Lecture 19
Introduction to C

15-122: Principles of Imperative Computation (Spring 2024)
Rob Simmons

In this lecture, we begin our transition to C. In many ways, the lecture is therefore about knowledge rather than principles, a return to the emphasis on programming that we had at the very beginning of the semester. In future lectures, we will explore some deeper issues in the context of C. Today's lecture is designed to get you to the point where you can translate a simple C0/C1 program or library (one that doesn't use arrays, which we'll talk about in the next lecture) from C0/C1 to C. An important complement to this lecture is the "C for C0 programmers" tutorial at `https://bitbucket.org/c0-lang/docs/wiki/From_C0_to_C_-_Basics`.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/handouts/slides/review/19-cintro.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/19-cintro.tgz`)
- There are two short videos associated with this lecture:
  - Freeing in C (`https://youtu.be/J_FzhQAWiJ4`)
  - Using Valgrind (`https://youtu.be/2e_u2eXe7P4`)

There are two big ideas you need to know about. First, C has a whole separate language wrapped around it, the *C preprocessor language*. The preprocessor language can be used for a bunch of things: you only need to understand a couple of ways that it gets used:

- *Macro constant definitions*: you'll need to know how these are used in the `<limits.h>` and `<stdbool.h>` libraries.
- *Macro function definitions*: you'll need to know how these are used to implement the `"lib/contracts.h"` library, and you'll need to know why they're generally a dangerous idea.

 © Carnegie Mellon University 2024

- *Conditional compilation*: you need to know how `#ifdef` and `#ifndef` are used, along with macro constant definitions, to make *separate compilation* of libraries work in C.

Second, C has a different notion of allocating memory than C0. In particular, C is not garbage collected, so whenever we allocate memory, we have to make sure that memory eventually gets *freed*.

1 Running Example

Our discussion will center around translating a very simple C0 interface and implementation, and a little program that uses that interface.

1.1 A simple interface `simple.c0`

```
#use <util>

/*** Interface ***/
int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0; @*/ ;

struct point2d {
  int x;
  int y;
};

/*** Implementation ***/
int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  int res = x < 0 ? -x : x;
  return res;
}
```

1.2 A simple test program: `test.c0`

```
#use <conio>
int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); println("\n");
  return 0;
}
```

We can compile this program by running: `cc0 -d simple.c0 test.c0`

2 Introducing the Preprocessor Language

In C0 programs, just about the only time we typed the '**#**' key was to include a built-in library like `conio` by writing: `#use <conio>`. The C preprocessor language is built around different directives that all start with '**#**'. The first two you need to know about are **`#include`** and **`#define`**.

The **`#include`** directive is what replaces `#use` in C0. Here are some common **`#include`** directives you'll see in C programs:

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

The `<stdlib.h>` library is related to C0's `<util>` library, `<stdio.h>` is related to `<conio>` in C0, and `<string.h>` is related to `<string>` in C0.

The `<stdbool.h>` file is also important: the type **`bool`** and the constants `true` and `false` aren't automatically included in C, so this library includes them. We'll talk more about libraries, and in particular the `.h` extension, later.

3 Macro Definitions

C0 has a very simple rule: an interface can describe types, structs, and functions. This leads to some weirdnesses, though: the C0 `<util>` library has to give you a *function*, `int_max()`, for referring to the maximum representable 32-bit two's complement integer.

The **`#define`** macro gives you a way to define this as a *constant* in C.

```
#define INT_MAX 0x7FFFFFFF
```

In C, the directives of the preprocessor language are used by a *preprocessor*, a component that gets executed *before* the C compiler. The preprocessor does a textual replacement of all macro definitions with the expression they are defined as. So, whenever the preprocessor sees `INT_MAX` in your program, it replaces it with `0x7FFFFFFF`. The C compiler itself will never see `INT_MAX`.

This textual replacement must be done very carefully: for instance, this is a valid, if needlessly verbose, definition of `INT_MIN`:

```
#define INT_MIN -1 ^ 0x7FFFFFFF
```

Then imagine that later in the program we wrote `INT_MIN / 256`, which ought to be equal to $-2^{31}/2^8 = -2^{23} = -16777216$. This would get expanded by the C preprocessor language to `-1 ^ 0x7FFFFFFF / 256`, which

the compiler would happily treat as `-1 ^ (0x7FFFFFFF / 256)`, which is -8388608. The problem is that the preprocessor doesn't know or care about the order of operations in C: it's just blindly substituting text. Parentheses would fix this particular problem:

```
#define INT_MIN (-1 ^ 0x7FFFFFFF)
```

The best idea is to use **`#define`** sparingly and mostly get your macro definitions from standard libraries. The definitions `INT_MIN` and `INT_MAX` are already provided by the standard C library `<limits.h>`.

4 Conditional Compilation

Another very powerful but very-easy-to-get-wrong feature of the macro language is *conditional compilation*. Based on whether a symbol is defined or not, the preprocessor can choose to ignore a whole section of text or choose between separate sections of text. This is used in a couple of different ways. Sometimes we use **`#ifndef`** (if *not* defined) to make sure we're not defining something twice:

```
#ifndef INT_MIN
#define INT_MIN (~0x7FFFFFFF)
#endif
```

We can also use **`#ifdef`** and **`#else`** to pick between different pieces of code to define. The code below is very different from C0/C code with a condition **`if`** `(version_one)` statement, because only one of the two print statements below will ever even get compiled. The other one will be cut out of the program by the preprocessor before the compiler even sees it!

```
#ifdef VERSION_ONE
printf("This is version 1\n");
#else
printf("This is not version 1\n");
#endif
```

One interesting thing about this example is that we don't care what `VERSION_ONE` is defined to be: we're just using the information about whether it is defined or not. We'll use the `DEBUG` symbol in some of our C programs to include certain pieces of code only when `DEBUG` is defined.

```
#ifdef DEBUG
printf("Some helpful debugging information\n");
#endif
```

5 Macro Functions

A more powerful version of macro definition is the *macro function*. For example:

```
#define MULT(x,y) ((x)*(y))
```

Using parentheses defensively is very important here, because otherwise the precedence issues we described before will only get worse. The only place we'll use macro functions in 15-122 is to define something like C0 contracts in C. The macro functions `ASSERT`, `REQUIRES`, and `ENSURES` turn into assertions when the `DEBUG` symbol is present, but otherwise they are replaced by `((void)0)`, which just tells the compiler to do nothing at all.

```
#ifndef DEBUG

#define ASSERT(COND) ((void)0)
#define REQUIRES(COND) ((void)0)
#define ENSURES(COND) ((void)0)

#else

#define ASSERT(COND) assert(COND)
#define REQUIRES(COND) assert(COND)
#define ENSURES(COND) assert(COND)

#endif
```

The code above isn't something you have to write yourself: it's provided for you in the file `contracts.h` that will be in the `lib` directory of all of our C projects in 15-122. Therefore, we write:

```
#include "lib/contracts.h"
```

in order to include these macro-defined contracts in our programs. When we use quotes instead of angle brackets for **`#include`**, as we do here, it just means that we're looking for a library we wrote ourselves and are using locally, not a standard library that we expect the compiler will find wherever it stores its standard library interfaces.

6 C0 Contracts in C

There's no assertion language in C: everything starting with `//@` and everything written inside `/*@... @*/` is just a treated as a comment and ignored.

We'll still write C0-style contracts in our interfaces, but those contracts are now just comments, good for documentation, but not for runtime checking.

All contracts, including preconditions and postconditions, have to be written inside of the function if we want them to be checked at runtime.

```
int absval(int x) {
  REQUIRES(x > INT_MIN);
  int res = x < 0 ? -x : x;
  ENSURES(res >= 0);
  return res;
}
```

There's not a good replacement for loop invariants in C; they just have to be replaced with careful uses of `ASSERT`.

7 Memory Allocation

In C0, we allocate pointers of a particular *type*; in C, we allocate pointers of a particular *size*: the operator **`sizeof`** takes a type and returns the number of bytes in this type, and it is this size that we pass to the allocation function. The default way of allocating a struct or integer (or similar) in C is to use the function `malloc`, provided in the standard `<stdlib.h>` library.

```
C0: int* x = alloc(int);
C:  int* x = malloc(sizeof(int));
```

One quirk with `malloc` is that it *does not initialize memory*, so dereferencing `x` before storing some integer into `x` could return an arbitrary value. (The computer is able to allocate memory slightly more efficiently if it doesn't have to initialize that memory.) This is *different* from C0, where allocated memory was always initialized to a default value: `NULL` for pointers, 0 for integers, `""` for strings, and so on.

Another quirk with `malloc` is that it is allowed to return `NULL`. Ultimately there is only a finite amount of memory accessible to the computer, and `malloc` will return `NULL` when there is no memory left to allocate. Therefore, we will usually use a 15-122 library `"lib/xalloc.h"`, which provides the function `xmalloc`. The `xmalloc` function provided by this library works the same way `malloc` does, except that the result is sure not to be `NULL`.

```
C:  int* x = xmalloc(sizeof(int)); // x is definitely not NULL
```

By replacing **`alloc`** with `xmalloc` and **`sizeof`**, we can now translate our `test.c0` file into `test.c`. The series of print statements has been replaced by a single function `printf`.

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "lib/xalloc.h"

int main() {
    struct point2d* P = xmalloc(sizeof(struct point2d));
    P->x = -15;
    P->y = 0;
    P->y = P->y + absval(P->x * 2);
    assert(P->y > P->x && true);
    printf("x coord: %d\n", P->x);
    return 0;
}
```

We needed an extra line, `P->y = 0;`, that wasn't present in the original file to cope with the fact that the `malloc`-ed `y` field isn't initialized to 0 the way it was in C0.

8 Compiling

Our code won't actually compile yet, but we can try to compile it now that we've translated both `simple.c` and `test.c`. When we call `gcc`, the C compiler, we'll give it a long series of flags:

```
% gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG ...
```

The flags `-Wall`, `-Wextra`, and `-Wshadow` represent a bunch of optional compilation Warnings we want to get from the compiler, and `-Werror` means that if we get any warnings the code should not be compiled. The flag `-std=c99` means that the version of C we are using is the one that was written down as the C99 standard, a standard we want to adhere to in a `-pedantic` way.

The flag `-g` keeps information in the compiled program which will be helpful for the `valgrind` utility tool (see below after the discussion of `free`). The flag `-DDEBUG` means that we want the preprocessor to run with the `DEBUG` symbol Defined. As we talked about before, this means that contracts will actually be checked at runtime: `-DDEBUG` is the C version of the `-d` flag for the C0 compiler and interpreter.

9 Separate Compilation

If we try to compile the translated C files we have so far, it won't work:

```
% gcc ...all those flags... lib/*.c simple.c test.c
test.c: In function "main":
test.c:8:38: error: invalid application of sizeof to incomplete type...
   struct point2d* P = xmalloc(sizeof(struct point2d));
                                      ^
test.c:10:3: error: implicit declaration of function absval...
   P->y = P->y + absval(P->x * 2);
   ^
```

If compiling C worked like compiling C0, `test.c` would be able to see the interface from `simple.c`, which includes the definition of **`struct`** `point2d` and the type of `absval`, because `simple.c` came ahead of `test.c` on the command line. However, C doesn't work this way: *every C file is compiled separately from all the other C files*.

To get our code to compile, we want to split up the `simple.c` file into two parts: the interface, which will go in the header file `simple.h`, and the implementation, which will stay in `simple.c` and will **`#include`** the interface `"simple.h"`. Then, we can also **`#include`** the simple interface in `test.c`.

This is actually a good thing from the perspective of respecting the interface: `test.c` will have access to the interface in `simple.h`, but couldn't accidentally end up relying on extra things defined in `simple.c`.

9.1 Interface: `simple.h`

In addition to containing the interface from `simple.c0`, the header file containing the `simple.h` interface, like all C header files, needs to use **`#ifndef`**, **`#define`**, and **`#endif`**. These three preprocessor declarations, in combination, make sure that we can only end up including this code one time, even if we intentionally or accidentally write **`#include`** `"simple.h"` more than once.

```
#ifndef SIMPLE_H
#define SIMPLE_H

int absval(int x)
/*@requires x >= INT_MIN; @*/
/*@ensures \result >= 0; @*/ ;

struct point2d {
    int x;
    int y;
};

#endif
```

9.2 Implementation: `simple.c`

The C file will include both the necessary libraries and the interface. *The implementation should always* **`#include`** *the interface.*

```
#include <limits.h>
#include "lib/contracts.h"
#include "simple.h"

int absval(int x) {
    REQUIRES(x > INT_MIN);
    int res = x < 0 ? -x : x;
    ENSURES(res >= 0);
    return res;
}
```

9.3 Main file: `test.c`

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "lib/xalloc.h"
#include "simple.h"

int main() {
    struct point2d* P = xmalloc(sizeof(struct point2d));
    P->x = -15;
    P->y = 0;
    P->y = P->y + absval(P->x * 2);
    assert(P->y > P->x && true);
    printf("x coord: %d\n", P->x);
    return 0;
}
```

At this point, compilation will proceed without errors.

10 Memory Leaks

See the short video on Freeing in C at https://www.youtube.com/embed/J_FzhQAWiJ4.

Unlike C0, C does not automatically manage memory. Thus, programs have to free the memory they allocate explicitly; otherwise, long-running or memory-intensive programs are likely to run out of space. For that, the C standard library provides the function `free`, declared with

```
void free(void* p);
```

The restrictions as to its proper use are

1. It is only called on pointers that were returned from `malloc` or `calloc` (possibly indirectly via the `xalloc` library).[1]
2. After memory has been freed, it is no longer referenced by the program in any way.

Freeing memory counts as referencing it, so the restrictions imply that you should not free memory twice. And, indeed, in C the behavior of freeing memory that has already been freed is undefined and may be exploited

[1] or `realloc`, which we have not discussed.

by an adversary. If these rules are violated, the result of the operations is undefined. The `valgrind` tool will catch dynamically occurring violations of these rules, but it cannot check statically if your code will respect these rules when executed.

Managing memory in your C programs means walking the narrow way between two pitfalls: all allocated memory should be freed after it is no longer used, but no allocated memory should be referenced after it is freed! Falling into the first pit causes *memory leaks*, which cause long-running programs to run out of unallocated memory. Falling into the second one causes undefined, i.e. unpredictable, behavior.

The *golden rule of memory management* in C is

> *You allocate it, you free it!*

By inference, if you *didn't* allocate it, you are *not* allowed to free it! But this rule is tricky in practice, because sometimes we do need to transfer ownership of allocated memory so that it "belongs" to a data structure.

Binary search trees are one example. When client code adds an element to the binary search tree, is it in charge of freeing that element, or should the library code free it when it frees the binary search tree? There are arguments to be made for both of these options. If we want the library code for the BST to "own" the reference, and therefore be in charge of freeing it, we can write the following function that frees a binary search tree, given a function pointer that frees elements. The library can allow this function pointer to be `NULL`: if it's `NULL` the library code doesn't own the elements, and doesn't do anything to them. We also show the function that frees a dictionary implemented as a binary search tree.

```
typedef void entry_free_fn(entry e);

void tree_free(tree *T, entry_free_fn *Fr) {
  REQUIRES(is_bst(T));
  if (T != NULL) {
    if (Fr != NULL) (*Fr)(T->data);
    tree_free(T->left, Fr);
    tree_free(T->right, Fr);
    free(T);
  }
  return;
}

void dict_free(dict *B, entry_free_fn *Fr) {
  REQUIRES(is_dict(B));
```

```
  tree_free(B->root, Fr);
  free(B);
  return;
}
```

We should never free elements allocated elsewhere; rather, we should use the appropriate function provided in the interface to free the memory associated with the data structure. Freeing a data structure, for instance by calling `free(T)`, is something the client itself cannot do reliably, because it would need to be privy to the internals of the data structure implementation. If the client called `free(B)` on a dictionary it would only free the header; the tree itself would be irrevocably leaked memory.

11 Detecting Memory Mismanagement

See the short video on Using Valgrind at https://www.youtube.com/embed/2e_u2eXe7P4.

Memory leaks can be quite difficult to detect by inspecting the code. To discover whether memory leaks may have occurred at runtime, we can use the `valgrind` tool.

For example, our `test.c` program that allocates but does not free memory, like this,

```
int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = 0;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  return 0;
}
```

gets a report from `valgrind` like this, indicating a memory leak:

```
% valgrind ./a.out
...
HEAP SUMMARY:
==40284==     in use at exit: 8 bytes in 1 blocks
==40284==   total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==40284==
==40284== LEAK SUMMARY:
==40284==    definitely lost: 8 bytes in 1 blocks
```

```
...
```

If we add code to free `P` just before the **`return`** statement, we get a clean bill of health from `valgrind`:

```
...
HEAP SUMMARY:
==41495==     in use at exit: 0 bytes in 0 blocks
==41495==   total heap usage: 1 allocs, 1 frees, 8 bytes allocated
==41495==
==41495== All heap blocks were freed --- no leaks are possible
...
```

If, on the other hand, we free `P` at the wrong point in our code, like this:

```
int main() {
    struct point2d* P = xmalloc(sizeof(struct point2d));
    ...
    free(P);
    printf("x coord: %d\n", P->x);
    return 0;
}
```

`valgrind` detects that we have referenced memory after freeing it (this is our second pitfall):

```
...
==43895== Invalid read of size 4
==43895==    at 0x400886: main (test.c:25)
==43895==  Address 0x51f6040 is 0 bytes inside a block of size 8 free'd
...
```

`valgrind` is capable of flagging errors in code that didn't appear to have any errors when run without `valgrind`. It slows down execution, but if at all feasible you should test all your C code in this manner to uncover memory problems. For best error messages, you should pass the `-g` flag to `gcc` which preserves some correlation between binary and source code.

12 Exercises

Exercise 1 (sample solution on page 20). *Translate the following C1 stack library into C, including any contracts, allocations, and the inclusion of any system libraries it needs. You may assume that the functions seen in class for C0-style contracts and safe allocation are declared in* `lib/contracts.h` *and* `lib/xalloc.h`*, respectively. Do not worry about freeing memory.*

```
/**********************************************************************/
/*************************** Client Interface *************************/

typedef void* stackelem;        // Element type

/************************ End Client Interface ************************/
/**********************************************************************/

/**********************************************************************/
/************************** BEGIN IMPLEMENTATION **********************/

typedef struct slist_node slist;
struct slist_node {               // structure of linked lists
  stackelem data;
  slist* next;
};


typedef struct stack_header stack;
struct stack_header {             // Stacks
  slist* top;
  slist* bottom;
};

bool stack_empty(stack* S)
//@requires S != NULL;
{
  return S->top == S->bottom;
}

stack* stack_new()
//@ensures \result != NULL;
//@ensures stack_empty(\result);
{
```

```
  stack* S = alloc(stack);
  slist* p = alloc(slist);
  S->top = p;
  S->bottom = p;
  return S;
}

void push(stack* S, stackelem e)
//@requires S != NULL;
//@ensures !stack_empty(S);
{
  slist* p = alloc(slist);
  p->data = e;
  p->next = S->top;
  S->top = p;
}

stackelem pop(stack* S)
//@requires S != NULL;
//@requires !stack_empty(S);
{
  stackelem e = S->top->data;
  S->top = S->top->next;
  return e;
}

typedef stack* stack_t;

/************************** END IMPLEMENTATION **************************/
/************************************************************************/

/************************************************************************/
/************************** Library Interface ***************************/

// typedef ______* stack_t;

bool stack_empty(stack_t S)          /* O(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()                  /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;
```

```
void push(stack_t S, stackelem x)    /* O(1) */
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

stackelem pop(stack_t S)             /* O(1) */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Exercise 2 (sample solution on page 23)**.** *Here is some client code that uses the C1 stack library in the previous exercise.*

```
#use <conio>

int main() {
  // Create a stack of ints
  stack_t S = stack_new();

  int* elem1 = alloc(int);
  *elem1 = 1;
  push(S, (void*)elem1);
  int* elem2 = alloc(int);
  *elem2 = 2;
  push(S, (void*)elem2);
  int* elem3 = alloc(int);
  *elem3 = 3;
  push(S, (void*)elem3);

  int i = 3;
  while (i > 1)
  //@loop_invariant 1 <= i && i <= 3;
  {
    int* elem = (int*)pop(S);
    assert(*elem == i);
    i--;
  }

  printf("All tests passed!\n");
  return 0;
}
```

Following the approach seen in this chapter, translate it to C. Also here, do not worry about freeing memory.

Exercise 3 (sample solution on page 24)**.** *Recall that everything that is* `malloc`*'ed must have a corresponding* `free`*. In the last exercise, the stack is never freed. Implement the function*

```
void stack_free(stack_t S, free_elem* f)
```

that allows a client to free a stack created by `stack_new` *when it is not needed any more. The last argument,* `f`*, can be either* `NULL` *to indicate that* `stack_free` ***shall not*** *free the data contained in the stack, or a pointer to a function that knows how to free the data.*

Besides implementing `stack_free`*, equip it with any contracts it may need, define the type* `free_elem`*, and describe the changes that need to be made to both the interface and the implementation files in the previous exercise.*

Because this code calls `free`, we will want to include the header file that defines it. That's `<stdlib.h>`.

Exercise 4 (sample solution on page 26)**.** *Update the client code you translated earlier to free allocated memory before returning.*

Exercise 5 (sample solution on page 27)**.** *The translated stack library and client code still leak memory. Run this code using Valgrind and determine where this leak comes from. (You may call Valgrind with the flag* `--leak-check=full` *to locate the source of the leak). Then fix it.*

Exercise 6 (sample solution on page 29)**.** *Write the C function* `stack_copy` *that takes in a stack and returns a copy of this stack using the interface functions in the previous exercises — is a client function. Then, write some test code that use* `stack_copy`*. For both parts, make sure to free any memory that would become unreachable.*

One thing we need to remember is to free the temporary stack `temp`. It is empty so we pass `NULL` as the second argument of `stack_free`. If we do not free this stack, its header will be leaked.

Exercise 7 (sample solution on page 29)**.** *Here's a main function that uses* `stack_copy`*:*

```
int main() {
  int* elem1 = xmalloc(sizeof(int));  // Create some elements
  *elem1 = 1;
  int* elem2 = xmalloc(sizeof(int));
  *elem2 = 2;
  int* elem3 = xmalloc(sizeof(int));
  *elem3 = 3;
```

```
  stack_t S = stack_new();            // Create a stack
  push(S, (void*)elem1);
  push(S, (void*)elem2);

  stack_t S_copy = stack_copy(S);     // Make a copy

  stack_free(S, &free);               // free the stack

  int* x = pop(S_copy);
  printf("%d\n", *x);
  push(S_copy, (void*)elem3);
  stack_free(S_copy, &free);          // free the copy

  printf("All tests passed!\n");
  return 0;
}
```

However, when we run it, it doesn't seem to work. Valgrind tells us that there is an invalid read. Valgrind also tells us that there is an invalid free. Why are these issues occurring and what simple fix we can make them go away?

Sample Solutions

Solution of exercise 1 As done in this chapter, we split our translation of this library into two files. We put the interface in a header file (ending in `.h`) and the implementation in a `.c` file. Let's look at them in turn.

Like all header files, the header file for this library uses a header guard (controlled by the macro definition `STACK_H`) to prevent declaring the types and function prototypes therein multiple times. One difference with the example seen in this chapter is the way the type `stack_t` is declared. C does not support pseudo-typedefs like we had in C0 and, were we to define `stack_t` as `stack*`, the compiler would complain that the type `stack` is undefined. C does let us define `stack_t` as `struct stack_header*`, which is what we do. Clearly, this exposes the fact that the implementation has a type called `struct stack_header` but it does not reveal the fields of this struct. We also need to include `<stdbool.h>` since the function `stack_empty` returns a `bool`. The resulting contents of this header file is as follows:

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

typedef void* stackelem;        // Element type

typedef struct stack_header* stack_t;

bool stack_empty(stack_t S)          /* O(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()                  /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, stackelem x)    /* O(1) */
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

stackelem pop(stack_t S)             /* O(1) */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;

#endif
```

Notice that the header file contains both the client and the library portions of our C1 file.

The translation of the implementation of the stack library follows closely the steps we saw in this chapter. We omit the definition of `stack_t` since we defined it in the header file. The rest is standard: we implement the contracts of each function using `REQUIRES` and `ENSURES` *inside* their body, and we replace the uses of **`alloc`** with `xmalloc` (and **`sizeof`**!).

```c
#include <stdbool.h>
#include "lib/contracts.h"
#include "lib/xalloc.h"
#include "stack.h"

typedef struct slist_node slist;
struct slist_node {               // structure of linked lists
  stackelem data;
  slist *next;
};

typedef struct stack_header stack;
struct stack_header {             // Stacks
  slist *top;
  slist *bottom;
};

bool stack_empty(stack *S) {
  REQUIRES(S != NULL);
  return S->top == S->bottom;
}

stack *stack_new() {
  stack *S = xmalloc(sizeof(stack));
  slist *p = xmalloc(sizeof(slist));
  S->top = p;
  S->bottom = p;
  ENSURES(S != NULL);
  ENSURES(stack_empty(S));
  return S;
}

void push(stack *S, stackelem e) {
  REQUIRES(S != NULL);
  slist *p = xmalloc(sizeof(slist));
  p->data = e;
  p->next = S->top;
  S->top = p;
  ENSURES(!stack_empty(S));
}

stackelem pop(stack *S) {
  REQUIRES(S != NULL);
  REQUIRES(!stack_empty(S));
  stackelem e = S->top->data;
  S->top = S->top->next;
  return e;
}
```

We include all the system and local libraries needed for this code to compile: `"lib/contracts.h"` allows us to use our C0 contract emulation, `"lib/xalloc.h"` is for `xmalloc`, `"stack.h"` is not strictly necessary but it is commonplace for a library to include its interface. The system libraries `<stdbool.h>` is not strictly necessary as the it is already imported by `"stack.h"`. We follow the convention of including a library whenever a file uses one of its functions, even if we know another library already imports it — the header guards take care of avoiding duplicate declarations.

Solution of exercise 2 There are two things we need to do to translate the `main`: the now routine replacement of **`alloc`** with `xmalloc` (and **`sizeof`**), and the more interesting translation of the loop invariant using `ASSERT`. Recall that a loop invariant is checked just before the loop guard. Therefore we need to do so once before the loop guard is checked for the very first time, and once at the very end of each iteration (just before the loop guard is checked again).

Our C file begins however by translating the included header files. The header file that corresponds to `<conio>` is `<stdio.h>`. We also need to include `"lib/contracts.h"` and `"lib/xalloc.h"` since our translated code makes use of `xmalloc` and contract emulations. And since we call functions from the translated stack library, we need to include its header file `"stack.h"`.

```c
#include <stdio.h>
#include "lib/contracts.h"
#include "lib/xalloc.h"
#include "stack.h"

int main() {
  // Create a stack of ints
  stack_t S = stack_new();

  int *elem1 = xmalloc(sizeof(int));
  *elem1 = 1;
  push(S, (void*)elem1);
  int *elem2 = xmalloc(sizeof(int));
  *elem2 = 2;
  push(S, (void*)elem2);
  int *elem3 = xmalloc(sizeof(int));
  *elem3 = 3;
  push(S, (void*)elem3);

  int i = 3;
  ASSERT(1 <= i && i <= 3);
  while (i > 1) {
    int *elem = (int*)pop(S);
    assert(*elem == i);
    i--;
    ASSERT(1 <= i && i <= 3);
  }

  printf("All tests passed!\n");
  return 0;
}
```

Solution of exercise 3

We need to include the prototype of the function `stack_free` and the definition of the type `free_elem` in the in interface of our translated library:

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

typedef void* stackelem;        // Element type

typedef void free_elem(stackelem e);   // Function to free an element

...

void stack_free(stack_t S, free_elem* f) /* O(n) */
/*@requires S != NULL; @*/ ;

#endif
```

Although we have a lot of freedom as to where we place them in this file, a sensible organization is to declare `free_elem` together with the rest of the client interface portion of the header file and to declare `stack_free` in the library interface part.

The type declaration for `free_elem` is similar to what we saw in this chapter. The function `stack_free` has one precondition, that the stack parameter not be `NULL`.

The code for the function `stack_free` goes in the implementation file. It can be written in many ways. One organization that helps avoiding mistakes is to follow the structure of the types. Since this stack library relies on list segments, we write the helper function `free_slist` that frees the nodes (and possibly the data) of a list segment given its endpoints. Since list segments are exclusive on the right (their rightmost node is a dummy node), this function shall not free the dummy node. The function `stack_free` calls `free_slist` to free the segment, it frees the dummy node, and finally frees the stack header.

```
void free_slist(slist *start, slist *end, free_elem *f) {
  REQUIRES(start != NULL && end != NULL);
  slist *node = start;
  while (node != end) {
    if (f != NULL)
      (*f)(node->data);
    slist *copy = node;
    node = node->next;
    free(copy);
  }
}

void stack_free(stack* S, free_elem* f) {
  REQUIRES(S != NULL);
  free_slist(S->top, S->bottom, f);  // free segment
  free(S->bottom);                   // free dummy node
  free(S);                           // free stack
}
```

Solution of exercise 4 In the client code file, we need to free the stack before returning. Because the stack is not empty at this point, we need to free the data remaining in it. We do so by passing the function `free` as the second argument of `stack_free`. We also need to free the elements we popped since they were allocated with `xmalloc`. The resulting code is as follows:

```
int main() {
  // Create a stack of ints
  stack_t S = stack_new();

  int *elem1 = xmalloc(sizeof(int));
  *elem1 = 1;
  push(S, (void*)elem1);
  int *elem2 = xmalloc(sizeof(int));
  *elem2 = 2;
  push(S, (void*)elem2);
  int *elem3 = xmalloc(sizeof(int));
  *elem3 = 3;
  push(S, (void*)elem3);

  int i = 3;
  ASSERT(1 <= i && i <= 3);
  while (i > 1) {
    int *elem = (int*)pop(S);
    assert(*elem == i);
    free(elem);                          // ADDED
    i--;
    ASSERT(1 <= i && i <= 3);
  }
  stack_free(S, &free);                  // ADDED

  printf("All tests passed!\n");
  return 0;
}
```

Since this code uses `free`, we also want to include `<stdlib.h>`.

Solution of exercise 5 Calling Valgrind with `--leak-check=full` produces the following output (yours may look a bit different):

```
% valgrind --leak-check=full a.out
  ...
==8978== Command: a.out
==8978==
All tests passed!
==8978==
==8978== HEAP SUMMARY:
==8978==     in use at exit: 32 bytes in 2 blocks
==8978==   total heap usage: 9 allocs, 7 frees, 1,116 bytes allocated
==8978==
==8978== 32 (16 direct, 16 indirect) bytes in 1 blocks are definitely lost in loss record 2 of
==8978==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8978==    by 0x10889A: xmalloc (xalloc.c:32)
==8978==    by 0x10894A: push (stack.c:38)
==8978==    by 0x108AEA: main (stack-test.c:19)
==8978==
==8978== LEAK SUMMARY:
==8978==    definitely lost: 16 bytes in 1 blocks
==8978==    indirectly lost: 16 bytes in 1 blocks
==8978==      possibly lost: 0 bytes in 0 blocks
==8978==    still reachable: 0 bytes in 0 blocks
==8978==         suppressed: 0 bytes in 0 blocks
==8978==
==8978== For counts of detected and suppressed errors, rerun with: -v
==8978== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The Valgrind output says that there are `32 bytes in two blocks` `in use at exit`. It then identifies line 19 of the `main` function (`push(S, (void*)elem3);`) and specifically line 38 of `push` (that's `slist *p = xmalloc(sizeof(slist));`) as where 16 of these 32 bytes are lost. Now, if we think about it, this allocation is never freed!

Line 38 of `push` creates the list node that contains the pushed element. This is the leaked memory! When popping an element from the stack, the implementation of `pop` we wrote earlier return the data value but never frees the node itself (remember that we told you not to worry about freeing memory in that exercise…).

Our code creates a 3-node list, and yet Valgrind identifies only line 19 (pushing `elem3`) as the culprit. What about the other two nodes? Line 38 accounts for the `16 bytes in 1 block` that are `definitely lost`. Line 17 (pushing `elem2`) accounts for the `16 bytes in 1 block` that are `indirectly lost`. This adds up to 32, which is the total amount of leaked memory. However, `elem1` is not leaked: this is because we never pop it and therefore it is freed when we call `stack_free`.

We fix this problem by freeing the node when popping from the stack. The revised code for `pop` is as follows:

```
stackelem pop(stack *S) {
  REQUIRES(S != NULL);
  REQUIRES(!stack_empty(S));
  stackelem e = S->top->data;
  slist *tmp = S->top;                  // ADDED
  S->top = S->top->next;
  free(tmp);                            // ADDED
  return e;
}
```

Solution of exercise 6 The code for `stack_copy` moves all items in its input stack `S` into a temporary stack, and them moves them back both in `S` and in the stack to be returned. The resulting code is as follows:

```
stack_t stack_copy(stack_t S) {
  REQUIRES(S != NULL);
  stack_t temp = stack_new();

  while (!stack_empty(S))
    push(temp, pop(S));

  stack_t copy = stack_new();
  while (!stack_empty(temp)) {
    stackelem e = pop(temp);
    push(S, e);
    push(copy, e);
  }
  stack_free(temp, NULL);
  ENSURES(copy != NULL);
  return copy;
}
```

Solution of exercise 7 When running the compiled code, it either prints a number that doesn't look a bit like what we put in `elem2` (2) or it aborts execution with a segmentation fault.

Valgrind reveals that there is an invalid read on the line where we print `x`. This suggest that `x`, the data element we just popped from `S_copy` is invalid. Valgrind further informs us that the call to `free` on the next line is invalid. A good reason for this to be the case is if that the top element of `S_copy` had already been freed (we would then have a double-free). The only thing we free up to this point of the execution is `S`. This is the issue! We call `stack_free` with `free` as its second argument. This frees the stack and

all the elements it contains. Since `S_copy` contains aliases to these elements, any attempt to dereference them will result in an invalid read, and any attempt to free them will result in a invalid free.

The fix is to call `stack_free` with `NULL` as its second argument:

```
stack_free(S, NULL);
```