15-122: Principles of Imperative Computation, Spring 2011
Assignment 7: From C_0 to C

William Lovas (`wlovas@cs`) Ananda Gunawardena Tom Cortina
Jason Chow

Out: Tuesday, June 14, 2011
Due: Friday, June 17, 2011

(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: Fundamentals of C (25 points)

The written portion of this week's homework will give you some practice in transitioning from C_0 to C programming basics. As you know, C_0 is a safe subset of C and protects you from many of the nuances in C such as illegal memory access errors or data type violations. In this homework, we will discuss questions related to typedef's, macros, memory management and basics of C pointers. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 C Programming Issues

Exercise 1 (7 pts)**.** For each of the following problems, state what is wrong with the code and show how to correct it. Do not just try to compile it and write down the error message. (Some of these will compile without error, and some will even run and produce output, but they all contain conceptual errors that may affect correctness.) Read the code and explain what is being done wrong, conceptually. Think about all the ways to incur undefined behavior in C, including accessing unallocated or uninitialized memory, dereferencing `NULL`, dividing by zero, and arithmetic overflow.

(a)
```
#include <stdio.h>
#include <string.h>

int main() {
  char *w;
  strcpy(w,"C programming");
  printf("%s\n", w);
  return 0;
}
```

(b)
```
#include <stdio.h>
#define MULT(X,Y) (X*Y)

int main() {
  int c = MULT(2+3,3+4);
  printf("(2+3)*(3+4) is = %d\n", c);
  return 0;
}
```

(c)
```
#include <stdlib.h>
#include "xalloc.h"

int main() {
  int* a = xmalloc(100);
  for (int i=0; i<100; i++)
    a[i]=i;
  return 0;
}
```

(d)
```
#include <stdlib.h>
#include <string.h>
#include "xalloc.h"

int main() {
  char* name = xmalloc(strlen("wordpress"+1));
  strcpy(name,"wordpress");
  return 0;
}
```

(e) The standard string library function `strncpy(dest, src, n)` copies the specified number of characters `n` from the source string `src` to the destination string `dest`.

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

int main() {
  char *letter_data = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char a[16];
  strncpy(a, letter_data, sizeof(a));
  printf("The first sixteen letters are: %s\n", a);
  return 0;
}
```

(f) This code fragment shows a C function that is called from another function. It is supposed to return the result only if no overflow occurs.

```
#include <assert.h>

int oadd(int x, int y) {
   int result = x + y;
   if (x > 0 && y > 0) assert(result > 0);
   if (x < 0 && y < 0) assert(result < 0);
   return result;
}
```

(g) This code fragment shows a C function that is used as part of an implementation for stacks. Assume that `is_stack` returns true.

```
#include "contracts.h"
#include <stdlib.h>

int stack_size(stack S) {
   REQUIRES(is_stack(S));
   list L = malloc(sizeof(struct list));
   int size = 0;
   for (L = S->top; L != S->bottom; L = L->next)
      size++;
   return size;
}
```

Exercise 2 (7 pts)**.** Answer the following questions briefly and clearly. Your answers should not be more than few lines of explanation. Provide a simple example when appropriate to support your explanation.

(a) Assume that ints are 4 bytes and that signed quantities are represented using two's complement. What is the behavior of the following program, and why?

```
#include <stdio.h>

int main() {
    signed char sc = -5;
    unsigned char uc = sc;

    signed int si1 = (int) sc;
    signed int si2 = (int) uc;

    unsigned int ui1 = (int) sc;
    unsigned int ui2 = (int) uc;

    printf("sc: %d, uc: %u\n", sc, uc);
    printf("si1: %d, si2: %d\n", si1, si2);
    printf("ui1: %u, ui2: %u\n", ui1, ui2);

    return 0;
}
```

(b) Which of the following is legal and which one is illegal? Why? Justify your answer.

```
#define banana int
unsigned banana i;

typedef int banana;
unsigned banana i;
```

(c) The first 14 bytes of a bitmap file are reserved for partial header information is as defined by the following struct.

```
struct bmp_header {
    unsigned short type;        /* BMP type identifier */
    unsigned int size;          /* size of the image file in bytes*/
    unsigned short reserved1;
    unsigned short reserved2;
    unsigned int offset;        /* starting address of the byte */
};
```

Assume that `unsigned short` values are 2 bytes and `unsigned int` values are 4 bytes. Suppose we read first 14 bytes of some bitmap file into a header that is defined as follows.

```
char header[14];
```

Assuming that struct fields are laid out contiguously in memory, how would you extract the size of the image from the given data? Write one or two lines of C code to extract the bitmap size from `header`.

1.2 Implementing an Abstract Data Type in C

Exercise 3. (11 pts) A polynomial of degree n, $n \geq 0$, can be expressed as follows:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x^1 + a_0$$

where $a_n, a_{n-1}, ..., a_1, a_0$ are integer coefficients.

Consider a C struct definition for a polynomial of degree n, $n \geq 0$, as shown below:

```
struct poly {
    int* coeffs;      // array of coefficients: a0, a1, a2, ..., an
    int degree;       // degree of polynomial: n
};
typedef struct poly* poly;
```

Also assume that you can specify preconditions and postconditions for functions using the following annotation macros in C:

```
#define REQUIRES(X) assert(X)
#define ENSURES(X) assert(X)
```

For each of the following exercises, write the required C function and include appropriate preconditions and postconditions (as necessary) using the macros above.

(a) (1 pt) Write a C function that returns true if polynomial p is a "valid" polynomial using the following interface:

```
bool is_poly(poly p);
```

You may assume `stdbool.h` is included in the same file.

(b) (2 pts) Write a C function that allocates and returns a new polynomial of degree d with all coefficients set to 0 using the following interface:

```
poly new_poly(int d);
```

(c) (2 pts) Write a C function that sets the coefficient of the *i*th term of a polynomial *p* to *c* using the following interface:

```
void set_coeff(poly p, int i, int c);
```

(d) (2 pts) Write a C function that allocates and returns a new polynomial equal to the sum of two polynomials. To add two polynomials, simply add the coefficients of like-powered terms. The degree of the resulting polynomial should be equal to the larger of the degrees of the two polynomials being added. Use the following interface:

```
poly add_poly(poly p, poly q);
```

(e) (3 pts) Write a C function that computes the value of a polynomial *p* at the given value *x* and returns the result. Your function should have a worst-case runtime complexity of $O(n)$ for a polynomial of degree n. (Consider using Horner's rule to minimize the number of multiplications.) Use the following interface:

```
int eval_poly(poly p, int x);
```

(f) (1 pts) Write a C function that frees a given polynomial. Use the following interface:

```
void free_poly(poly p);
```

Be sure to follow the golden rule: free only the memory that was allocated by `new_poly`!

2 Programming: Ropes - An Alternative to Strings (25 points)

For the programming portion of this week's homework, you'll implement in C the "rope" data structure, an alternative representation of strings. You'll learn how to transition your knowledge of C_0 programming to C, with an emphasis on explicit memory management. In particular, you will learn how to allocate and deallocate memory, C style.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw7-starter.zip` from the course website.

Compiling and running. Compile your code using GCC. The following set of options will catch many common mistakes at compile-time:

```
gcc -Wall -Wextra -std=c99 -pedantic -Werror <files...>
```

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. To enable assertion checking, ensure that `DEBUG` is defined using the `-DDEBUG` option. **Warning:** *You will lose credit if your code does not compile.*

To detect any invalid memory accesses or memory leaks, you can run your compiled binary through Valgrind:

```
valgrind ./a.out
```

Use the `-v` option for verbose output, or the `--leak-check=full` option to generate a more complete report of possible memory leaks. **Warning:** *You will lose credit if your code has invalid memory accesses or memory leaks.*

Submitting. Once you've completed some files, you can submit them by running

```
handin -a hw7 <file1> ... <fileN>
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Use the macros in `contracts.h` to write appropriate annotations for your code in a style similar to what we've been doing in C_0. Remember that writing these annotations *before* writing the code will help you understand the problem more clearly and save debugging time later. **Annotations are part of your score for the programming problems; you will not receive full credit if they are weak or missing.**

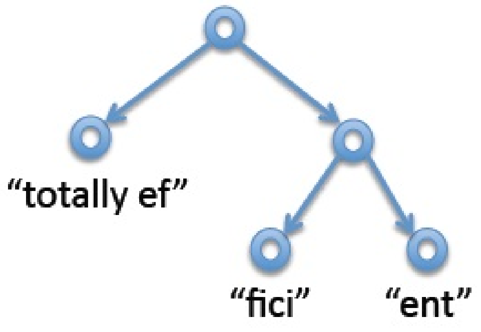

Figure 1: One possible rope representing the string "totally efficient".

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

2.1 Background

In C_0 and C, strings are typically represented as arrays of characters. This allows constant-time access of a character at an arbitrary position, but it also has some disadvantages. In particular concatenating two strings (function `string_join`) is an expensive operation since we have to create a new character array and copy the two given strings into the new array character by character. A data structure called a "rope" attempts to improve efficiency of concatenation by representing strings as binary trees, where the leaves contain ordinary strings and the intermediate nodes represent concatenations. For example, the string "totally efficient" might look as shown in Figure 1 (among many other possibilities).

2.2 Ropes

The rope is a data structure proposed by researchers Boehm, Atkinson, and Plass in 1995. The summary of their 1995 paper [1] states that,

> Programming languages generally provide a string or text type to allow manipulation of sequences of characters. This type is usually of crucial importance, since it is normally mentioned in most interfaces between system components. We claim that the traditional implementations of strings, and often the supported functionality, are not well suited

> to such general-purpose use. They should be confined to applications with specific, and unusual, performance requirements. We present ropes or heavyweight strings as an alternative that, in our experience leads to systems that are more robust, both in functionality and in performance.

The rope data structure is designed to support efficient, scalable, non-destructive operations on immutable strings.[1] Ropes gain much of their efficiency from clever use of sharing, so the main subtlety in their implementation is tracking when it is safe to free associated memory.

In order to implement ropes, we start by defining a rope node. A rope node is defined similar to a BST node as follows.

```
typedef struct rope* rope;
struct rope {
  size_t size;          /* size of the string in this rope */
  size_t position;      /* length of the left rope */
  struct rope* left;    /* pointer to left child, NULL for leaves */
  struct rope* right;   /* pointer to right child, NULL for leaves */
  char* data;           /* string data, NULL for interior nodes */
  int ref_count;        /* number of references to this rope */
};
```

Each rope node represents an immutable string. The type `size_t` stands for unsigned integers between `0` and `SIZE_MAX`, a macro defined in `stdint.h`. We use a reference counting scheme to track how many references to a given pointer exist; when no references remain, the memory pointed to may be freed. A newly constructed rope returned from the `rope_new(char *str)` has the following characteristics:

- the `size` field contains the length of the string, as computed by `strlen(str)`,
- the `position` field is set to 0,
- the `data` field is set to `str`, and
- the `ref_count` field is set to 1.

The result of a `rope_join(str1, str2)` (see Figure 2) has the following characteristics:

- the `size` field contains the total length of the represented string,
- the `position` field contains the starting index of the right rope (or equivalently, the size of the left rope),
- the `left` and `right` fields are pointers to the child ropes,

[1] C_0 `strings` and Java `Strings` are *immutable*: they cannot be modified once constructed.

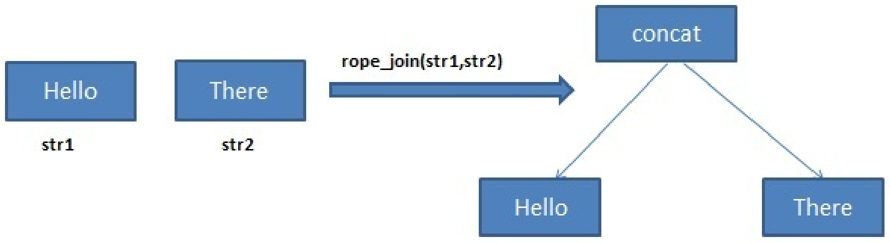

Figure 2: Concatenating two leaf ropes to form a new rope.

- the `ref_count` field is set to 1, and
- the children's `ref_count` fields are increased by 1.

The diagram in Figure 2 shows how two individual ropes are merged to form one rope.

The following tasks ask you to implement a simple rope data structure. The functions represent roughly the same interface as the C$_0$ `string` library, so let your understanding of its contracts guide your thinking!

Task 1 (3 pts). Implement the specification function `is_rope`. This function takes a rope and returns `true` or `false` depending on whether the rope satisfies the rope invariants. This function must be used to write contracts in your later code.

```
bool is_rope(rope s);
```

Task 2 (1 pt). Implement the function `rope_length`, which returns the length of the string represented by a rope.

```
size_t rope_length(rope str);
```

Task 3 (3 pts). Implement the function `rope_new`. This function takes a pointer to a `NUL`-terminated character array (i.e., a C string) and returns a new rope reperesenting it. For efficiency's sake, the function should *not* make a copy its argument—the client is responsible for the memory associated with the string.

```
rope rope_new(char* str);
```

Task 4 (3 pts). Implement the function `rope_join`. This function takes two ropes and combines them to create a single rope by creating a new parent node to connect the ropes (see Figure 2). The function should "retain" a copy of the child ropes by incrementing their reference counts.

```
rope rope_join(rope str1, rope str2);
```

Task 5 (3 pts)**.** Implement the function `rope_release`. This function takes a rope and decrements its reference count; if the count becomes zero, it also releases any child ropes and frees the memory associated with the node itself. Be careful not to free any memory that might still be in use! Use `valgrind` to test your code for invalid memory accesses and memory leaks.

```
void rope_release(rope str);
```

Task 6 (3 pts)**.** Implement the function `rope_charat`. This function takes a rope and an index and returns the character at the index. Note that character can be in the left or right subtree.

```
char rope_charat(rope str, size_t i);
```

Task 7 (2 pts)**.** Implement the function `rope_compare`. This function takes two ropes and returns 1, 0, or -1 if the first rope is lexicographically greater than, equal to, or less than the second, respectively. (This behavior is similar to the C_0 `string_compare` function and the C standard library's `strcmp` function.)

```
int rope_compare(rope str1, rope str2);
```

Task 8 (2 pts)**.** Implement the function `rope_to_chararray`. This function takes a rope and returns a `NUL`-terminated array of characters (i.e., a C string). You must explicitly allocate associated memory for the `char*` returned as the client is not expected to do any memory allocation. (But by the golden rule, the client will be responsible for deallocating that memory!)

```
char* rope_to_chararray(rope str);
```

Task 9 (5 pts)**.** We will allocate a total of 5 points for including proper annotations and avoiding any memory leaks from your code. Don't forget to include all annotations using the macros in `contracts.h`, and be sure to test your code using `valgrind` for memory leaks!

References

[1] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. Software: Practice and Experience, 25(12):1315–1330, December 1995.