Chapter 19

Program Design

Introduction

- Most full-featured programs are at least 100,000 lines long.
- Although C wasn't designed for writing large programs, many large programs have been written in C.
- Writing large programs is quite different from writing small ones.

Introduction

- Issues that arise when writing a large program:
  - Style
  - Documentation
  - Maintenance
  - Design
- This chapter focuses on design techniques that can make C programs readable and maintainable.

Modules

- It's often useful to view a program as a number of independent ***modules.***
- A module is a collection of services, some of which are made available to other parts of the program (the ***clients***).
- Each module has an ***interface*** that describes the available services.
- The details of the module—including the source code for the services themselves—are stored in the module's ***implementation.***

Modules

- In the context of C, “services” are functions.
- The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files).
- The implementation of a module is a source file that contains definitions of the module’s functions.

Modules

- The calculator program sketched in Chapter 15 consists of:
  - `calc.c`, which contains the `main` function
  - A stack module, stored in `stack.h` and `stack.c`
- `calc.c` is a *client* of the stack module.
- `stack.h` is the *interface* of the stack module.
- `stack.c` is the *implementation* of the module.

```c
#include <stdbool.h>

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);
```

stack.h

```c
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

calc.c

```c
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

bool is_empty(void)
{ … }

bool is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

stack.c

Modules

- The C library is itself a collection of modules.
- Each header in the library serves as the interface to a module.
  - `<stdio.h>` is the interface to a module containing I/O functions.
  - `<string.h>` is the interface to a module containing string-handling functions.

Modules

- Advantages of dividing a program into modules:
  - Abstraction
  - Reusability
  - Maintainability

Modules

- ***Abstraction.*** A properly designed module can be treated as an ***abstraction;*** we know what it does, but we don't worry about how it works.
- Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it.
- Abstraction also makes it easier for several members of a team to work on the same program.

Modules

- ***Reusability.*** Any module that provides services is potentially reusable in other programs.
- Since it's often hard to anticipate the future uses of a module, it's a good idea to design modules for reusability.

Modules

- ***Maintainability.*** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix.
- Rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program).
- An entire module implementation can be replaced if necessary.

Modules

- Maintainability is the most critical advantage.
- Most real-world programs are in service over a period of years
- During this period, bugs are discovered, enhancements are made, and modifications are made to meet changing requirements.
- Designing a program in a modular fashion makes maintenance much easier.

Modules

- Decisions to be made during modular design:
  - What modules should a program have?
  - What services should each module provide?
  - How should the modules be interrelated?

Cohesion and Coupling

- In a well-designed program, modules should have two properties.
- ***High cohesion.*** The elements of each module should be closely related to one another.
  - High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible.
  - Low coupling makes it easier to modify the program and reuse modules.

Types of Modules

- Modules tend to fall into certain categories:
  - Data pools
  - Libraries
  - Abstract objects
  - Abstract data types

Types of Modules

- A ***data pool*** is a collection of related variables and/or constants.
  - In C, a module of this type is often just a header file.
  - `<float.h>` and `<limits.h>` are both data pools.
- A ***library*** is a collection of related functions.
  - `<string.h>` is the interface to a library of string-handling functions.

Types of Modules

- An ***abstract object*** is a collection of functions that operate on a hidden data structure.
- An ***abstract data type (ADT)*** is a type whose representation is hidden.
  - Client modules can use the type to declare variables but have no knowledge of the structure of those variables.
  - To perform an operation on such a variable, a client must call a function provided by the ADT.

Information Hiding

- A well-designed module often keeps some information secret from its clients.
  - Clients of the stack module have no need to know whether the stack is stored in an array, in a linked list, or in some other form.
- Deliberately concealing information from the clients of a module is known as ***information hiding*.**

Information Hiding

- Primary advantages of information hiding:
  - ***Security.*** If clients don't know how a module stores its data, they won't be able to corrupt it by tampering with its internal workings.
  - ***Flexibility.*** Making changes—no matter how large—to a module's internals won't be difficult.

Information Hiding

- In C, the major tool for enforcing information hiding is the `static` storage class.
  - A `static` variable with file scope has internal linkage, preventing it from being accessed from other files, including clients of the module.
  - A `static` function can be directly called only by other functions in the same file.

A Stack Module

- To see the benefits of information hiding, let's look at two implementations of a stack module, one using an array and the other a linked list.
- `stack.h` is the module's header file.
- `stack1.c` uses an array to implement the stack.

stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>   /* C99 only */

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);

#endif
```

stack1.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  top = 0;
}
```

```
bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    terminate("Error in push: stack is full.");
  contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    terminate("Error in pop: stack is empty.");
  return contents[--top];
}
```

A Stack Module

- Macros can be used to indicate whether a function or variable is "public" (accessible elsewhere in the program) or "private" (limited to a single file):

```
#define PUBLIC  /* empty */
#define PRIVATE static
```

- The word `static` has more than one use in C; `PRIVATE` makes it clear that we're using it to enforce information hiding.

A Stack Module

- The stack implementation redone using `PUBLIC` and `PRIVATE`:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { … }

PUBLIC void make_empty(void) { … }

PUBLIC bool is_empty(void) { … }

PUBLIC bool is_full(void) { … }

PUBLIC void push(int i) { … }

PUBLIC int pop(void) { … }
```

A Stack Module

- `stack2.c` is a linked-list implementation of the stack module.

stack2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  while (!is_empty())
    pop();
}
```

```
bool is_empty(void)
{
  return top == NULL;
}

bool is_full(void)
{
  return false;
}

void push(int i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}
```

```
int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty())
    terminate("Error in pop: stack is empty.");

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```

A Stack Module

- Thanks to information hiding, it doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module.
- Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

Abstract Data Types

- A module that serves as an abstract object has a serious disadvantage: there's no way to have multiple instances of the object.
- To accomplish this, we'll need to create a new *type*.
- For example, a `Stack` type can be used to create any number of stacks.

Abstract Data Types

- A program fragment that uses two stacks:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));  /* prints "1" */
```

- To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Abstract Data Types

- Converting the `stack.h` header so that it provides a `Stack` type requires adding a `Stack` (or `Stack *`) parameter to each function.

Abstract Data Types

- Changes to `stack.h` are shown in **bold**:

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

Abstract Data Types

- The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack.
- The parameter to `is_empty` and `is_full` doesn't need to be a pointer.
- Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is done for efficiency, since the latter would result in a structure being copied.

Encapsulation

- Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is.
- Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

- Providing access to the `top` and `contents` members allows clients to corrupt the stack.

Encapsulation

- Worse still, we can't change the way stacks are stored without having to assess the effect of the change on clients.
- What we need is a way to prevent clients from knowing how the `Stack` type is represented.
- C has only limited support for ***encapsulating*** types in this way.
- Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

Incomplete Types

- The only tool that C gives us for encapsulation is the ***incomplete type.***
- Incomplete types are "types that describe objects but lack information needed to determine their sizes."
- Example:

```
struct t;  /* incomplete declaration of t */
```

- The intent is that an incomplete type will be completed elsewhere in the program.

Incomplete Types

- An incomplete type can't be used to declare a variable:

```
struct t s;    /*** WRONG ***/
```

- However, it's legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

- We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers.

A Stack Abstract Data Type

- The following stack ADT will illustrate how abstract data types can be encapsulated using incomplete types.
- The stack will be implemented in three different ways.

Defining the Interface for the Stack ADT

- `stackADT.h` defines the stack ADT type and gives prototypes for the functions that represent stack operations.
- The `Stack` type will be a pointer to a `stack_type` structure (an incomplete type).
- The members of this structure will depend on how the stack is implemented.

stackADT.h
(version 1)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Defining the Interface for the Stack ADT

- Clients that include `stackADT.h` will be able to declare variables of type `Stack`, each of which is capable of pointing to a `stack_type` structure.
- Clients can then call the functions declared in `stackADT.h` to perform operations on stack variables.
- However, clients can't access the members of the `stack_type` structure, since that structure will be defined in a separate file.

Defining the Interface for the Stack ADT

- A module generally doesn't need `create` and `destroy` functions, but an ADT does.
  - `create` dynamically allocates memory for a stack and initializes the stack to its "empty" state.
  - `destroy` releases the stack's dynamically allocated memory.

Defining the Interface for the Stack ADT

- `stackclient.c` can be used to test the stack ADT.
- It creates two stacks and performs a variety of operations on them.

stackclient.c

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

int main(void)
{
  Stack s1, s2;
  int n;

  s1 = create();
  s2 = create();

  push(s1, 1);
  push(s1, 2);

  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);
```

```
  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);

  destroy(s1);

  while (!is_empty(s2))
    printf("Popped %d from s2\n", pop(s2));

  push(s2, 3);
  make_empty(s2);
  if (is_empty(s2))
    printf("s2 is empty\n");
  else
    printf("s2 is not empty\n");

  destroy(s2);

  return 0;
}
```

Defining the Interface for the Stack ADT

- Output if the stack ADT is implemented correctly:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```

Implementing the Stack ADT Using a Fixed-Length Array

- There are several ways to implement the stack ADT.
- The simplest is to have the `stack_type` structure contain a fixed-length array:

```
struct stack_type {
  int contents[STACK_SIZE];
  int top;
};
```

stackADT.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100

struct stack_type {
  int contents[STACK_SIZE];
  int top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = 0;
  return s;
}

void destroy(Stack s)
{
  free(s);
}

void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}
```

```
bool is_full(Stack s)
{
  return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

Changing the Item Type in the Stack ADT

- `stackADT.c` requires that stack items be integers, which is too restrictive.
- To make the stack ADT easier to modify for different item types, let's add a type definition to the `stackADT.h` header.
- It will define a type named `Item`, representing the type of data to be stored on the stack.

stackADT.h
(version 2)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

Changing the Item Type in the Stack ADT

- The `stackADT.c` file will need to be modified, but the changes are minimal.
- The updated `stack_type` structure:

```
struct stack_type {
  Item contents[STACK_SIZE];
  int top;
};
```

- The second parameter of `push` will now have type `Item`.
- `pop` now returns a value of type `Item`.

Changing the Item Type in the Stack ADT

- The `stackclient.c` file can be used to test the new `stackADT.h` and `stackADT.c` to verify that the `Stack` type still works.
- The item type can be changed by modifying the definition of `Item` in `stackADT.h`.

Implementing the Stack ADT Using a Dynamic Array

- Another problem with the stack ADT: each stack has a fixed maximum size.
- There's no way to have stacks with different capacities or to set the stack size as the program is running.
- Possible solutions to this problem:
  - Implement the stack as a linked list.
  - Store stack items in a dynamically allocated array.

Implementing the Stack ADT Using a Dynamic Array

- The latter approach involves modifying the `stack_type` structure.
- The `contents` member becomes a *pointer* to the array in which the items are stored:

```
struct stack_type {
  Item *contents;
  int top;
  int size;
};
```

- The `size` member stores the stack's maximum size.

Implementing the Stack ADT Using a Dynamic Array

- The `create` function will now have a parameter that specifies the desired maximum stack size:

```
Stack create(int size);
```

- When `create` is called, it will create a `stack_type` structure plus an array of length `size`.
- The `contents` member of the structure will point to this array.

Implementing the Stack ADT Using a Dynamic Array

- `stackADT.h` will be the same as before, except that `create` will have a `size` parameter.
- The new version will be named `stackADT2.h`.
- `stackADT.c` will need more extensive modification, yielding `stackADT2.c`.

stackADT2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
  Item *contents;
  int top;
  int size;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  s->size = size;
  return s;
}

void destroy(Stack s)
{
  free(s->contents);
  free(s);
}
```

```
void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}

bool is_full(Stack s)
{
  return s->top == s->size;
}
```

```
void push(Stack s, Item i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

Item pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

Implementing the Stack ADT Using a Dynamic Array

- The `stackclient.c` file can again be used to test the stack ADT.
- The calls of `create` will need to be changed, since `create` now requires an argument.
- Example:

```
s1 = create(100);
s2 = create(200);
```

Implementing the Stack ADT Using a Linked List

- Implementing the stack ADT using a dynamically allocated array provides more flexibility than using a fixed-size array.
- However, the client is still required to specify a maximum size for a stack at the time it's created.
- With a linked-list implementation, there won't be any preset limit on the size of a stack.

Implementing the Stack ADT Using a Linked List

- The linked list will consist of nodes, represented by the following structure:

```
struct node {
  Item data;
  struct node *next;
};
```

- The `stack_type` structure will contain a pointer to the first node in the list:

```
struct stack_type {
  struct node *top;
};
```

Implementing the Stack ADT Using a Linked List

- The `stack_type` structure seems superfluous, since `Stack` could be defined to be `struct node *`.
- However, `stack_type` is needed so that the interface to the stack remains unchanged.
- Moreover, having the `stack_type` structure will make it easier to change the implementation in the future.

Implementing the Stack ADT Using a Linked List

- Implementing the stack ADT using a linked list involves modifying the `stackADT.c` file to create a new version named `stackADT3.c`.
- The `stackADT.h` header is unchanged.
- The original `stackclient.c` file can be used for testing.

stackADT3.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}

void destroy(Stack s)
{
  make_empty(s);
  free(s);
}

void make_empty(Stack s)
{
  while (!is_empty(s))
    pop(s);
}
```

```
bool is_empty(Stack s)
{
  return s->top == NULL;
}

bool is_full(Stack s)
{
  return false;
}

void push(Stack s, Item i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = s->top;
  s->top = new_node;
}
```

```
Item pop(Stack s)
{
  struct node *old_top;
  Item i;

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

  old_top = s->top;
  i = old_top->data;
  s->top = old_top->next;
  free(old_top);
  return i;
}
```

Design Issues for Abstract Data Types

- The stack ADT suffers from several problems that prevent it from being industrial-strength.

Naming Conventions

- The stack ADT functions currently have short, easy-to-understand names, such as `create`.
- If a program has more than one ADT, name clashes are likely.
- It will probably be necessary for function names to incorporate the ADT name (`stack_create`).

Error Handling

- The stack ADT deals with errors by displaying an error message and terminating the program.
- It might be better to provide a way for a program to recover from errors rather than terminating.
- An alternative is to have the `push` and `pop` functions return a `bool` value to indicate whether or not they succeeded.

Error Handling

- The C standard library contains a parameterized macro named `assert` that can terminate a program if a specified condition isn't satisfied.
- We could use calls of this macro as replacements for the `if` statements and calls of `terminate` that currently appear in the stack ADT.

Generic ADTs

- Other problems with the stack ADT:
  - Changing the type of items stored in a stack requires modifying the definition of the `Item` type.
  - A program can’t create two stacks whose items have different types.
- We’d like to have a single “generic” stack type.
- There’s no completely satisfactory way to create such a type in C.

Generic ADTs

- The most common approach uses `void *` as the item type:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

  `pop` returns a null pointer if the stack is empty.

- Disadvantages of using `void *` as the item type:
  - Doesn't work for data that can't be represented in pointer form, including basic types such as `int` and `double`.
  - Error checking is no longer possible, because stack items can be a mixture of pointers of different types.

ADTs in Newer Languages

- These problems are dealt with much more cleanly in newer C-based languages.
  - Name clashes are prevented by defining function names within a ***class.***
  - ***Exception handling*** allows functions to "throw" an exception when they detect an error condition.
  - Some languages provide special features for defining generic ADTs. (C++ ***templates*** are an example.)