

CS 136

.RKT IN C

PROFESSOR M. PETRICK • WINTER 2014 • UNIVERSITY OF WATERLOO

Last Revision: February 17, 2015

Table of Contents

Todo list	i
1 Modularization	1
1.1 Scope	1
2 Imperative Programming	3
3 C Model	4
3.1 Memory	5
3.2 Loops	5
4 Pointers	6
4.1 Pointer Assignment	7

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 available in the module.

fun.rkt

```

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

```

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

implementation.rkt

```

1 (require "fun.rkt")
2 ;;Able to use provided functions in required file
3 (fun? 7) ; => #t
4 (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

```
1 int main (void) {
2     printf("`Hello World! \n'")
3 }
```

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

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

In racket, a is used as a placeholder.

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

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

```
1 struct posn {
2     int x;
3     int y;
4 }; //Do not forget the semicolon
5
6 const struct posn p = {3,4}; // Initialization
7 const struct posn pp = {y=4,x=3}; // This works too
8 const struct posn pp = {x=3}; // Uninitialized integers are set to 0.
9
10 const int a = p.x;
11 const int b = p.y;
```

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

```
1 (define (mystery)
2     (begin ; implicit, this line not needed
3         (+ 1 2) ; evaluated, not used
4         (+ 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**.

```
1 3 + 4;
```

Definition 2.4. A **block** {}, is known as a compound statement, and contains a sequence of statements. Within a block, **local scope definitions** 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**.

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

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

```
1 x++ // Produces old value, and increments as side effect  
2 ++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 $0xFFFF$.

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.

```
1 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. Uninitialized local variables have an arbitrary initial value.

3.2 Loops

Definition 3.10. **while** repeatedly 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.

```
1 int num = 6;
2 while (num!=0) { // 6,3,2,1, end
3     if (num == 6) {
4         num -= 3;
5         continue;
6     }
7     num --;
8 }
```

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

```
1 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.

```
1 i = 42;
2 int *p = &i; // p points to i
3 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.

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

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

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

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

```
1 int i = 42;
2 int *pi = &i; // address of i
3 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.

```
1 int i = 5;
2 int j = 6;
3 int *p = &i;
4 int *q = p;
5 *q = j; // i = 6
```

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

```
1 void inc(int *p) {
2     *p += 1;
3 }
4
5 int main(void) {
6     int x = 5;
7     inc(&x); // note the &
8     printf("x = %d\n", x); // NOW it's 6
9 }
```

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.

```
1 int sqr_dist (struct posn*p1, struct posn*p2) {
2     const int xdist = p1->x - p2->x;
3     const int ydist = p1->y - p2->y;
4     return xdist * xdist + ydist * ydist;
5 }
```