Module 1: Functional C

Programming Paradigms

There are several different ways of thinking about computer programming. You will see several during your studies.

1. In CS135, you focused on the *functional* paradigm: our programs consisted of *pure functions*, that is, functions with no *side effects*. (We never changed the value of a variable; we instead created a new value.)
2. in CS136 we focus on the *procedural* paradigm: we will write "procedures" that are a collection of instructions, and these instructions will often have *side effects*, such as changing what is in memory.

There are several others. Different languages are designed to work best in one or more paradigms, but will generally allow work in other paradigms as well.

(For example, we *can* change the value of a variable in Racket, using the `set!` command.)

Running C in edX; introducing `printf`

Later in CS136L you will see other ways to run C code.
For now we have a simple “sandbox” in edX.

Exercise

In edX, write a simple C program that contains a function called `main` that prints a message using the function `printf`.

The first argument to `printf` must be a string, consisting of characters inside quotation marks. The characters `\n` are a special symbol called “newline”.

Exercise

Modify your program so when you run it, it displays this lovely long-stemmed rose:

```
@
|
v
|
```

The characters `%d` are a “placeholder”; each will be replaced by an **`int`** value provided as an extra argument to `printf`:

```
printf("The answer is %d\n", 2 * 3 * (3 + 4));
```

In C we write arithmetic the “normal” way; use brackets `()` as needed to specify order.

Exercise

Use `printf` along with mathematical operators like `+`, `-`, `*`, and `/` to write a program to calculate and display the value of $\frac{4(7+3)}{1+1}$.

Big picture: sequential programs

Experienced programmers are so used to this that it might feel like it goes without saying: in most languages programs are executed *sequentially*, top to bottom, left to right, one expression at at time.

In Racket: evaluate the expressions, one at a time, starting at the top of the file.

and also in C: execute the code in the function, one statement at a time. Start at the top of the function called `main`.

> **!** Every C program has exactly one function called `main`.

The `main` function always returns an `int`.

In this course, it will always return 0.

Doing otherwise will cause problems with the testing environment.

Creating new functions in C

The syntax for defining a function in C:

```
int add_3 ( int a , int b , int c ) {
    return a + b + c;
}
```

Now in our `main` function we can use this function:

```
printf("add_3 returns %d\n", add3(2,3,4));
```

In C, the “contract” is part of the language syntax.

- `int` indicates the type of the value that the function will return;
- `add_3` is the name of the new function;
- Inside round brackets `(` , `)`: `int a` , `int b` , `int c` , specify the names and types of the parameters;
- Curly brackets `{` `}` form a block of code containing: `return a + b + c;` , the body of the function.

Dynamic vs Static Typing

In Racket, we *could* write code that does not follow the contract.

```
;; qux: Int Int -> Int
(define (qux a b)
  (cond [(even? a)
         (+ a b)]
        [else
         (* a 2)]))
```

```
(qux 6 3)
```

```
(qux 6 3) ⇒ 9
```

```
(qux 5 3)
```

```
(qux 5 3) ⇒ 10
```

```
(qux 5 "trois")
```

```
(qux 5 "trois") ⇒ 10
```

But we might get an error when we run the code (a “runtime error”):

Dynamic vs Static Typing

Doing the equivalent in C is impossible; The code will not even compile.

```c
int qux(int a, int b) {
  if (0 == a % 2) {
    return a + b;
  } else {
    return a * 2;
  }
}

int main(void) {
  printf("%d\n", qux(6,3));       // OK
  printf("%d\n", qux(5,3));       // OK
  printf("%d\n", qux(5,"trois")); // compile-time error
  printf("%d\n", qux("cinq", 3)); // compile-time error
}
```

(Technically, it is only a warning, and it will “guess” that we mean something else, but its guess makes no sense.)

Returning a value

When we wrote a function in Racket, it was always just an expression. Calling the function means evaluating the expression; the function returns the value of the expression.

In C, a function may contain more than one statement. We use the keyword `return` to indicate the value that the function returns. When we call a function, the function is executed sequentially, until it reaches a `return` statement.

```c
#include <stdio.h>

int chatty_add(int a, int b) {
  printf("I got %d and %d\n", a, b);
  printf("So I'm going to add them.\n");
  printf("I should get %d\n", a + b);
  return a + b;
}

int main(void) {
  printf("%d\n", chatty_add(3, 4));
}
```

Numeric types

In Racket, you had two numeric types:

- `Int` for integers of any size
- `Num` for numbers including integers and certain non-integer real numbers:
  - rational numbers like `(/ 7 2)` ⇒ `3.5`
  - inexact numbers like `(sqrt 2)` ⇒ `#i1.4142135623730951`

  It's impossible to represent real numbers "properly". See Cantor, 1891.

In this course, we are not going to use any non-integer numbers. We will use only:

- **`int`** for integers between -2^{31} and $2^{31} - 1$ (i.e. −2147483648 to 2147483647)
- **`char`** for integers -2^7 and $2^7 - 1$ (i.e. −128 to 127)
- **`bool`** for integers 0 and 1, representing false and true

Integer operators

Several operators work exactly as the corresponding Racket function:

```
6 * 7 ⇒ 42          6 + 7 ⇒ 13          6 - 7 ⇒ -1
```

But division is different. `(/ 7 2)` ⇒ `3.5`; this is not an integer, so we cannot store it.

On integers, the `/` operator works like the Racket function `quotient`:

```
Racket: (quotient 22 7) ⇒ 3   (quotient -22 7) ⇒ -3   (quotient 22 -7) ⇒ -3
     C:          22 / 7 ⇒ 3          -22 / 7 ⇒ -3          22 / -7 ⇒ -3
```

And the `%` operator works like the Racket function `remainder`:

```
Racket: (remainder 22 7) ⇒ 1   (remainder -22 7) ⇒ -1   (remainder 22 -7) ⇒ 1
     C:          22 % 7 ⇒ 1          -22 % 7 ⇒ -1          22 % -7 ⇒ 1
```

The behaviour with negatives may be surprising. (It's not quite what Python does!)
We will mostly be using non-negative integers.

Working with digits

Exercise

Write a function `last_digit` that takes a **`int`**, and returns a **`int`** that is the last digit of the number.

```
printf("should be 5: %d\n", last_digit(245));
```

Exercise

Write a function `rest_digits` that takes a **`int`**, and returns a **`int`** that is all the digits except the last of the number.

```
printf("should be 24: %d\n", rest_digits(245));
```

Boolean values

In Racket, we had predicate functions like `<`, `>`, and `=` that return a `Bool` value:

```
(< 3 5) ⇒ #true          (> 3 5) ⇒ #false          (= 3 4) ⇒ #false
```

Similarly in C, except these are operators instead of functions. The number `0` is used for "false", and the number `1` is used for "true".

```
3 < 5 ⇒ 1
3 > 5 ⇒ 0
```

Also, a single equal sign `=` is used to change the value of a variable. We cannot use `=` to check if values are equal! Instead we must use `==`.

```
3 == 5 ⇒ 0
3 == 3 ⇒ 1
3 = 5 // error
```

! We cannot use `=` to check if values are equal! Instead we must use `==`.

`if` statements

We can now write code that **branches**, that is, does different things in different situations.

A `if/else` statement checks the "question" inside the brackets, then executes either the first or second block of code.

```
int qux(int a, int b) {
  if (0 == a % 2) {
    return a + b;
  } else {
    return a * 2;
  }
}
```

Exercise

Translate the following function into C, and run it using `printf` from your `main` function to check it works properly.

```
;; (sum-to n) Calculate 1+2+3+...+n.
;; sum-to: Nat -> Nat
(define (sum-to n)
  (cond [(zero? n)
         0]
        [else
         (+ n (sum-to (- n 1)))]))

(check-expect (sum-to 5) 15)
```

Chained Conditionals

In Racket, a `cond` statement could have any number of clauses:

```
(define (aeso n)
  (cond [(zero? n) 0]
        [(and (< n 10) (even? n))
         (+ (aeso (sub1 n)) n)]
        [(and (< n 10) (odd? n))
         (- (aeso (sub1 n)) n)]
        [(and (>= n 10) (even? n))
         (- (aeso (sub1 n)) n)]
        [else
         (+ (aeso (sub1 n)) n)]))
```

Equivalently, in C, use a sequence of **`if/else`** statements:

```
int aeso(int n) {
  if (n == 0) {
    return 0;
  } else if (n < 10 && 0 == n % 2) {
    return aeso(n-1) + n;
  } else if (n < 10 && 1 == n % 2) {
    return aeso(n-1) - n;
  } else if (n >= 10 && 0 == n % 2) {
    return aeso(n-1) - n;
  } else {
    return aeso(n-1) + n;
  }
}
```

If we omit the curly brackets are each “else”, the statements line up neatly like this, and it acts a fair bit like a `cond`.

The design recipe in C

In Racket, the design recipe helped us understand the problem, design our programs, and communicate with other programmers.

We're going to do something similar in C.

```
;; (my-div x y) produce the fraction of x/y
;; my-div: Int Int -> Num
;; Requires: y is not 0
(define (my-div x y)
  (/ x y))
```

```
// my_div(x, y) Returns the fraction of x/y
// Requires: y is not 0
int my_div(int x, int y) {
  assert(y != 0);
  return x / y;
}
```

- Like in Racket, start with a **purpose statement** as a comment, showing how it would be called and a description of what it does.
- The **contract** is already part of the code, so we don't write a comment for it.
- Any **requirements** we write a comment for...and also use `assert`.

The design in C: `assert` your requirements

```c
// my_div(x, y) Returns the fraction of x/y
// Requires: y is not 0
int my_div(int x, int y) {
  assert(y != 0);
  return x / y;
}
```

When we call `assert(expr)`

- if the value `expr` is `0`, meaning false, this indicates an error.
  The program will display a helpful message then halt immediately.
- if the value of `expr` is non-zero, meaning true, this indicates no error.
  Nothing happens.

This helps us identify where we went wrong. **As much as possible, we will always use `assert` to verify that all function requirements are met.**

In Racket we used the special form `check-expect` to test our functions:

```
;; (my-div x y) produce the fraction of x/y
;; my-div: Int Int -> Num
;; Requires: y is not 0
(define (my-div x y)
  (/ x y))

(check-expect (my-div 10 2) 5)
```

In C, we can write `assert` statements, in the `main` function, to test our other functions:

```
// Do some tests using assert.
int main(void) {
  assert(my_div(10, 2) == 5);
  assert(my_div(22, 7) == 3);
  printf("All tests passed!\n");
}
```

Practice writing recursive C

For these exercises, start by writing tests. Then include all parts of the design recipe.

Exercise

Write a function `int sum_divisible(int n, int d0, int d1)` that returns the sum of all the numbers from `0` to `n` that are divisible by one or more of `d0`, `d1`.
For example, `sum_divisible(10, 3, 5)` $\Rightarrow$ `33`

Exercise

Write a function `bool has_divisor_below(int n, int d)` that determines if any natural number between `2` and `d` divides `n`.
Then write a non-recursive function `bool is_prime(int n)` to determine if a natural number is prime.
Test carefully.

Operator precedence

We know the ordinary order of operations: BEDMAS, or possibly PEMDAS.
So our usual operators are evaluated in order:

1. **B**rackets: `()`
2. ~~**E**xponents – there is no such operator in C.~~
3. **D**ivision and **M**ultiplication: `* /` also `%`. These are evaluated *left to right*.
   So `100 / 4 / 2` means `(100 / 4) / 2`, not `100 / (4 / 2)`.
4. **A**ddition and **S**ubtraction: `+` and `-`. These are also evaluated *left to right*.
   So `100 - 4 - 2` means `(100 - 4) - 2`, not `100 - (4 - 2)`.

Where do we put `&&` and `||`? If in doubt, use brackets.

5. Logical **and**: `&&`
6. Logical **or**: `||`

This slide is very incomplete; see a C operator precedence chart for details.

tracing tools

To help debug our code, we can use `printf`. But this is "output". These debugging messages can mess up what we want to print.

For this course, we have created some macros to display values as "errors", separate from the output: `trace_int`, `trace_bool`, `trace_char`, `trace_ptr`, and more.

When we run this:

```c
int sum_to(int n) {
  trace_int(n);
  if (0 == n) {
    return 0;
  } else {
    return n + sum_to(n - 1);
  }
}

int main(void) {
  assert(sum_to(5) == 15);
  printf("All tests passed!\n");
}
```

We see this:

```
>>> [sum_to.c|sum_to|4] >> n => 5
>>> [sum_to.c|sum_to|4] >> n => 4
>>> [sum_to.c|sum_to|4] >> n => 3
>>> [sum_to.c|sum_to|4] >> n => 2
>>> [sum_to.c|sum_to|4] >> n => 1
>>> [sum_to.c|sum_to|4] >> n => 0
All tests passed!
```

The "All tests passed!" is secretly in one **stream**, and the `trace_int` messages are in another. So our "output" is **not** affected by the trace messages.

Using `trace_int`

Exercise

Write a simple-recursive `int fib(int n)` to calculate the `n`th Fibonacci number. Then add a call to `trace_int(n)`. See how many calls there are when you call `fib(15)`.

Accumulative Recursion

Consider a simple Racket function:

```
;; (sum-to n) Calculate 1+2+3+...+n.
;; sum-to: Nat -> Nat
(define (sum-to n)
  (cond [(zero? n)
         0]
        [else
         (+ n (sum-to (- n 1)))]))
```

If we do a trace, formally:

```
(sum-to 5)
⇒ (+ 5 (sum-to 4))
⇒ (+ 5 (+ 4 (sum-to 3)))
⇒ (+ 5 (+ 4 (+ 3 (sum-to 2))))
⇒ (+ 5 (+ 4 (+ 3 (+ 2 (sum-to 1)))))
⇒ (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum-to 0))))))
⇒ (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
```

...etc...

```
⇒ 15
```

At every step, the computer has to keep track of:
"remember to add this to the result, **later**, when the recursion is done".

It takes more and more memory as we recurse. `(sum-to 10000000)` needs "too much".

Instead, we will use an extra parameter, an **accumulator, to** do the calculation as we go.

Accumulative Recursion

Rewritten using an accumulator:

```
;; (sum-to-acc n acc)
;;     Calculate acc+1+2+3+...+n.
;; sum-to-acc: Nat Nat -> Nat
(define (sum-to-acc n acc)
  (cond [(zero? n)
         acc]
        [else
         (sum-to-acc (sub1 n) (+ n acc))]))

;; (sum-to n) Calculate 1+2+3+...+n.
;; sum-to: Nat -> Nat
(define (sum-to n)
  (sum-to-acc n 0))
```

Now trace:

```
(sum-to 5)
(sum-to-acc 5 0)
(sum-to-acc 4 5)
(sum-to-acc 3 9)
(sum-to-acc 2 12)
(sum-to-acc 1 14)
(sum-to-acc 0 15)
```

It does not need to keep track of a bunch of additions to do “later”.

Accumulative Recursion

To write a function using accumulative recursion, make your function be a wrapper. Then make a “helper” function that has one or more extra parameters to store the accumulation.

```
;; (sum-to-acc n acc)
;;     Calculate acc+1+2+3+...+n.
;; sum-to-acc: Nat Nat -> Nat
(define (sum-to-acc n acc)
  (cond [(zero? n)
         acc]
        [else
         (sum-to-acc (sub1 n) (+ n acc))
    ]))

;; (sum-to n) Calculate 1+2+3+...+n.
;; sum-to: Nat -> Nat
(define (sum-to n)
  (sum-to-acc n 0))
```

Exercise Translate `sum-to` into C, also creating a helper function `int sum_to_acc(int n, int acc)`.

Exercise Use accumulative recursion to write a function `int fact(int n)` that calculates $n!$.

Exercise Use accumulative recursion to write a function `int fib(int n)` that calculates the nth Fibonacci number.

There is a close connection between *accumulative* recursion and *tail* recursion.

Comments

We have seen `// comments` already in the design recipe.

In Racket, anything after a `;` until the end of a line is a comment.
In C, anything after `//` until the end of a line is a comment.

We can also have multi-line comments, starting with `/* and ending with */`.

This is especially useful if you want to comment out a block of code temporarily:

```c
int foo(int x, int y) {
  thing1();
  thing2();
  /*
  experimental_code1();  // try something...
  experimental_code2();  // try something else...
  */
  experimental_code3();  // yet another thing...
  return 0;
}
```

You cannot have nested comments using `/* this syntax */`.

`/* foo /* bar */ baz */`
will be parsed as:

- a comment "`/* foo /* bar */`",
- then nonsense code "`baz */`".

Module summary

- Informally walk through code C, step by step starting in the `main` function or elsewhere, to work out what a piece of code does.
- Write functions in C using the design recipe, to make your code easier to create and understand.
- Work with **`bool`** values, and use **`if`**/**`else`** statements.
- Use `printf`, with a format string, using `%d` to display integers, and `\n` to end each line.
- Write functions in C using simple recursion or accumulative recursion.

! In class we work with the key ideas of the module. We sometimes skip a few details. Review the official CS136 slides to ensure you see all the material.