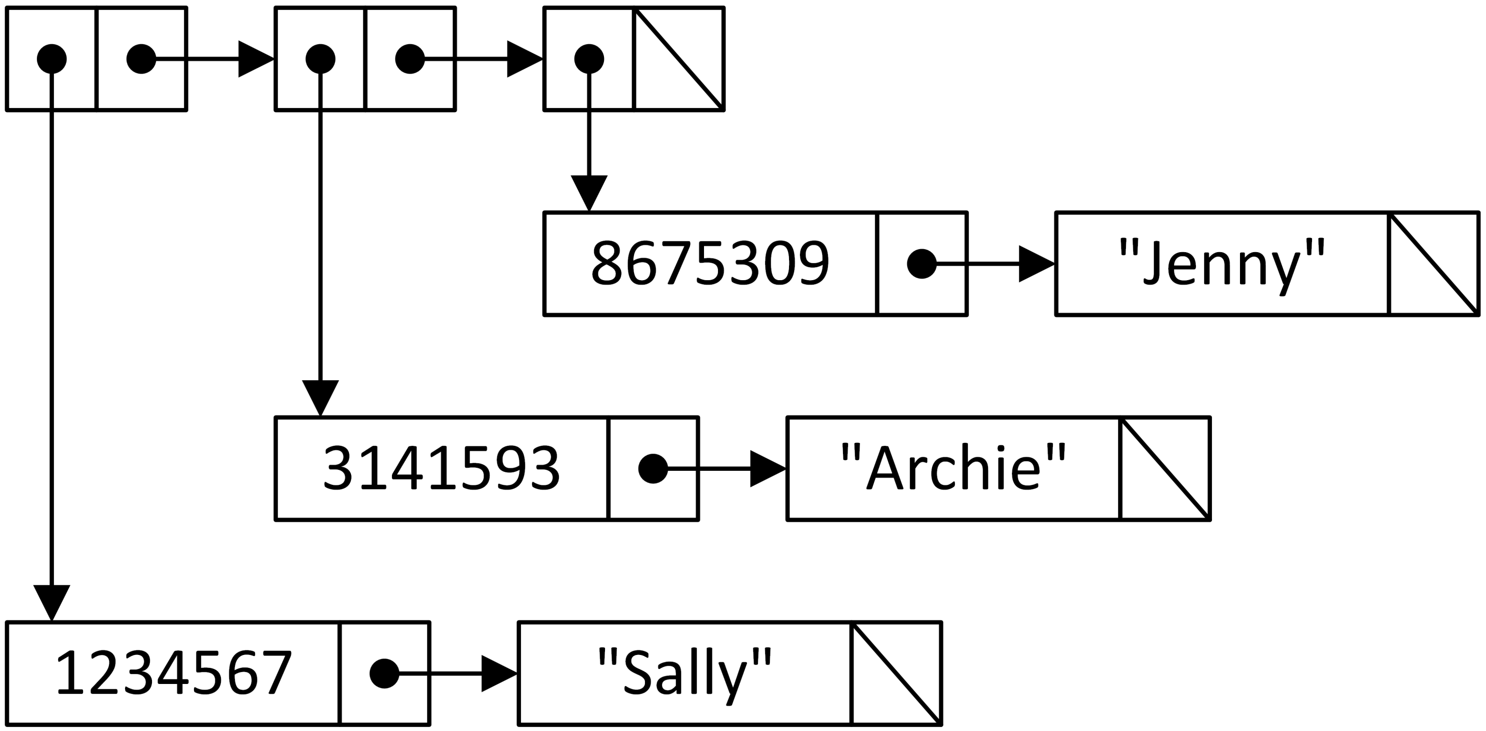
- **lookup:** for a given key, retrieve the corresponding value or “not found”
- **insert:** adds a new key/value pair (or replaces the value of an existing key)
- **remove:** *deletes* a key and its value

example: student numbers

key (student number)	value (student name)
`1234567`	`"Sally"`
`3141593`	`"Archie"`
`8675309`	`"Jenny"`

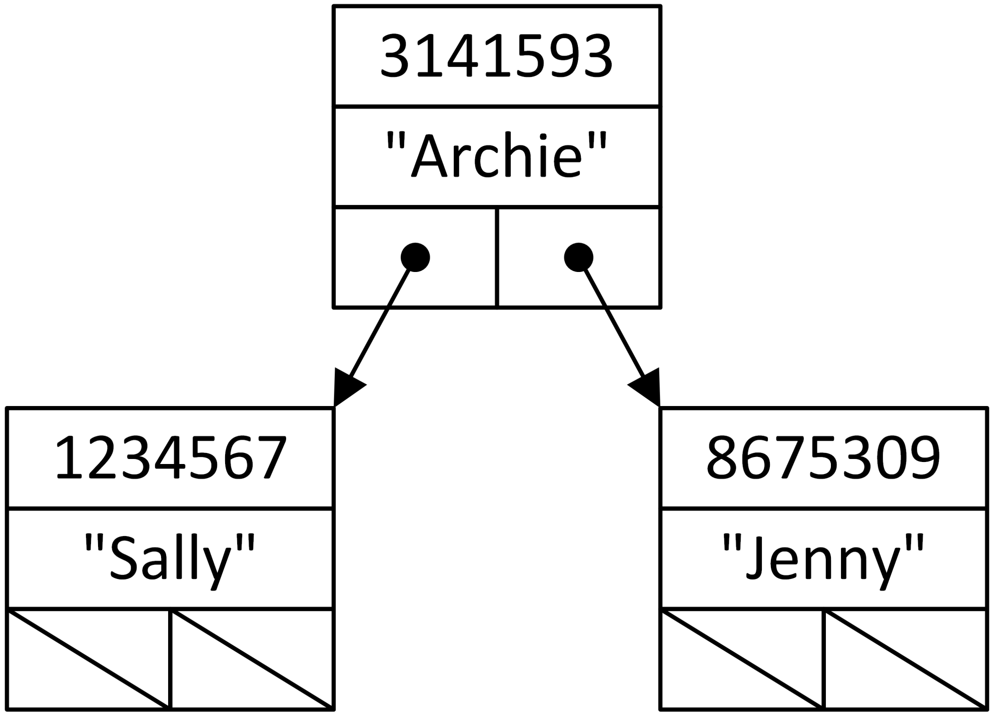
In CS 135 we implemented a dictionary with an ***association list data structure*** (a list of key/value pairs with each pair stored as a two-element list).

```
(define al '((1234567 "Sally") (3141593 "Archie")
             (8675309 "Jenny")))
```



We also implemented a dictionary with a ***Binary Search Tree (BST)*** *data structure*.

```
(define bst (make-node 3141593 "Archie"
                       (make-node 1234567 "Sally" empty empty)
                       (make-node 8675309 "Jenny" empty empty)))
```



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.

You likely have an intuition that BSTs are “more efficient” than association lists. In Section 08 we introduce a formal notation to describe the efficiency of an implementation.

More collection ADTs

Three additional collection ADTs that are explored in this course:

- stack
- queue
- sequence

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

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

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.

Goals of this Section

At the end of this section, you should be able to:

- explain and demonstrate the three core advantages of modular design: abstraction, re-usability and maintainability
- identify two characteristics of a good modular interface: high cohesion and low coupling
- explain and demonstrate information hiding and how it supports both security and flexibility
- explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.

- explain the difference between a declaration and a definition
- explain the differences between local, module and program scope and demonstrate how `static` and `extern` are used
- write modules in C with implementation and interface files
- write good interface documentation
- use a module as a client (including writing a test client)