CS 136 Review

Elementary Algorithm Design and Data Abstraction

Dave Tompkins • Winter 2016 • University of Waterloo

Programming Paradigms

- Functional – no mutation, garbage collector, hidden pointers
- Imperative – mutation, no garbage collectors, explicit pointers
- Object Orientated

Modularization & ADTs

Modules provides a collection of functions (also data structures and variables) that share a common aspect or purpose. Modules usually have their own files, but a file is only a module if it provides functions for use outside of the file.

A client **requires** functions that a module **provides**. A program can be made up of many modules. However, there can be no cyclical dependencies and there may only be one main function.

Three advantages to modularization:
- **Re-usability**: can be used in many other programs.
- **Maintainability**: it's easier to test/update modules, rather than an entire program.
- **Abstraction**: the client only needs to understand what a function does, not how it is implemented.

Scope

- **local** identifiers are only visible inside the local region or function body where it is defined
- **global** identifiers are defined at the top-level, and are visible to all code following its definition. Global identifiers are either:
  - **module** identifiers only visible inside the module they are defined in
  - **program** identifiers visible outside of the module they are in (they are provided)

Interface

A client is provided an interface, the implementation is hidden from the client. The interface should include (1) a description of the module, (2) a list of function it provides and (3) the contract and purpose for each provided function.

Information hiding is an important component to interface design, providing:
- **security**: prevent the client tampering with data used by the module. Not always because of malice, but to protect clients from themselves.
- **flexibility**: since the client doesn't see the implementation details, we can change the implementation without worry of hassling with the client.

Cohesion & Coupling

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

- **High cohesion** means that all interface functions are working towards a "common goal."
- **Low coupling** means that there is little interaction between modules.

ADTs

Abstract Data Types (ADTs), in practice, are implemented as data storage modules that only allow access to data through interface functions (ADT operations). The underlying data structure and implementation of the ADT is hidden from the client.

They differ from **data structures** because with a data structure, you know how the data is structured and can access it directly.

Example ADTs

- A **dictionary** is a collection of pairs (key, value). Each key is unique, and has a corresponding value that is not necessarily unique. A dictionary can be implemented using an **association list** or a **binary search tree (BST)**.
- A **collection ADT** is an ADT designed to store an arbitrary number of items. An example of a collection ADT is a dictionary.
- A **stack** ADT follows LIFO (last in, first out).
- A **queue** ADT follows FIFO (first in, first out).
- A **sequence** ADT allows items to be inserted/removed/accessed at certain positions.

Functional Intro to C

C identifiers ("names") are more limited than in Racket. They *must* start with a letter, and can only contain letters, numbers, underscores.

Typing

- Racket uses **dynamic typing**: the type of the identifier is determined at runtime.
- C uses **static typing**: the type of an identifier must be known before the program is run, and cannot change.

Function terminology

In functional terminology, we apply a function. The function consumes arguments and produces a value.
In new terminology, we call a function. A function is passed arguments, and returns a value.
C will allow you to omit parameters `my_fn()` for an arbitrary number of parameters, but better style would be `my_fn(void)` as `void` communicates and enforces that there are no parameters.

Function documentation

```
; (my-sqr x) squares x
; my-sqr: Int -> Int

(define (my-sqr x)
  (* x x))
```

```
// my_sqr(x) squares x

int my_sqr(const int x) {
  return x * x;
}
```

True & False

In C, `false` is represented by `0`, and `true` is represented by one or any non-zero value.

Declaration vs. definition

A declaration specifies the type, whereas a definition specifies the type and instructs C to create the identifier.

```
int func (int a); // Function declaration

int func (int a){ // Function definition
   return a + a;
}


extern int a; // Variable declaration

int a = 10; // Variable definition
```

Creating a module in C

To create a module in C, we place the interface and implementation into separate files.

In the **interface (.h) file**, we place declarations and documentation for the functions and variables that the module provides.

In the **implementation (.c) file**, we place all the definitions.

Include

The behaviour of C's `#include` is different than Racket's `require`. `#include` is a preprocessor directive that inserts the contents of the included interface (**.h**) file directly into the source file. Note `#include "filename"` is used for files in the same directory, and `#include <filename>` searches directories designated by the compiler. Use the latter approach for standard library header files.

Scope

Note that in C, the **static** keyword is equivalent to private, which means it is only available to the current file.

Imperative C

Side effects change the state of the program. **State** refers to the value of some data at some moment in time.

An **expression statement** is an expression followed by a semicolon.

A C block ({}) is known as a compound statement, and contains a sequence of statements.

Operators

Initialization Operator

```
const int my_number = 42; // Initialization
struct posn p = {3, 4}; // Also initalization
```

The "= 42" portion of the above definition is called initialization.

Assignment Operator

The assignment operator also produces the value of the expression.

```
int my_number;
struct posn p;
my_number = 42; // assignment operator
p = {5, 7}; // Invalid assignment!
```

sizeof Operator

Produces the number of bytes to store a type, works on identifiers or types. Ex: `sizeof(int)`

Structure operator (.)

The structure operator selects the value of the requested value. Ex: `posna.x`

Address operator (&)

The address operator produces the starting address of where the value of an identifier is storing in memory.

```
int g = 42;
int main(void) {
  printf("the value of g is: %d\n", g);
  printf("the address of g is: %p\n", &g);
}
the value of g is: 42
the address of g is: 0x68a9e0
```

Indirection Operator (*)

The indirection operator or dereference operator produces the value of what a pointer points at.

Arrow Selection Operator ->

The arrow selection operator: `ptr->field` is equivalent to `(*ptr).field`. This can only be used with a pointer (`ptr`) to a structure.

Ternary operator

It has the form: `[condition] ? [condition true] : [condition false]`. The following are equivalent:

```
int abs1(int n){
  return n < 0 ? -n : n;
}

int abs2(int n){
  if(n < 0){
    return -n;
  }
  return n;
}
```

Other C Operators

Some operators: +, -, *, /

When working with integers, the division operator (`/`) rounds toward zero.
The modulus operator (`%`) behaves the same as the remainder function.

C Model

Memory

The smallest unit of *accessible* memory is a byte, which is made out of 8 bits. Thus, it has a total of $2^8 = 256$ states. Each byte of memory has an address.

Sizes

int	4 bytes
char	1 byte
pointers	8 bytes
struct	at least sum of fields

Overflow

Trying to represent values outside of the integer limits results in overflow. Adding one to `INT_MAX` will result in it wrapping around to `INT_MIN`.

C Characters

Each character corresponds to a value in the ASCII table, which makes the following two statements equivalent:

```
char a_1 = 'a';
char a_2 = 97;
```

This also allows us to perform comparisons on chars. For instance:

```
bool is_digit(char c){
   return (c >= '0') && (c <= '9');
}
```

Sections of Memory

- **Code**: machine code is stored here.
- **Read-Only Data**: constant global data.
- **Global Data**: mutable global data. Note that all global variables are identified, reserved space for, and then their memory is initialized. All before main is called.
- **Heap**
- **Stack**: the history of return addresses is known as the call stack. Once a function is called, a new entry is pushed onto the stack, and once it is returned, it is popped off.

Stack Frames

Each stack frame contains:
- argument values
- local variables
- return address

Pass by Value

It is important to note that C makes a copy of each argument value and places a copy in the stack frame.

```c
void func(int a){
   a = 20;
}

int main(void){
   int a = 10;
   func(a);
   printf("a = %d\n", a); // prints a = 10
}
```

Loops

```c
setup statement
while (expression) {
body statement(s)
update statement
}
```

which can be re-written as a single for loop:

```c
for (setup; expression; update) { body statement(s) }

// Counting up from 0 to n-1
for (i = 0; i < n; i++) {...}
// Counting up from 1 to n
for (i = 1; i <= n; i++) {...}
// Counting down from n-1 to 0
for (i = n-1; i >= 0; i--) {...}
// Counting down from n to 1
for (i = n; i > 0; i--) {...}
```

printf & scanf

The placeholders:

%d	integers
%f	floats
%c	chars
%s	strings
%p	pointers

```c
int count = printf("ab%d\n", 42); //=> count = 5 (number of characters printed out)
// newline is one character

char c;
int count2 = scanf("%d", &c); //=> count2 = input items read in (should be 1)
```

Pointers

Pointers

```c
int val = 10;
int *p = &val; // pointer to val
int **pp = &p; //pointer to p
```

Pointer Assignment

```
int i = 5;
int j = 6;
int *p = &i;
int *q = &j;
p = q;
```

The statement `p = q` is a pointer assignment. Now, `p` points to what `q` points at. It does not change the value of `i`!

Aliasing

```
int i = 5;
int j = 6;
int *p = &i;
int *q = &j;
*p = *q;
```

However, changes the value of what p points at to the value of what q points at, so i is changed to 6. This is known as aliasing.

Note that structures are passed by value, so it is recommended that you pass the address to avoid wasteful copying of large structures.

Pointer Applications

```
void func(int *a){
   *a = 20;
}

int main(void){
   int a = 10;
   func(&a);
   printf("a = %d\n", a); // prints a = 20
}
```

Important Syntax

The operations (*p)++ and *p++ are not equivalent as shown below.

```
int a = 10;
int *p = &a;

*p++; // increments the pointer, then gets the value of the incremented pointer
printf("a = %d\n", a); // a = 10

(*p)++; //increments the value of what p is point at by 1
printf("a = %d\n", a); // a = 11
```

const and pointers

```
int a = 10;
int b = 20;

const int *p = &a; //p may point to anything, but cannot change its value
*p = 20; // INVALID!
p = &b; // This is OK

int * const p = &a; // p must always point to a, but can change its value
*p = 20 // This is OK
p = &b; // INVALID!

const int * const p = &a; // p must always point to a and cannot changes it value
```

Opaque Structures in C

C also supports opaque structures by using an incomplete declaration (no fields are declared). With incomplete declarations, only pointers to the structure can be defined.

```
struct posn; // INCOMPLETE DECLARATION

struct posn my_posn; // INVALID
struct posn *posn_ptr; // VALID
```

Arrays

Arrays store a fixed number of elements that have the same type. Each value in array is referred to as an element, which is accessed at a specific index. The index starts at zero.

Once an array is defined, all values of an array cannot be assigned at once.

The **length** refers to the number of elements in the list, whereas the **size** is the number of bytes occupied by the array.

Initialization

The value of an uninitialized array depends on the scope. An uninitialized global array is zero-filled, but local arrays are filled with garbage values from the stack.

If there aren't enough elements in the braces, the remaining values are initialized to zero (including local arrays).

```
int arr[10] = {0}; // a[0] = ... = a[9] = 9
int arr2[5] = {1, 2}; // a[2] = a[3] = a[4] = 0
```

If an array is initialized, the length can be omitted from the declaration and is automatically determined by the number of elements in the initialization.

The following are equivalent:

```
int arr[] = {1, 2, 3, 4, 5};
int arr1[5] = {1, 2, 3, 4, 5};
```

Array Pointer Notation

The array indexing syntax is an operator that performs pointer arithmetic (ie `a[i]` and `*(a + i)`). Array pointer notation is using pointer arithmetic instead of the square brackets. The following are equivalent

```
int arr[5] = {11, 21, 31, 41, 51};
printf("%d\n", a[3]); // 41
printf("%d\n", *(a + 3)); // 41
```

Example

```
int sum_array(const int *a, int len) {
  int sum = 0;
  for (const int *p = a; p < a + len; ++p) {
    sum += *p;
  }
  return sum;
}

int sum_array(const int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}
```

Strings

There is built-in C string type. However, the convention is that a C string is an array of characters terminated by a null terminator (`'\0'` or `0`).

String Initialization

In C, char arrays also support the double quote initialization syntax.

The following are all equivalent:

```
char a[] = {'c', 'a', 't', '\0'};
char b[] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[] = { 99, 97, 116, 0};
char e[4] = "cat";
char f[] = "cat";
```

It is important to note that there are differences between arrays and pointers.

```
char a[] = "pointers are not arrays";
char *p = "pointers are not arrays";

a[0] = 'P'; // Valid
p[0] = 'P'; // Invalid
```

This is because `char *p` is initialized to point to a string literal.

String Comparison – Lexicographical order

Strings are compared character by character. The string with a smaller first character (by ASCII) than the other string precedes it. Otherwise, the characters are the same, and the next characters are compared. If the end of one of string is reached, it precedes the other.

The following are in lexicographical order:

```
 "" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
```

The function `strcmp(s1,s2)` returns zero if strings are identical. If `s1` precedes `s2`, then it returns a negative integer, Otherwise, it returns a positive integer.

```
char str_a[] = "aa";
char str_a1[] = "aa";

if (str_a == str_a1) ... // False (diff. addresses)
if (strcmp(str_a, str_a1) == 0) ... // True (proper comparison)
if (strcmp("aa", "bb") < 0) ... // True ("aa" precedes "bb")
if (strcmp("bb", "aa") > 0) ... // True ("bb" follows "aa")
```

String Literal

C strings used in statements (with `printf` and `scanf`) are known as string literals. For each string literal, a null terminated `const char` array is created in the read-only data section.

Efficiency

Time efficiency is how long it takes an algorithm to solve a problem, where as **space efficiency** is how much memory an algorithm requires to solve a problem.

Big O Notation

We are interested in the **order** of the running time. The order is the dominant term in the running time without any constant coefficients.

Ranked from smallest to largest, the orders we use are:

$$O(1) \quad O(\log n) \quad O(n) \quad O(n \log n) \quad O(n^2) \quad O(n^3) \quad O(2^n)$$

When adding orders, the result is the largest of the two orders. When multiplying orders, the result is the product of the two orders.

Running Times of Common Functions

strlen, strcpy	$O(n)$, where n is the length of the string
strcmp	$O(n)$, where n is the length of the shortest string
printf, scanf (no strings)	$O(1)$
printf, scanf (strings)	$O(n)$, where n is the length of the string

Iterative Analysis

1. Work from the innermost loop to the outermost
2. Determine the number of iterations in the loop (in the worst case) in relation to the size of the input (n) or an outer loop counter
3. Determine the running time per iteration
4. Write the summation(s) and simplify the expression

Common Summations

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

$$\sum_{i=1}^{n} O(1) = O(n)$$

$$\sum_{i=1}^{n} O(n) = O(n^2)$$

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

$$\sum_{i=1}^{n} O(i^2) = O(n^3)$$

Recurrence relations

1. Identify the order of the function excluding any recursion
2. Determine the size of the input for the next recursive call(s)
3. Write the full recurrence relation (combine step 1 & 2)
4. Look up the closed-form solution in a table

$T(n) = O(1) + T(n - k_1)$	$= O(n)$
$T(n) = O(n) + T(n - k_1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n - k_1)$	$= O(n^3)$
$T(n) = O(1) + T(\frac{n}{k_2})$	$= O(\log n)$
$T(n) = O(1) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n)$
$T(n) = O(n) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n \log n)$
$T(n) = O(1) + T(n - k_1) + T(n - k_1'))$	$= O(2^n)$

Where $k_1, k_1' \geq 1$ and $k_2 > 1$

The Heap

The heap can be thought of as a big pool of memory that available to your program. Memory is dynamically borrowed from the heap. This is called **allocation**. When the memory is no longer needed, it is returned. This is called **deallocation**. If too much memory has been already allocated, attempts to borrow additional memory fail.

malloc

The `malloc` function obtains memory from the heap dynamically.

```
// malloc(s) requests s bytes of memory from the heap
//   and returns a pointer to a block of s bytes, or
//   NULL if not enough memory is available
// time: O(1) [close enough for this course]
```

When allocating use **`sizeof`** with `malloc` to improve communication and portability.

free

For every block of memory allocated by `malloc`, you must `free` it eventually.

```
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects: the memory at p is invalid
// time: O(1)
```

You cannot `free` memory that is already has been freed. That's why it is usually good style to assign `NULL` to a freed pointer variable.

realloc

The function `realloc` will automatically resize a memory block. If the size is smaller, the extraneous memory is freed. If is is larger, additional memory is allocated.

```
// realloc(p, newsize) resizes the memory block at p
//   to be newsize and returns a pointer to the
//   new location, or NULL if unsuccessful
// requires: p must be from a previous malloc/realloc
// effects: the memory at p is invalid (freed)
// time: O(n), where n is newsize
```

Memory Leaks

Memory leaks occur when allocated memory is not eventually freed. Programs that leak memory may suffer degraded performance or eventually crash.

```
int *ptr;
ptr = malloc(sizeof(int));
ptr = malloc(sizeof(int)); // Memory Leak!
```

In this example, the address allocated by the first `malloc` call has been overwritten. Since we no longer have any pointers to the memory, we cannot free it.

Garbage Collection

Many modern languages including Racket have a garbage collector. A garbage collector detects when memory is no longer being used and frees it automatically. However, the convenience often can come at the cost of performance.

Amortized Analysis

To maximize performance, since `realloc` is $O(n)$, we must use `realloc` judiciously. This can be achieved by allocating more memory than necessary and only call `realloc` once we have reached its max size.

A popular strategy is to double the size of the array when it is full. This requires we keep track of the current length as well as the allocated length. For n iterations, the total resizing is at most $O(n)$.

Misc

Imports

stdio.h	for printing/reading input
assert.h	for assert(exp);
stdbool.h	for true/false
stdlib. h	includes exit/malloc/etc
limits.h	for limits such as INT_MAX
string.h	for strings

typedef

The **typedef** keyword allows you to create your own "type" from existing ones. It is common to use a different coding style (in this case CamelCase) when defining a new type with **typedef**.

```
typedef int Integer;
typedef int *IntPtr;

Integer i;
IntPtr p = &i;
```

void pointers

The **void** pointer (**void** *) is the closest C has to a generic type. **void** pointers can be use to point to any type. However, *they cannot be directly referenced.*

```
int i = 42;
void *vp = &i;

int j = *vp; // INVALID

int *ip = vp;
int k = *ip; // VALID
```

void pointers are well-suited for ADT implementations because they allow the use of multiple types. However, two potential complications arise because the type is unknown. The first is memory management, and the second is comparisons.

The solution to the first is make the interface state whose responsibility it is to free an item. Either solution presents problems. If it's the client's responsibility, the client has to be extra careful not to cause any memory leaks. If it's the ADT's responsibility, problems arise with structures.

The solution to the second is to use a comparison function pointer. Thus, to compare items, the ADT just calls the comparison function.

Racket Snippet

```
(define (noisy-add x y)
  (begin
  (printf "HEY! I'M ADDING ~a PLUS ~a!\n" x y)
  (+ x y)))
```

```
;; begin only produces the last value, but evaluates all expressions before it

(define (noisy-add x y)
  (printf "HEY! I'M ADDING ~a PLUS ~a!\n" x y)
  (+ x y))

;; in full racket, there is an implicit begin
;; note ~a is a placeholder for any value
```