Module 3: C Model

Models of Computation: Racket

In Racket, we used “substitution rules” to describe how to interpret an expression:

```
(define (my-sqr x) (* x x))
   (+ 2 (my-sqr (+ 3 1)))
```

⟹ `(+ 2 (my-sqr 4))`

⟹ `(+ 2 (* 4 4))`

⟹ `(+ 2 16)`

⟹ `18`

The model we used to describe what Racket does is precise and correct. But we can’t use the same strategy to describe how C works.

To “really” understand a C program, we need to look at in assembly language.
We aren’t going to do this usually. This example is just for context.

The compiler turns C code in **machine code**, where each byte has a specific meaning.

```c
int main(void) {
  int x = 0x41;
  int y = 0x42;
  while (y != 0) {
    x = x + y;
    y = y - 1;
  }
  return x;
}
```

becomes:

```
55
48 89 e5
c7 45 f8 41 00 00 00
c7 45 fc 42 00 00 00
eb 0a
8b 45 fc
01 45 f8
83 6d fc 01
83 7d fc 00
75 f0
8b 45 f8
5d
c3
```

Machine code is essentially incomprehensible, but we can disassemble it to **assembly language**, where we write a specific instruction name instead of each byte.

```
55
48 89 e5
c7 45 f8 41 00 00 00
c7 45 fc 42 00 00 00
eb 0a
8b 45 fc
01 45 f8
83 6d fc 01
83 7d fc 00
75 f0
8b 45 f8
5d
c3
```

becomes:

```
00000000000011e9 <main>:
11ed:   push   rbp
11ee:   mov    rbp,rsp
11f1:   mov    DWORD PTR [rbp-0x8],0x41
11f8:   mov    DWORD PTR [rbp-0x4],0x42
11ff:   jmp    120b
1201:   mov    eax,DWORD PTR [rbp-0x4]
1204:   add    DWORD PTR [rbp-0x8],eax
1207:   sub    DWORD PTR [rbp-0x4],0x1
120b:   cmp    DWORD PTR [rbp-0x4],0x0
120f:   jne    1201
1211:   mov    eax,DWORD PTR [rbp-0x8]
1214:   pop    rbp
1215:   ret
```

Models of Computation: C

The computer stores a **program counter** (PC), which tracks which assembly instruction it is on. Each instruction either increments the PC, or otherwise changes it.

```c
int main(void) {
  int x = 0x41;
  int y = 0x42;
  while (y != 0) {
    x = x + y;
    y = y - 1;
  }
  return x;
}
```

```
00000000000011e9 <main>:
11ed:    push    rbp
11ee:    mov     rbp,rsp
11f1:    mov     DWORD PTR [rbp-0x8],0x41
11f8:    mov     DWORD PTR [rbp-0x4],0x42
11ff:    jmp     120b
1201:    mov     eax,DWORD PTR [rbp-0x4]
1204:    add     DWORD PTR [rbp-0x8],eax
1207:    sub     DWORD PTR [rbp-0x4],0x1
120b:    cmp     DWORD PTR [rbp-0x4],0x0
120f:    jne     1201
1211:    mov     eax,DWORD PTR [rbp-0x8]
1214:    pop     rbp
1215:    ret
```

Tracing C

We're not going to look at assembly often. This was just for context.

The point is to see that inside a function, there is one instruction at a time, so a "counter" is sufficient to see which instruction we are on at a time.

We will trace our C code, reading it "line by line", somewhat informally.

> The language was designed to be readable.
> Build intuition; don't try to memorize these "rules" as disconnected facts.

We have:

```
if (expression) statement0 [else statement1]
```

> **`if`-Rule:** Execute `expression`; if it is non-zero, execute `statement0`; otherwise, execute `statement1` when present.

```
void describe(int n) {
  if (!(n % 2)) {
    printf("%d is even\n", n);
  } else {
    printf("%d is even\n", n);
  }

  if (!(n % 10)) {
    printf("%d ends with 0\n", n);
  }
}
```

if-Rule: Execute `expression`; if it is non-zero, execute `statement0`; otherwise, execute `statement1` when present.

```
if (0 == n % 3) {
  printf("%d is a multiple of 3\n", n);
} else {
  if (1 == n % 3) {
    printf("%d-1 is a multiple of 3\n", n);
  } else {
    printf("%d-2 is a multiple of 3\n", n);
  }
}
```

```
if (0 == n % 3) {
  printf("%d is a multiple of 3\n", n);
} else if (1 == n % 3) {
  printf("%d-1 is a multiple of 3\n", n);
} else {
  printf("%d-2 is a multiple of 3\n", n);
}
```

Often we will write iterative code using `while`. We have:

`while (expression) statement.`

`while`-Rule:

1. Execute `expression`;
2. if it is non-zero, execute `statement`, then start again at step 1.

```
int n = 5;
while (n > 0) {
  printf("%d\n", n);
  n = n - 1;
}
```

Exercise

Without using recursion, write a function `int factorial(int n)` that calculates `n!`.
E.g. `factorial(5)` $\Rightarrow$ `120`

Exercise

Without recursion, write a function `int sum_squares(int n)` that calculates the sum of the squares from `0` to `n`.
E.g. `sum_squares(4)` $\Rightarrow$ `4*4 + 3*3 + 2*2 + 1*1` $\Rightarrow$ `30`

`do while` loops

There are also `do-while` loops. These execute the code once before the first time they check the condition.

We have:

```
do statement while (expression);
```

`do-while`-Rule:

1. Execute `statement`.
2. Execute `expression`;
3. if it is non-zero, start again at step 1.

I can't think of the last time I used one.

(Possibly because I write a lot of Python, and it doesn't have it. No great loss.)

Iteration: `break` and `continue`

`break` jumps out of the innermost loop.

```c
int n = 10;
while (true) {
  --n;
  printf("I see %d\n", n);
  if (5 == n) {
    break;
  }
}
printf("n must be 5: %d\n", n);
```

`continue` jumps to the top of the innermost loop.

```c
int n = 10;
while (n > 0) {
  --n;
  if (!(n % 3)) {
    printf("Skipping %d\n", n);
    continue;
  }
  printf("Including %d\n", n);
}
```

It is always possible to rewrite code to not use **continue**.

xerci

Rewrite the following snippet so it does not use **continue**.

```
while (n > 0) {
  --n;
  if (!(n % 3)) {
    printf("Skipping %d\n", n);
    continue;
  }
  printf("Including %d\n", n);
}
```

`for` loops

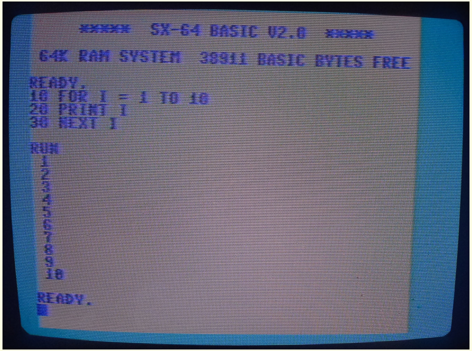
Many languages have a special loop called a `for` loop, that is specialized for counting.

Here are snippets that count from 1 to 10:

in Fortran:

```
do i = 1, 10
   print *, i
enddo
```

In BASIC:



in Pascal:

```
for i := 1 to 10 do
begin
     writeln(i);
end;
```

In C, a `for` loop can do much more. But usually we use it to count.

In C, a `for` loop is a shorthand for a `while` loop:

```
for (setup_statement; expression; update_statement) {
  body_statements;
}
```

This block loop is **almost**‡ identical →

```
{
  setup_statement;
  while (expression) {
    body_statements;
    update_statement;
  }
}
```

Exercise: Rewrite each function below so it uses a `while` loop instead of a `for` loop.

```
int main(void) {
  for (int i = 10; i > 0; --i) {
    printf("%d\n", i);
  }
}
```

```
int main(void) {
  for (int i = 10; i > 0; --i) {
    printf("%d\n", i);
    if (0 == i % 2) { continue; }
  }
}
```

‡This exercise highlights the “**almost**”.

Kinds of memory

In this course we model five **sections** (or “regions”) of memory:

Code
Read-Only Data
Global Data
Heap
Stack

Our **code** is stored in the code block.

Global **constants** are stored in the read-only data section.

Global **variables** are used in a separate section. But we don’t use them.

The **heap** we will start working with later, in module 09.

Most of what we have done so far sits in the memory section called the **stack**. Let’s explore it further.

Tracing in C: the call stack

We know what the computer does inside a function: it walks through the machine code, one instruction at a time, almost the same as walking through the C code one line at a time.

What about when we call a new function? Then the computer will:

- Create a **stack frame** that contains:
  - where it was in the previous function: the **return address**,
  - memory for parameters and local variables.
- Initialize parameters and local variables;
- Jump to the top of the new function.

This is stored in a region of memory called the **call stack**, or simply “the stack”.

When the function ends (usually at a `return` statement), it cleans up this memory and goes back (i.e. returns) to where it was in the previous function.

Tracing in C: the call stack

Each **stack frame** contains:

- where it was in the previous function: the **return address**,
- memory for parameters and local variables.

Example: I will trace this code:

```c
int bat(int n) {
  return n % 2;
}

int bar(int x) {
  int res = x * 2;
  return res;
}

int foo(int x) {
  int b = x / 3;
  int res = 64 + bat(b);
  return res;
}

int main(void) {
  int result = foo(50);
}
```

Tracing in C: the call stack

The step-by step construction of the stack is crucial. A slide can only capture a “snapshot” of it. When we reach `bat`, the call stack looks like this:

`bat`	
return address:	`foo`, line 12
`n`:	16
`foo`	
return address:	`main`, line 17
`x`:	50
`b`:	16
`res`:	???
`main`	
return address:	OS
`result`:	???

```
1   int bat(int n) {
2     return n % 2;
3   }
4
5   int bar(int x) {
6     int res = x * 2;
7     return res;
8   }
9
10  int foo(int x) {
11    int b = x / 3;
12    int res = 64 + bat(b);
13    return res;
14  }
15
16  int main(void) {
17    int result = foo(50);
18  }
```

Tracing in C: the call stack

Exercise

Use the same ideas to trace this.
Consider carefully what the stack looks like:

1. When it prints, and
2. When `main` returns.

Each **stack frame** contains:

- where it was in the previous function: the **return address**,
- memory for parameters and local variables.

```c
void base(int x) {
  printf("Here: %d\n", x);
}

int ace(int x, int y) {
  int z = x + y;
  if (x >= y) {
    base(x);
    return x;
  } else {
    x = ace(x + 1, y - 1);
    ++x;
    return x;
  }
}

int main(void) {
  int y = 5;
  y = ace(y, 8);
}
```

Overflow

On most modern architectures an `int` value is stored using 32 bits.

Thus the smallest number it can store is -2^{31}, and the largest is $2^{31} - 1$.

These are defined as macros `INT_MIN` and `INT_MAX`.

If you attempt to go beyond these values, **the behaviour is undefined**.

Depending on how your compiler is configured, it could do different things:

- it might "roll over" so `2147483647 + 1` $\Rightarrow$ `-2147483648`
- it might detect the overflow, and crash with an error.
  (Use `-ftrapv` in gcc to get this behaviour.)
- potentially, it could do something else. **It depends on your hardware architecture.**

Most of the time we stick to numbers small enough that this doesn't matter.

If we needed larger numbers, we could use a 64-bit integer, or a library that provides arbitrarily-large integer types, like in Racket.

`char` values

In C, `char` is a kind of very small integer.

```
char mychar = 42;
printf("%d\n", mychar);
```

A `char` is constrained to be between -128 and 127.

We use `char` values 0–127 to represent various characters, including printable characters like letters, digits and punctuation, as well as special characters like newline and tab.

We can create a `char` value either by typing in a number, or by putting the corresponding ASCII character inside single quotation marks. They're equivalent.

```
char ch65 = 65;
char ch_A = 'A';
assert(ch_A == ch65);
```

```
char ch10 = 10;
char ch_newline = '\n';
assert(ch10 == ch_newline);
```

```
char ch0 = 0;
char ch_bslash0 = '\0';
assert(ch0 == ch_bslash0);
```

`char` values and UNICODE

Negative numbers are used in different ways by different systems.

Normally we map the negative numbers to lie from 128 to 255.
Then in UTF-8, a part of UNICODE:

- the Ukrainian letter Ґ is represented as the 2 numbers 210, 144,
- the Korean character 눈 is represented as the 3 numbers 235, 136, 136.

In this course we will stick to single-byte characters.

Reading and writing `char` values

In `cs136.h` we have defined the function `read_char` that reads a single character from stdin, and returns `READ_CHAR_FAIL` if it cannot read.

It has a parameter, that should be either `READ_WHITESPACE`, or `IGNORE_WHITESPACE`.

If it is `IGNORE_WHITESPACE`, it will read input until it finds a character that is not “whitespace”: space, newline, etc.

Exercise

Write a program that reads all characters, and prints one line, indicating how many characters there are.
e.g. if input is: `Able was I ere I saw Elba` it prints `26`.

Exercise

Make a **tiny** change so it instead reads all characters, and prints one line, indicating how many **non-whitespace** characters there are.
e.g. if input is `Able was I ere I saw Elba`, it prints `19`.

Assume the last line always ends with a newline.

In a `printf` format string, we can interpret a `char` value:

- as an `int` using `%d`, or
- as a character using `%c`.

```
char c = 65;
printf("As a number: (%d); as a character, (%c)\n", c, c);
```

We see:

```
As a number: (65); as a character, (A)
```

Exercise: Write a program that reads all characters, and displays the numeric value of each.

Racket Symbols

In Racket we might have use symbols, so e.g. `'hearts`, `'diamonds`, `'spades`, `'clubs` are distinct values.

```
(symbol=? 'hearts 'spades) ⟹ #false
```

To do something similar, we simply create global `int` constants with different values:

```
const int HEARTS = 1;
const int DIAMONDS = 2;
const int SPADES = 3;
const int CLUBS = 42;
```

```
HEARTS == SPADES ⟹ 0
```

(The keyword `enum` lets us do this automatically, but we're not going to use it.)

A structure is a way to join together multiple pieces of data into a single value.

In Racket, we made available a new type of structure by writing something like this:

```
(define-struct myposn (x y))
```

Then we created such a value and stored it:

```
(define p (make-myposn 3 5))
```

And used “selectors” to get the fields:

```
(myposn-x p) ⇒ 3
(myposn-y p) ⇒ 5
```

The equivalent is C to define the new type is:

```
struct posn { // this defines the new type
  int x;
  int y;
}; // You will forget this semicolon. :D
```

To define such a variable:

```
struct posn p = {3, 5};
```

Or, equivalently,

```
struct posn p = { .y = 5, .x = 3 };
```

To get fields, use the dot operator:

```
printf("(%d, %d)\n", p.x, p.y); // (3, 5)
p.x += 4;
printf("(%d, %d)\n", p.x, p.y); // (7, 5)
```

```c
struct posn { // this defines the new type
  int x;
  int y;
}; // You will forget this semicolon. :D
```

```c
int main(void) {
  struct posn p = {3, 5};
  printf("(%d, %d)\n", p.x, p.y); // (3, 5)
  p.x += 4;
  printf("(%d, %d)\n", p.x, p.y); // (7, 5)
}
```

Exercise

Write a function `struct posn add_posn(struct posn a, struct posn b)` so we can do this:

```c
struct posn p0 = {3, 5};
struct posn p1 = {10, 20};
p0 = add_posn(p0, p1);
printf("(%d, %d)\n", p0.x, p0.y); // (13, 25)
```

We cannot use `==` to compare `struct` values.

If we try `p0 == p1`, we get a compiler error: `error: invalid operands to binary ==`.

Exercise

Write a function `bool posn_eq(struct posn p, struct posn q)` that determines if the contents of `p` and `q` are equal.
Try to do this in one line, without using an `if` statement.

Module summary

- Understand that C is compiled to machine code, and that in the machine code, it is sufficient to know which instruction we are on.
- Be able to trace C code, including `if` statements, loops, and stack traces.
- Be aware of issues of overflow.
- Work with `char` values, including reading (`read_char`) and printing with `printf` using `%c`.
- Be aware of the idea of using distinct integer constants to represent distinct symbols.
- Work with `struct` values.

! 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.