Functions and Text

David Marchant
Based on slides by Troels Henriksen

2022-09-11

Procedures at machine level

- The call stack
- Calling conventions
- Recursion by example

Data representations

- Examining data in C
- A machine view of text
- Binary IO

The problem with procedures

```
f(a,b,c);
g(x,y,z);
h(a,y,c);
```

- Calling a procedure (or function) requires jumping to *different code*.
- But:
  - How do we pass arguments to the procedure?
  - How does it return its results?
  - How do we prevent the procedure we call from overwriting the registers we are using?

The basic problem

How do we “suspend” execution of the current procedure, let another procedure take over, and resume execution from the *call site* afterwards?

Main instructions: `jal` and `jalr`

Instruction	Meaning
`jal` $\mathtt{x}_i$, k	x_i = PC + 4; PC += k
`jalr` $\mathtt{x}_i$, $k(x_j)$ `jalr` $\mathtt{x}_i$, x_j, k	x_i = PC + 4; PC = $\mathtt{x}_j$ + k x_i = PC + 4; PC = $\mathtt{x}_j$ + k

- Jump-and-Link jumps to an address and stores the call site address in the "link register" $\mathtt{x}_i$.
- Jump-and-Link-Register takes address from register value.
  - Used to *return* from procedure.

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	0x102	0x101	**0x100**	0x0ff	0x0fe	0x0fd	0x0fc	0x0fb	...
...	de	ad	be	ef	13	37	04	20	...
					↑				

SP: `0x0fe`

Operation:

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	0x102	0x101	**0x100**	0x0ff	0x0fe	0x0fd	0x0fc	0x0fb	...
...	de	ad	be	ef	13	38	04	20	...
						↑			

SP: `0x0fd`

Operation: `push(0x38)`

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	`0x102`	`0x101`	**`0x100`**	`0x0ff`	`0x0fe`	`0x0fd`	`0x0fc`	`0x0fb`	...
...	`de`	`ad`	`be`	`ef`	`13`	`38`	`14`	`20`	...
							↑		

SP: `0x0fc`

Operation: `push(0x14)`

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	0x102	0x101	**0x100**	0x0ff	0x0fe	0x0fd	0x0fc	0x0fb	...
...	de	ad	be	ef	13	38	14	20	...
						↑			

SP: `0x0fd`

Operation: `pop()`

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	0x102	0x101	**0x100**	0x0ff	0x0fe	0x0fd	0x0fc	0x0fb	...
...	de	ad	be	ef	13	38	14	20	...
					↑				

SP: `0x0fe`

Operation: `pop()`

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.

...	0x102	0x101	**0x100**	0x0ff	0x0fe	0x0fd	0x0fc	0x0fb	...
...	de	ad	be	ef	13	38	14	20	...
				↑					

SP: `0x0ff`

Operation: `pop()`

Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

The call stack

A program can use many stacks for various things, but when we say *the stack*, we almost alway means *the call stack*.

High level idea

- When we **call a function**, we *push* enough information to the stack to allow the called function to resume us when we are done.
- When we **return from a function**, we *pop* information from the stack to resume execution of the caller.
- Call stack is essentially a **queue of functions waiting to resume execution.**

The call stack

A program can use many stacks for various things, but when we say *the stack*, we almost alway means *the call stack*.

High level idea

- When we **call a function**, we *push* enough information to the stack to allow the called function to resume us when we are done.
- When we **return from a function**, we *pop* information from the stack to resume execution of the caller.
- Call stack is essentially a **queue of functions waiting to resume execution.**

Important

- The call stack is **not special hardware**, but uses ordinary memory.
- By RISC-V **convention**, `sp` register points to top of stack (the *stack pointer*).
- Grows from high address to low address.
  - To **pop**: *increment* `sp`.
  - To **push**: *decrement* `sp`.

Example for leaf procedure that calls no other procedures

```
leaf:
```

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

Example for leaf procedure that calls no other procedures

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12    # make room for 3 words
sw t0, 8(sp)        # save old value of t0
sw t1, 4(sp)        # save old value of t1
sw s4, 0(sp)        # save old value of s4
```

Example for leaf procedure that calls no other procedures

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
```

Example for leaf procedure that calls no other procedures

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12    # make room for 3 words
sw t0, 8(sp)        # save old value of t0
sw t1, 4(sp)        # save old value of t1
sw s4, 0(sp)        # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0      # return value in a0
```

Example for leaf procedure that calls no other procedures

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0     # return value in a0
lw s4, 0(sp)       # restore s4
lw t1, 4(sp)       # restore t1
lw t0, 8(sp)       # restore t0
addi sp, sp, 12    # pop 3 words from stack
```

Example for leaf procedure that calls no other procedures

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12     # make room for 3 words
sw t0, 8(sp)         # save old value of t0
sw t1, 4(sp)         # save old value of t1
sw s4, 0(sp)         # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0       # return value in a0
lw s4, 0(sp)         # restore s4
lw t1, 4(sp)         # restore t1
lw t0, 8(sp)         # restore t0
addi sp, sp, 12      # pop 3 words from stack
jalr zero, 0(ra)     # jump to call site
```

Procedures at machine level

- The call stack
- Calling conventions
- Recursion by example

Data representations

- Examining data in C
- A machine view of text
- Binary IO

Motivation for calling conventions

Terminology

Caller The procedure making the procedure call (jumping *from*).

Callee The procedure being called (jumping *to*).

Motivation for calling conventions

Terminology

Caller The procedure making the procedure call (jumping *from*).

Callee The procedure being called (jumping *to*).

- Large modular programs consist of procedures that
  - can be called from *any other procedure*.
  - can call *any other procedure* without knowing how they work internally.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

- **Differs between architectures and operating systems.**
  - We'll ignore lots of features that are important in practice, e.g. exception handlers, destructors, or debugging information.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

- **Differs between architectures and operating systems.**
  - We'll ignore lots of features that are important in practice, e.g. exception handlers, destructors, or debugging information.
- **Basics:**
  - Caller puts arguments in `a0`–`a7`.
  - Caller puts return address in `ra`.
  - Callee puts return value in `a0`.
- But what about the other registers?

Caller-save and callee-save registers (simplified)

Caller-save registers, e.g. `t0-t6`

From callers perspective

- *Not* preserved by procedure calls.
- Must be saved on stack before call if we want to preserve value.

From callees perspective

- May be overwritten freely.
- Can have any value when returning.

Caller-save and callee-save registers (simplified)

Caller-save registers, e.g. `t0-t6`

From callers perspective

- *Not* preserved by procedure calls.
- Must be saved on stack before call if we want to preserve value.

From callees perspective

- May be overwritten freely.
- Can have any value when returning.

Callee-save registers, e.g. `s0-s11`

From callers perspective

- Preserved by procedure calls.

From callees perspective

- Must save value on stack before overwriting.
- Must restore original value before returning.

Full table of integer registers

Register	Name	Description	Saver
`x0`	`zero`	Hard-wired zero	-
`x1`	`ra`	`Return address`	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	-
`x4`	`tp`	Thread pointer	-
`x5-7`	`t0-t2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-a1`	Arguments/return values	Caller
`x12-17`	`a2-a7`	Arguments	Caller
`x18-27`	`s2-s11`	Saved registers	Callee
`x28-31`	`t3-t6`	Temporaries	Caller

- 16 are caller-save.
- 13 are callee-save.

Leaf procedure, saving only callee-saves registers

```c
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
   int t0 = a0 + a1;
   int t1 = a2 + a3;
   int s4 = t0 - t1;
   return s4;
}
```

```
leaf:
addi sp, sp, -4      # make room for 1 word
sw s4, 0(sp)         # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0       # return value in a0
lw s4, 0(sp)         # restore s4
addi sp, sp, 4       # pop 1 word from stack
jalr zero, 0(ra)     # jump to call site
```

Leaf procedure, saving only callee-saves registers

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -4      # make room for 1 word
sw s4, 0(sp)         # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0       # return value in a0
lw s4, 0(sp)         # restore s4
addi sp, sp, 4       # pop 1 word from stack
jalr zero, 0(ra)     # jump to call site
```

Can we do even better?

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0     # first argument
addi a1, t1, 0     # second argument
addi a2, t2, 0     # third argument
addi a3, t3, 0     # fourth argument
addi sp, sp, -64   # make room for 16 words
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0       # first argument
addi a1, t1, 0       # second argument
addi a2, t2, 0       # third argument
addi a3, t3, 0       # fourth argument
addi sp, sp, -64     # make room for 16 words
sw ra, 0(sp)         # save ra
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0       # first argument
addi a1, t1, 0       # second argument
addi a2, t2, 0       # third argument
addi a3, t3, 0       # fourth argument
addi sp, sp, -64     # make room for 16 words
sw ra, 0(sp)         # save ra
sw t0, 4(sp)         # save t0
...
sw a7, 60(sp)        # save a7
jal ra, leaf         # jump to procedure
lw ra, 0(sp)         # restore ra
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, −64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
addi sp, sp, 64     # restore stack pointer
...                 # continue on
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
addi sp, sp, 64     # restore stack pointer
...                 # continue on
```

Should we restore all registers?

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0       # first argument
addi a1, t1, 0       # second argument
addi a2, t2, 0       # third argument
addi a3, t3, 0       # fourth argument
addi sp, sp, -64     # make room for 16 words
sw ra, 0(sp)         # save ra
sw t0, 4(sp)         # save t0
...
sw a7, 60(sp)        # save a7
jal ra, leaf         # jump to procedure
lw ra, 0(sp)         # restore ra
lw t0, 4(sp)         # restore t0
...
lw a7, 60(sp)        # restore a7
addi sp, sp, 64      # restore stack pointer
...                  # continue on
```

Should we restore all registers? *Possibly not* `a0` (would overwrite return value)

Procedures at machine level

- The call stack
- Calling conventions
- **Recursion by example**

Data representations

- Examining data in C
- A machine view of text
- Binary IO

```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

```
mul:
        mv a2, a0
        mv a0, zero
mul_loop:
        beq a2, zero, mul_end
        add a0, a0, a1
        addi a2, a2, -1
        jal zero, mul_loop
mul_end:
        jalr zero, ra, 0
```

```
fact:
        addi sp, sp, -8
        sw ra, 4(sp)
        sw a0, 0(sp)
        beq a0, zero, fact_base
        addi a0, a0, -1
        jal ra, fact
        lw a1, 0(sp)
        jal ra, mul
        jal zero, fact_return
fact_base:
        li a0, 1
fact_return:
        lw ra, 4(sp)
        addi sp, sp, 8
        jalr zero, ra, 0
```

Examining data representations

- **Code to print byte representation of data**
  - Casting pointer to `unsigned char*` allows treatment as byte array.

```
void show_bytes(unsigned char* start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++) {
    printf("%p\t0x%.2x\n", start+i, start[i]);
  }
  printf("\n");
}
```

`printf` directives:

- `%p`: Print pointer.
- `%x`: Print hexadecimal.

show_bytes execution example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((unsigned char*) &a, sizeof(int));
```

Result (Linux x86-64):

```
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

Procedures at machine level

- The call stack
- Calling conventions
- Recursion by example

Data representations

- Examining data in C
- A machine view of text
- Binary IO

Text IO

```
printf("Hello, world!\n");
```

Text IO

```c
printf("Hello, world!\n");
```

```
Hello, world!
```

Text IO

```
printf("Hello, world!\n");
```

```
Hello, world!
```

```
int x = 123;
printf("an integer: %d\n", x);
```

Text IO

```c
printf("Hello, world!\n");

int x = 123;
printf("an integer: %d\n", x);
```

```
Hello, world!

an integer: 123
```

Text IO

```
printf("Hello, world!\n");

int x = 123;
printf("an integer: %d\n", x);

printf("an integer: %5d\n", x);
```

```
Hello, world!

an integer: 123
```

Text IO

```
printf("Hello, world!\n");
```

```
Hello, world!
```

```
int x = 123;
printf("an integer: %d\n", x);
```

```
an integer: 123
```

```
printf("an integer: %5d\n", x);
```

```
an integer:   123
```

Text IO

```
printf("Hello, world!\n");

int x = 123;
printf("an integer: %d\n", x);

printf("an integer: %5d\n", x);

double y = 1.23;
printf("a float: %f\n", y);
```

```
Hello, world!

an integer: 123

an integer:   123
```

Text IO

```
printf("Hello, world!\n");

int x = 123;
printf("an integer: %d\n", x);

printf("an integer: %5d\n", x);

double y = 1.23;
printf("a float: %f\n", y);
```

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000
```

Text IO

```
printf("Hello, world!\n");

int x = 123;
printf("an integer: %d\n", x);

printf("an integer: %5d\n", x);

double y = 1.23;
printf("a float: %f\n", y);

printf("a mess: %d\n", y);
```

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000
```

Text IO

```
printf("Hello, world!\n");
```

```
Hello, world!
```

```
int x = 123;
printf("an integer: %d\n", x);
```

```
an integer: 123
```

```
printf("an integer: %5d\n", x);
```

```
an integer:   123
```

```
double y = 1.23;
printf("a float: %f\n", y);
```

```
a float: 1.230000
```

```
printf("a mess: %d\n", y);
```

```
a mess: 4202562
```

Text IO

```
printf("Hello, world!\n");
```

```
Hello, world!
```

```
int x = 123;
printf("an integer: %d\n", x);
```

```
an integer: 123
```

```
printf("an integer: %5d\n", x);
```

```
an integer:   123
```

```
double y = 1.23;
printf("a float: %f\n", y);
```

```
a float: 1.230000
```

```
printf("a mess: %d\n", y);
```

```
a mess: 4202562
```

Make sure format specifiers and argument types match!

Text representation

- **Machines only understand numbers, and text is an abstraction!**
- E.g. when the terminal receives a byte with the value 65, it draws an `A`.
- `printf()` determines which *bytes* must be written to the terminal to produce the text corresponding to e.g. the number `123`: `[49, 50, 51]`.

Character sets

A character set maps a *number* to a *character*.

- ASCII defines characters in the range 0—127 (`asciitable.com`).
- Some are invisible/unprintable *control characters*
- *Unicode* is a superset of ASCII that defines tens of thousands of characters for all the world's scripts.

We'll assume ASCII, which has the simple property that 1 byte = 1 character.

The ASCII table

Control characters				Normal characters											
000	nul	016	dle	032	␣	048	0	064	@	080	P	096	‘	112	p
001	soh	017	dc1	033	!	049	1	065	A	081	Q	097	a	113	q
002	stx	018	dc2	034	“	050	2	066	B	082	R	098	b	114	r
003	etx	019	dc3	035	#	051	3	067	C	083	S	099	c	115	s
004	eot	020	dc4	036	$	052	4	068	D	084	T	100	d	116	t
005	enq	021	nak	037	%	053	5	069	E	085	U	101	e	117	u
006	ack	022	syn	038	&	054	6	070	F	086	V	102	f	118	v
007	bel	023	etb	039	'	055	7	071	G	087	W	103	g	119	w
008	bs	024	can	040	(	056	8	072	H	088	X	104	h	120	x
009	tab	025	em	041	)	057	9	073	I	089	Y	105	i	121	y
010	lf	026	eof	042	*	058	:	074	J	090	Z	106	j	122	z
011	vt	027	esc	043	+	059	;	075	K	091	[	107	k	123	{
012	np	028	fs	044	,	060	<	076	L	092	␣	108	l	124	|
013	cr	029	gs	045	-	061	=	077	M	093	]	109	m	125	}
014	so	030	rs	046	.	062	>	078	N	094	^	110	n	126	~
015	si	031	us	047	/	063	?	079	O	095	_	111	o	127	del

Turning numbers into text

```
int x = 1234;
printf("x: %d\n", x);
```

Turning numbers into text

```
int x = 1234;
printf("x: %d\n", x);
```

The text *string* that is passed to `printf()` looks like this in memory:

Characters	`x`	`:`	` `	`%`	`d`	`\n`	`\0`
Bytes	120	58	32	37	100	10	0

Turning numbers into text

```
int x = 1234;
printf("x: %d\n", x);
```

The text *string* that is passed to `printf()` looks like this in memory:

Characters	x	:		%	d	\n	\0
Bytes	120	58	32	37	100	10	0

`printf()` rewrites format specifiers (`%d`) to the textual representation of their corresponding value argument:

Characters	x	:		1	2	3	4	\n	\0
Bytes	120	58	32	49	50	51	52	10	0

These bytes (except the `0`) are then written to *standard output* (typically the terminal) which interprets them as characters and eventually draws pixels on the screen.

Machine representation versus text representation

```
int x = 305419896;
```

- Written as hexadecimal (base-16), this number is `0x12345678`.
- One hexadecimal digit is 4 bit, so each group of two digits is one byte, and the number takes four bytes (32 bits).
- The *machine representation* in memory on an x86 CPU is
  `0x78 0x56 0x34 0x12`
- A *decimal text representation* in memory on *any* CPU is
  `0x33 0x30 0x35 0x34 0x31 0x39 0x38 0x39 0x36`
- Endianness has *no effect on text* (at least not with single-byte characters).
- In C, we have the additional convention that any string must be NUL-terminated.
- We identify a string with the address of its first character.

Writing bytes

The `fwrite` procedure writes raw data to an open file:

```
size_t fwrite(const void *ptr,
              size_t size,
              size_t nmemb,
              FILE *stream);
```

`ptr`: the address in memory of the data.

`size`: the size of each data element in bytes.

`nmemb`: the number of data elements.

`stream`: the target file (opened with `fopen()`).

- Returns the number of data elements written (equal to `nmemb` unless an error occurs).
- Usually no difference between writing one size `x*y` element or `x` size-`y` elements—do whatever is convenient.

Example of `fwrite()`

```
#include <stdio.h>

int main(void) {
  // Open for writing ("w")
  FILE *f = fopen("output", "w");

  char c = 42;

  fwrite(&c, sizeof(char), 1, f);

  fclose(f);
}
```

- Produces a file `output`.
- File contains the byte 42, corresponding to the ASCII character `*`.
- **`char` is just an 8-bit integer type!**
  - No special "character" meaning.
  - Most Unicode characters will not fit in a single `char` (e.g. 'æ' needs 16 bits in UTF-8).
  - Name is unfortunate/historical.
  - Signedness is *implementation-defined* for historical reasons.

Another example

```c
#include <stdio.h>

int main(void) {
  FILE *f = fopen("output", "w");

  int x = 0x53505048;
  // Stored as 0x48 0x50 0x50 0x53

  fwrite(&x, sizeof(int), 1, f);

  fclose(f);
}
```

- Writes bytes `0x48 0x50 0x50 0x53`.
- Corresponds to ASCII characters `HPPS`.
- A big-endian machine would produce `SPPH`.
- **Don't write code that depends on this!**

Converting a non-negative integer to its ASCII representation

```c
FILE *f = fopen("output", "w");
int x = 1337;               // Number to write;
char s[10];                 // Output buffer.
int i = 10;                 // Index of last character written.
while (1) {
  int d = x % 10;           // Pick out last decimal digit.
  x = x / 10;               // Remove last digit.
  i = i - 1;                // Index of next character.
  s[i] = '0' + d;           // Save ASCII character for digit.
  if (x == 0) { break; }    // Stop if all digits written.
}
fwrite(&s[i], sizeof(char), 10-i, f); // Write ASCII bytes.
fclose(f);                            // Close output file.
```

Reading bytes

```
size_t fread(void *ptr,
             size_t size,
             size_t nmemb,
             FILE *stream);
```

`ptr`: where to put the data we read.

`size`: the size of each data element in bytes.

`nmemb`: the number of data elements.

`stream`: the target file (opened with `fopen()`).

Very similar to `fwrite()`!

Reading all the bytes in a file

```c
#include <stdio.h>
#include <assert.h>

int main(int argc, char* argv[]) {
  FILE *f = fopen(argv[1], "r");
  unsigned char c;
  while (fread(&c, sizeof(char), 1, f) == 1) {
    printf("%3d ", (int)c);
    if (c > 31 && c < 127) {
      fwrite(&c, sizeof(char), 1, stdout);
    }
    printf("\n");
  }
}
```

Running `fread-bytes`

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
```

Running `fread-bytes`

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
$ ./fread-bytes fread-bytes.c
 35 #
105 i
110 n
 99 c
108 l
117 u
100 d
101 e
 32
 60 <
...
```

Running `fread-bytes`

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
```

```
$ ./fread-bytes fread-bytes.c
 35 #
105 i
110 n
 99 c
108 l
117 u
100 d
101 e
 32
 60 <
...
```

```
$ ./fread-bytes fread-bytes
127
 69 E
 76 L
 70 F
  2
  1
  1
  0
  0
  0
...
```

Text files versus binary files

- **To the system there is no difference between “text files” and “binary files”!**
- All files are just byte sequences.
- *Colloquially*: a text file is a file that is understandable when the bytes are interpreted as characters (in ASCII or some other character set).

Text files versus binary files

- **To the system there is no difference between “text files” and “binary files”!**
- All files are just byte sequences.
- *Colloquially*: a text file is a file that is understandable when the bytes are interpreted as characters (in ASCII or some other character set).

Compactness of storage

- A 32-bit integer takes up to 12 bytes to store as base-10 ASCII digits
- 4 bytes as raw data
- **Raw data takes up less space and is much faster to read.**
- But we need special programs to decode the data to human-readable form.

IO takeaways

- Use `printf()` for text output.
- (And `scanf()` for text *input*.)
- Use `fwrite()` to write raw data.
- Use `fread()` to read raw data.
- Raw data files are more compact and faster to read/write.