CS 136

.RKT IN C

PROFESSOR M. PETRICK • WINTER 2014 • UNIVERSITY OF WATERLOO

Last Revision: April 4, 2015

Table of Contents

Future Modifications

1 Modularization

Definition 1.1. A **module** is a collection of functions that share a common aspect or purpose. **Modularization** is dividing programs into modules.

- Reusability
- Maintainability
- Abstraction

Definition 1.2. **provide** is used in a module to specify the identifiers availabie in the module.

fun.rkt

```
(provide fun?) ;Allows use of function outside of program
(define lofn `(-3 7 42 136 1337 4010 8675309))
;; (fun? n) determines if n is a fun integer
;; fun?: Int -> Bool
(define (fun? n)
  (not (false? (member n lofn))))
```

Definition 1.3. **require** is used to identify a module that the current program depends on.

implementation.rkt

```
(require "fun.rkt")
;;Able to use provided functions in required file
(fun? 7) ; => #t
(fun? -7) ; => #f
```

1.1 Scope

- **Local:** Visible only in local region
- **Module:** Only visible in the module it is defined in
- **Program:** Visible outside the module.

Quote. **require** also outputs the final value of any of the top-level expressions in the module. Only definitions should be included in modules.

Definition 1.4. A module **interface** is the list of functions that a module provides. Documentation should be provided.

- Description of module
- List of functions provided

- Contract and purpose for each provided function

Definition 1.5. The **implementation** is the code for the module.

- Hides implementation details from client
- Security
- Flexibility to modify implementation

Definition 1.6. **High cohesion** means that all interface functions are related.

Definition 1.7. **Low coupling** means that there is little interaction between modules.

Quote. Always truncate decimals

```
int main (void) {
  printf(``Hello World! \n'')
}
```

Definition 1.8. %d is used as a placeholder to the values that follow.

```
printf("%d plus %d is: %d\n", 1 + 1, 2, 2 + 2);
```

In racket, a is used as a placeholder.

```
(printf ``There are ~a lights!\n'' ``four'')
(printf ``There are ~a lights!\n'' 'four) ; Both lines are same
```

Definition 1.9. Structures in C are very similar to racket.

```
struct posn {
  int x;
  int y;
}; //Do not forget the semicolon

const struct posn p = {3,4}; // Initialization
const struct posn pp = {y=4,x=3}; // This works too
const struct posn pp = {x=3}; // Uninitialized integers are set to 0.

const int a = p.x;
const int b = p.y;
```

Definition 1.10. **begin** produces the value of the last exprssion

```
(define (mystery)
  (begin ; implicit, this line not needed
    (+ 1 2) ; evaluated, not used
    (+ 2 2))) ;outputs 4
```

Quote. Anything that is not #f in Racket is true.

2 Imperative Programming

Definition 2.1. The **functional programming paradigm** is to only use constant values that never change. Functions produce new values rather than changing existing ones. In functional programming, there are no side effects.

Definition 2.2. A **side effect** does more than produce a value it also changes the state of the program. Sometimes used to debug.

Definition 2.3. In an expression statement, the **value** of the expression is **ignored**.

```
3 + 4;
```

Definition 2.4. A **block** {}, is known as a compound statement, and contains a sequence of statements. Within a block, **local scope definitons** can also be included.

Definition 2.5. **printf** in C returns an int representing the number of characters printed.

Definition 2.6. **Control flow statements** change the flow of a program and the order in which other statements are executed.

- **return** statement ends the execution of a function and returns a value.
- **if** and **else** statements execute statements conditionally

Quote. The defining characteristic of **imperative programming paradigm** is to **manipulate state**.

Definition 2.7. **State** refers to the value of a data at a moment in time.

Definition 2.8. When the value of a variable is changed, it is called **mutation**.

```
int x = 5;
struct posn p = {3,4};
```

Definition 2.9. **Prefix** and **postfix** increment operator:

```
x++ // Produces old value, and increments as side effect
++x // Increments x and then produces the value
```

3 C Model

Definition 3.1. A **bit** has two states: 0 or 1. A **byte** is 8 bits of storage. Each byte is in one of 256 possible states.

Definition 3.2. **Memory addresses** are represented in hex (prefixed with $0x$), so a typical address would be $0xFFFFF$.

Definition 3.3. **sizeof** produces the amount of space (bytes) a variable uses.

- A **char** is 1 byte.
- An **int** is 4 bytes.
- An **address** is 8 bytes.

Note. When a variable is initialized, three steps occur:

- Reserves space in memory to store the variable
- Records the address to the location
- Store the value of the variable at the address.

```
int n = 4;
```

identifier	type	bytes	address
n	int	4	$0x5000$

Quote. A variable definition reserves space, but declaration does not.

Note. If an int is larger than the maximum $2^31 - 1$ or smaller than the minimum -2^31, overflow will occur. Remember to always try and avoid chance of overflow wherever possible.

Note. For characters, A is 65, a is 97, space is 32, 0 is 48, and newline is 10 in ASCII.

Note. The sizeof a structure is at least the sum of the size of each field.

Definition 3.4. A **float** represents real numbers and has a larger range than int. Floats are very imprecise, and doubles are usually used instead.

3.1 Memory

Memory can be modelled as

Code
Read-Only Data
Global Data
Heap
Stack

Definition 3.5. Converting source code to machine code is known as **compiling**

Note. Global constants are stored in read-only, and global variables are stored in global data. The space is reserved before execution.

Definition 3.6. **control flow** is used to model how programs are executed.

Definition 3.7. The history of what a program needs to do is called the **call stack**. When a function is called, it is pushed onto the call stack. When a return is used, an entry is popped off the stack.

Definition 3.8. An entry pushed onto a call stack is a **stack frame**. A stack frame consists of

- Argument values
- Local variables
- Return address

Quote. When the function returns, the entire stack frame is destroyed along with its local variables.

Definition 3.9. When the stack frame is too large, it can collide with other sections of memory. This is called **stack overflow**.

Quote. All global variables that are uninitialized are automatically initialized to 0. Uninitiaized local variables have an arbitrary initial value.

3.2 Loops

Definition 3.10. **while** reeatedly loops back and executes the statement until the expression is false.

Definition 3.11. The **do** statement is similar to the while statement but evaluates the expression after execution. Because of this, the loop is always executed at least once.

Definition 3.12. **break** is used to break out of a loop.

Definition 3.13. **continue** skips the current block of execution and continues the loop.

```
int num = 6;
while (num!=0) { // 6,3,2,1, end
  if (num == 6) {
    num -= 3;
    continue;
  }
  num--;
}
```

Definition 3.14. **for** is similar to a condensed form of a while loop.

```
for (int i = 0 ; i < 5; i++) { body }
```

Any component may be omitted in a for loop. An omitted expression is always true. Commas may be used for compound statements in the setup of a for loop.

4 Pointers

Definition 4.1. The **address operator** & produces the starting address of where the value of an identifier is stored in memory.

Definition 4.2. By adding a * before an identifier, it becomes a pointer, and its value is an address.

```
i = 42;
int *p = &i; // p points to i
printf(``p is %p'', p); \\ prints the address of i
```

Definition 4.3. The **indirection operator** * is the inverse of address operator and produces the value of what a pointer points at.

```
int = 42;
int *p = &i; //points at address of i
int j = *p // 42
```

Note. C mostly ignores whitespace, so the following lines are all equivalent.

```
int *pi = &i; // style A (preferred)
int * pi = &i; // style B
int* pi = &i; // style C
```

Definition 4.4. By adding multiple asterisks, a pointer to a pointer may be declared.

```
int i = 42;
int *pi = &i; // address of i
int **ppi = &pi; // address of pi
```

Definition 4.5. **NULL** is a pointer value that represents that the pointer points to nothing.

4.1 Pointer Assignment

The value of what a pointer is pointing at may be changed. They can be dereferenced to change the value of the variable they point at without actually using the variable.

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

Note. Pointers may be used to emulate **pass by reference** even though C is pass by value.

```
void inc(int *p) {
  *p += 1;
}

int main(void) {
  int x = 5;
  inc(&x); // note the &
  printf("x = %d\n", x); // NOW it's 6
}
```

This may also be used on structures, but brackets must be added around the dereference **(*p).x**

Definition 4.6. The **arrow selection operator** (->) combines the indirection and selection operators. This may only be used with a pointer to a structure.

```
int sqr_dist (struct posn *p1, struct posn *p2) {
  const int xdist = p1->x - p2->x;
  const int ydist = p1->y - p2->y;
  return xdist * xdist + ydist * ydist;
}
```

Note. These parameters may also be mutated.

5 I/O & Testing

Definition 5.1. fprintf has an addition parameter that points to a file. It is similar to printf but prints directly to the file.

```
int main(void) {
  FILE * file_ptr;
  file_ptr = fopen("hello.txt","w");    // w for write
  fprintf(file_ptr, "Hello World!\n");
  fclose(file_ptr);
}
```

Definition 5.2. In Racket, **(read)** is used to get a value from keyboard.

```
(define key-inp (read))
```

Text is interpreted as symbols unless it is surrounded with double quotes.

Definition 5.3. In C, **scanf** is used for keyboard input. scanf returns the number of values successfully read. If there is an error, 0 is returned by scanf.

```
int count = scanf(``%d'', &i); // Reads integer, and stores it in i. Count sho
```

Note. When reading in characters, it may be beneficial to ignore whitespace.

```
int count = scanf(``%c'', &c); // May read whitespace
int count2 = scanf (`` %c'', &c); // Skips whitespace
```

6 Arrays and Strings

```
int a[5]; // Valid, size defined
int b[] = {4,8,15,16,23,42}; //Valid size can be computed
int c[]; // Invalid
```

Definition 6.1. The **length** if an array is the number of elements in the array.

Definition 6.2. The **size of an array** is the number of bytes it occupies in memory.

Quote. C does not explicitly keep track of the array length.

```
int a[] = {2,4}; // a by default points to a[0]
assert(&a == &a[0]);
assert(a == &a);
assert(*a == a[0])
```

Note. An array cannot be mutated. Only its elements can change.

```
int a[3] = {0, 0, 0};
int b[3] = {1, 2, 3};
a = b; // INVALID
```

Quote. When passing an array to a function, typically the length of the array is unknown and must be provided as a separate parameter.

Quote. In an array, pointers can be subtracted. However, pointer arithmetic is only valid within an array.

Note. If there are two pointers p and q,

$$p - q = \frac{(p-q)}{sizeof(*p)}$$

$$p + i = p + i \times sizeof(*p)$$

$p[i]$ is equivalent to $*(p+i)$.

Example 6.1. In **array pointer notation** square brackets are not used, and all array elements are accessed through pointer arithmetic.

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

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

Definition 6.3. Multi-dimensional data can be represented by mapping the higher dimensions down to one. That is, to select an element at a specific row and column, you would do data$[row \& NUMCOLS + col]$.

Definition 6.4. Function pointers store the starting address of a function within the code section. A function pointer in C can only point to a function that already exists.

```
int add1(int i) {
  return i + 1;
}

int (*fp)(int) = add1;
```

```
// Return value first, then name of function, then its parameters
// Now you can call the function with fp(int i)

// Another is to use it as a parameter
// The below function adds 70 to a number, n
int doSomething(int (*fcn)(int), int n) {
  return fcn(n) + 69;
}
```

Note. These function pointers are useful for abstract list functions seen in Racket.

```
void array_map(int (*f)(int), int a[], int len) {
  for (int i = 0; i < len; ++i) {
    a[i] = f(a[i]);
  }
}
```

6.1 Strings

Definition 6.5. A **string** in C is just an array of characters terminated by a null character ‘ 0’. If a string is initialized as an array, the null terminator is necessary, but if it is initialized with double quotes, then the null terminator is automatically added.

```
char a[] = {'c','a','t','\0'};
char b[] = ``cat'' // Equivalent
```

Definition 6.6. C strings used in statements (eg with printf) are known as **string literals**. For these statements, a null terminated const char array is created in the **read-only data section**.

Definition 6.7. The **strlen** function returns the length of the string, NOT THE LENGTH OF THE ARRAY. It also does not include the null character. It is found in the <string.h> library.

Definition 6.8. Strings are compared by their **lexicographical order**. That is, for each character, sort by the ASCII values of the characters. If the end of one string is encountered, it precedes teh other string. The **strcmp(s1,s2)** function returns 0 if the strings are identical, -1 if s1 < s2, and 1 if s1 > s2.

Note. Do not compare strings directly (eg: s1 == s2). This only compares pointers, not the actual content!.

Quote. When allocating space for a string, DO NOT FORGET THE NULL CHARACTER.

Definition 6.9. strcpy(char * dest, const char *src) copies the content of the string src to dest. **strcat(char * dest, const char *src)** appends the content of src to dest. Make sure that array is large enough so the content may be copied without **buffer overflow**.

7 Efficiency

Definition 7.1. An **algorithm** is a step-by-step description of how to solve a problem.

Definition 7.2. **Time efficiency** is how long an algorithm takes to solve a problem.

Definition 7.3. **Space efficiency** is how much space/memory an algorithm requires to solve a problem.

Definition 7.4. In this course, the running time of a function is a function of n, denoted $T(n)$. n is usually the length of the input (array, number size, etc). They are usually measured in the worst case.

Definition 7.5. **Big O Notation** showcases the **order** of a running time. That is, it is the dominant power as $n \to \infty$.

Example 7.1. When adding two orders, the result is the largest of the two orders.
$O(log\, n) + O(n) = (n)$ and $O(1) + O(1) = O(1)$.

Example 7.2. When multiplying two orders, the result is the product of the two orders.
$O(log\, n) \times O(n) = O(n\, log\, n)$ and $O(1) \times O(n) = O(n)$.

Definition 7.6. **Simple functions** are functions without recursion or iteration. In C, all operations and O(1), so the running time of a simple function is

$$O(1) + \cdots + O(1) = O(1)$$

Should include racket running times slide?

Definition 7.7. For recursive functions, we analyze the **recurrence relation**. For now, use a table to determine the runtime.

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

An example of 2^n is the recursive fibonacci sequence.

Method 7.1. Procedure for recursive functions:

1. Identify order of function excluding recursion

2. Determine size of input for next recursive calls
3. Write full recurrence relation, and look up in a table

Definition 7.8. **Iterative analysis** utilizes **summations** instead of recurrence relations.

```
for (i = 1; i <=n; i++) {
  printf("*");
} //O(n) time
```

$$T(n) = \sum_{i=1}^{n} O(1) = \underbrace{O(1) + \cdots + O(1)}_{n} = n \times o(1) = O(n)$$

Note. If a given list is of constant length (not dependant on input size), all operations are O(1).

```
for (int i = 0; i < 696969; i++) {
  sum+=a[i];
}
// O(1) linear time because a large number is still a constant.
```

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

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

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

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

Method 7.2. Procedure for iteration

1. Work from innermst loop to outermost
2. Determine number of iterations in the loop
3. Determine running time per iteration
4. Write summation and simplify expression

Note. When the loop counter changes geometrically, the number of Iterations is often logarithmic.

```c
while (n > 0) {
  // Do something
  n /= 10;
}
```

Example 7.3. Sorting Algorithm:

- Insertion Sort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Merge Sort: $O(n\,log\,n)$
- Quick Sort: $O(n^2)$

Definition 7.9. A function is **tail recursive** if the recursive call is alwasy the last expression to be evaluated. With tail recursion, the previous stack frame can be **reused** for the next recursion.

8 Dynamic Memory

Definition 8.1. **Dynamic memory** is allocaetd from the **heap** while the programming is running.

Dynamic Memory

9 Linked Data

Definition 9.1. A **linked list node** usually contains an item and a link (pointer) to the next node.

```c
struct llnode {
  int item;
  struct llnode * next;
}
```