Internet of Things

To C or Not to C

Aslak Johansen asjo@mmmi.sdu.dk

Feb 25, 2025

SDU

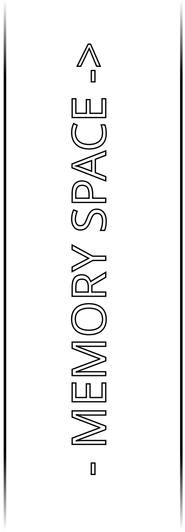
Part 1:
Introduction

Why C?

- Garbage collection is considered problematic for embedded programming
- Fast and has a low memory footprint
  (makes sense for resource constrained devices)
- Easy access to low-level operations
  - Bit-level manipulation
  - Pointer arithmetic
- Provides the most important high-level abstractions
  - Functions and a stack
  - Repetition/sequence/choice/indirection in both data and logic
- No unnecessary fluff
- Simple language
  Note: Otherwise, Rust (and Zig) would have been candidates.

Processor Model

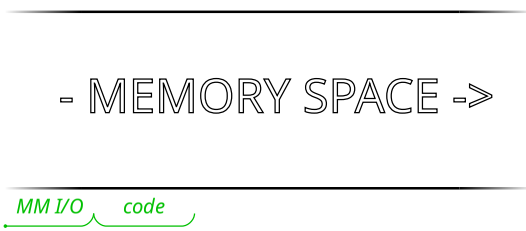
Processor Model

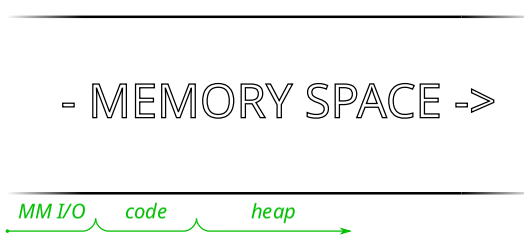


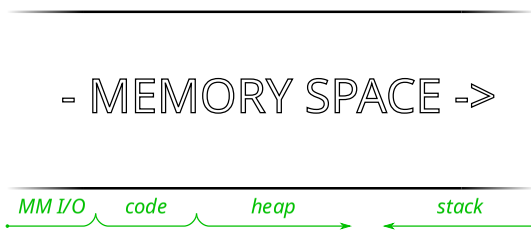
Processor Model

- MEMORY SPACE ->

MM I/O

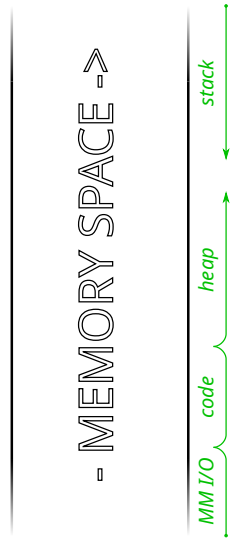

- MEMORY SPACE ->
MM I/O
code


- MEMORY SPACE ->
MM I/O
code
heap


- MEMORY SPACE ->
MM I/O
code
heap
stack

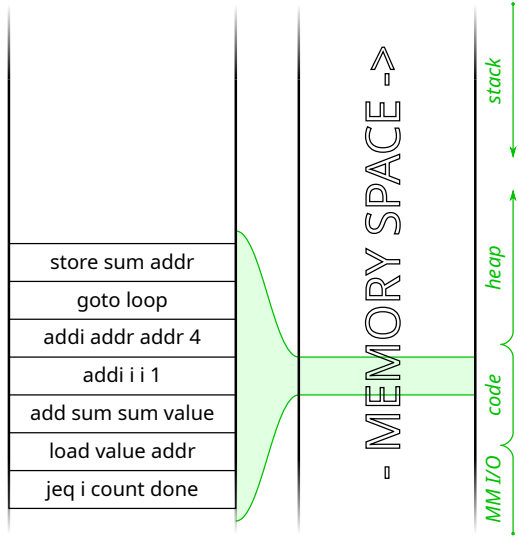
Processor Model

```
loop: jeq   i     count done
      load  value       addr
      add   sum   sum   value
      addi  i     i     1
      addi  addr  addr  4
      goto  loop
done: store sum   addr
```



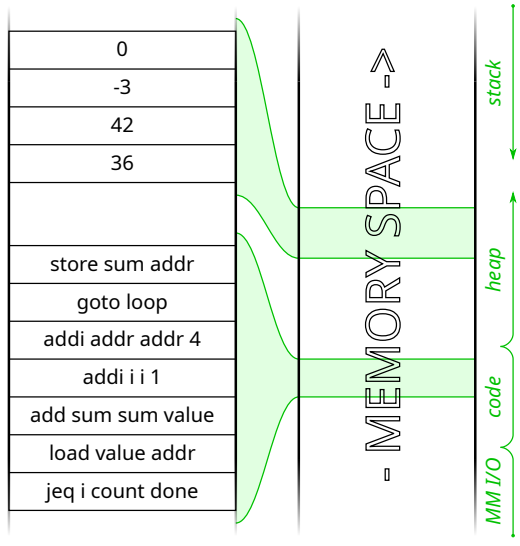
Processor Model

```
loop: jeq   i      count done
      load  value        addr
      add   sum    sum   value
      addi  i      i     1
      addi  addr   addr  4
      goto  loop
done: store sum    addr
```



Processor Model

```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

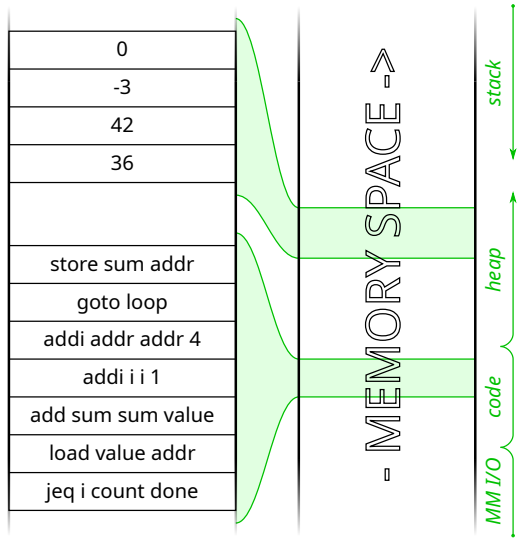


Processor Model

registers

r0:
r1:
value:
addr:
sum:
i:
count:
pc:

```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

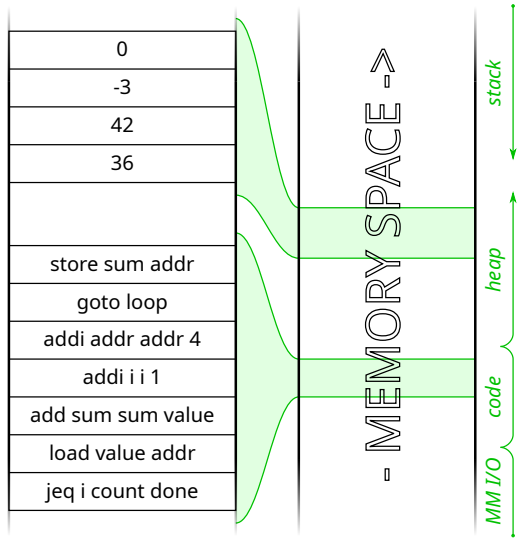


Processor Model

registers

r0:	0
r1:	0
value:	0
addr:	80
sum:	0
i:	1
count:	4
pc:	10

```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

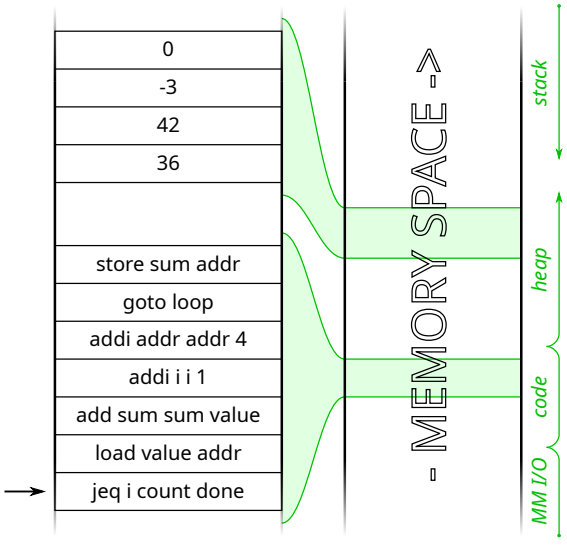


Processor Model

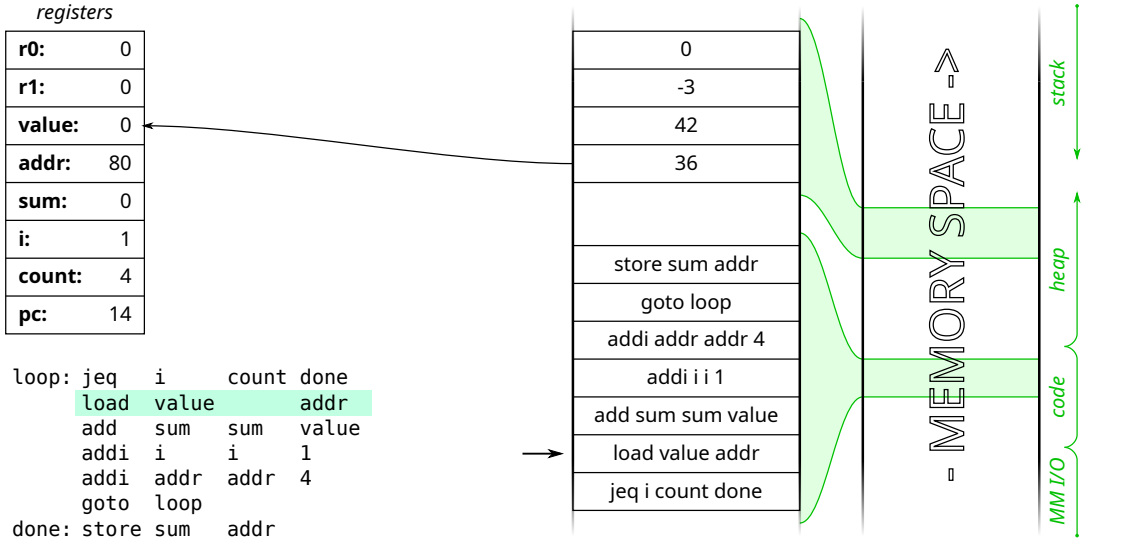
registers

r0:	0
r1:	0
value:	0
addr:	80
sum:	0
i:	1
count:	4
pc:	10

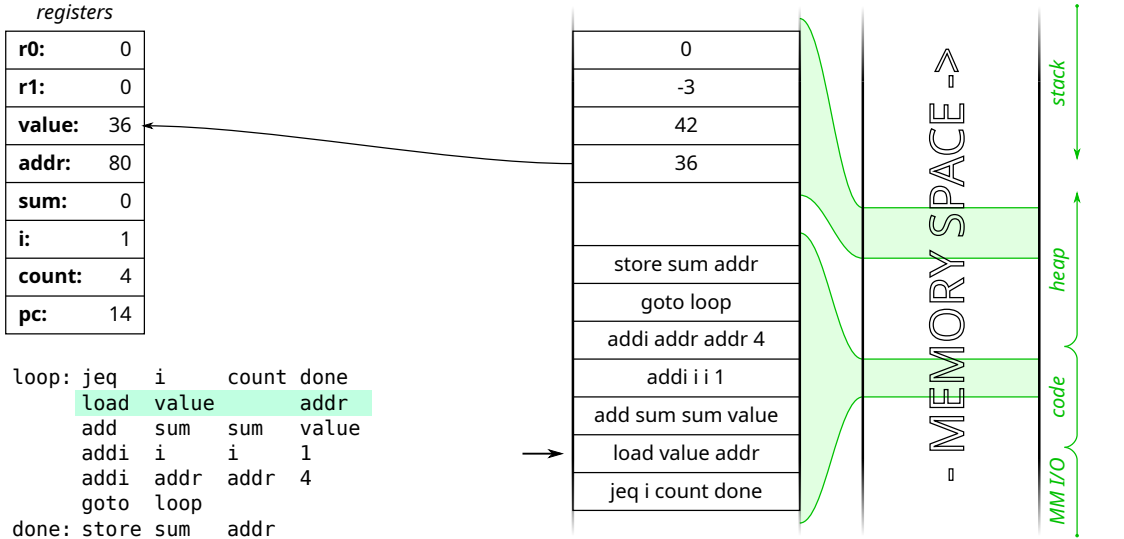

```
loop:  jeq   i      count  done
       load  value         addr
       add   sum    sum    value
       addi  i      i      1
       addi  addr   addr   4
       goto  loop
done:  store sum    addr
```



Processor Model



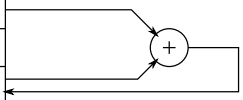
Processor Model



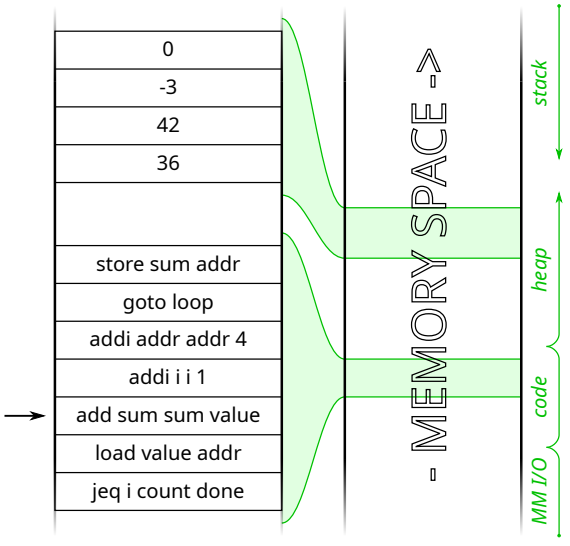
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	80
sum:	0
i:	1
count:	4
pc:	18

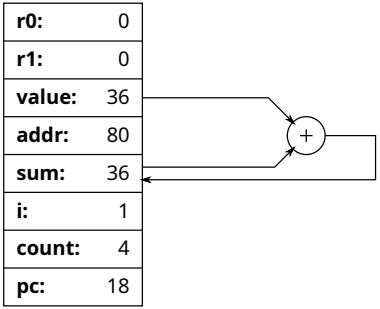



```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

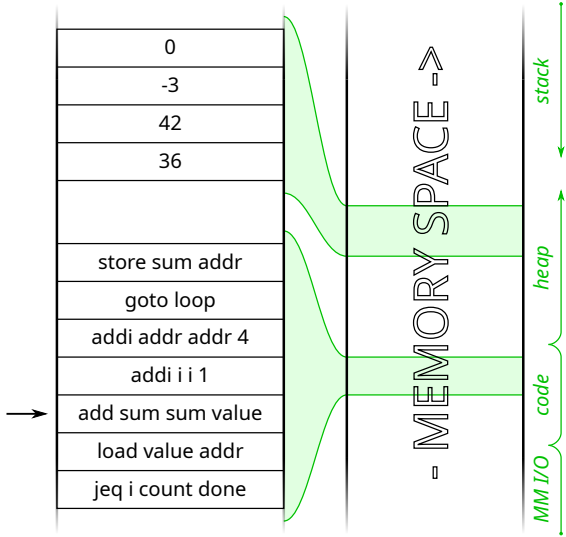


Processor Model

registers



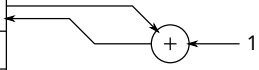

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



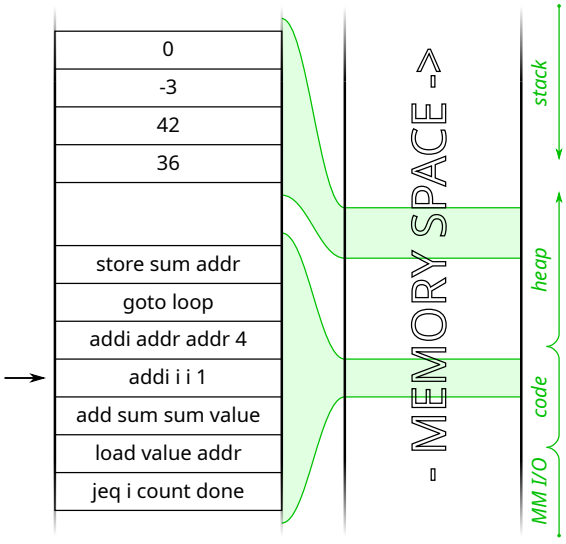
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	80
sum:	36
i:	1
count:	4
pc:	22



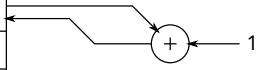

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



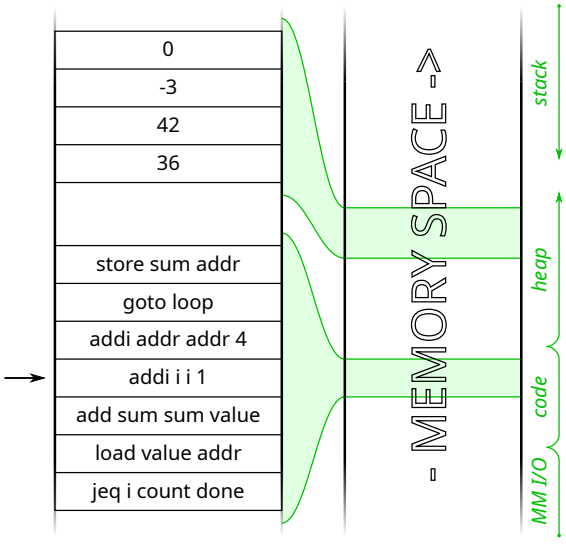
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	80
sum:	36
i:	2
count:	4
pc:	22



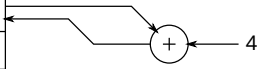

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



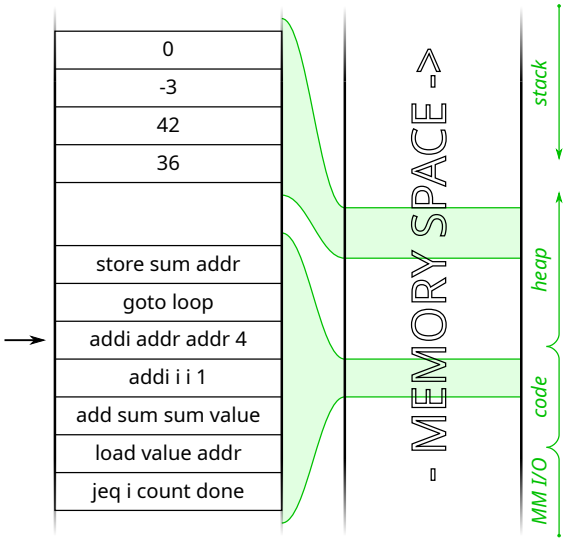
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	80
sum:	36
i:	2
count:	4
pc:	26



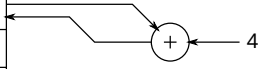

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



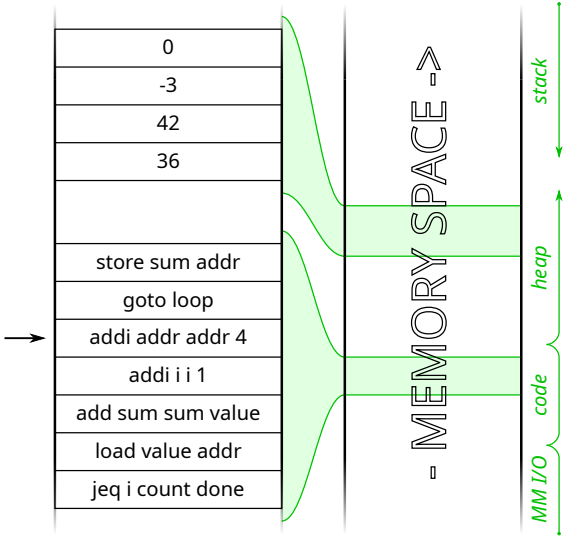
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	84
sum:	36
i:	2
count:	4
pc:	26




```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



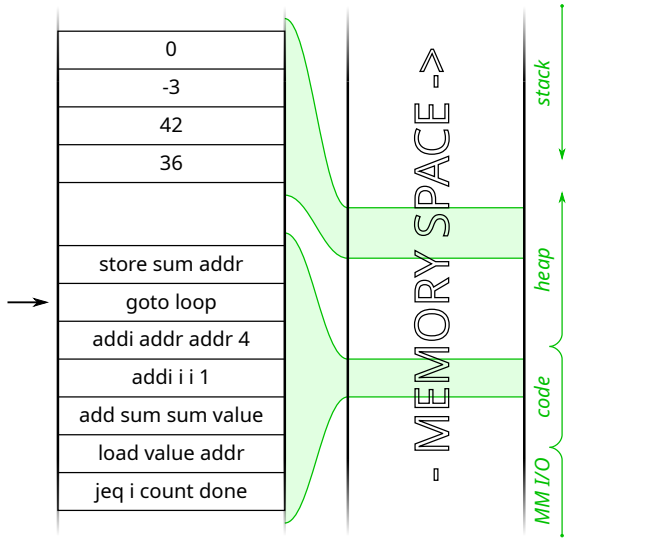
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	84
sum:	36
i:	2
count:	4
pc:	30

10

```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



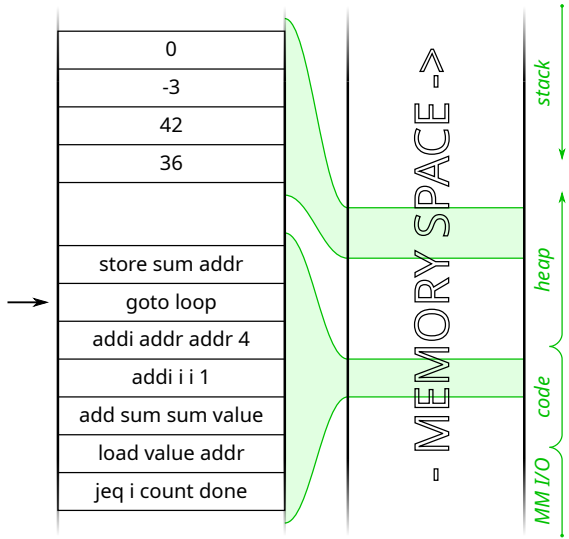
Processor Model

registers

r0:	0
r1:	0
value:	36
addr:	84
sum:	36
i:	2
count:	4
pc:	10

10

```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

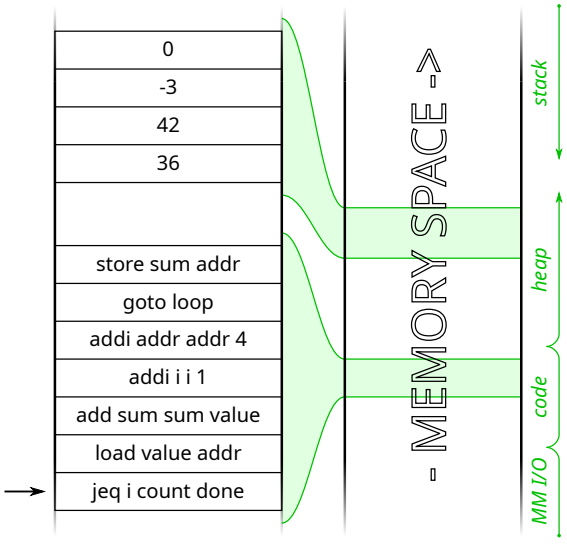


Processor Model

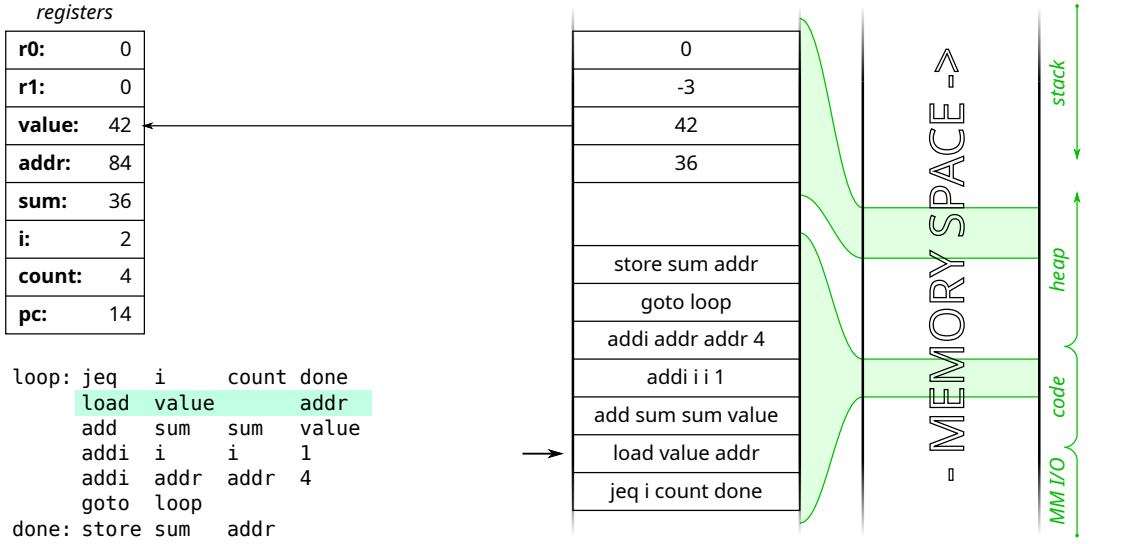
registers

r0:	0
r1:	0
value:	36
addr:	84
sum:	36
i:	2
count:	4
pc:	10


```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

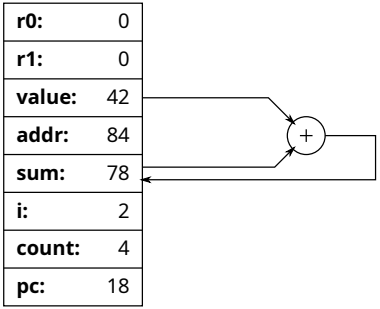


Processor Model

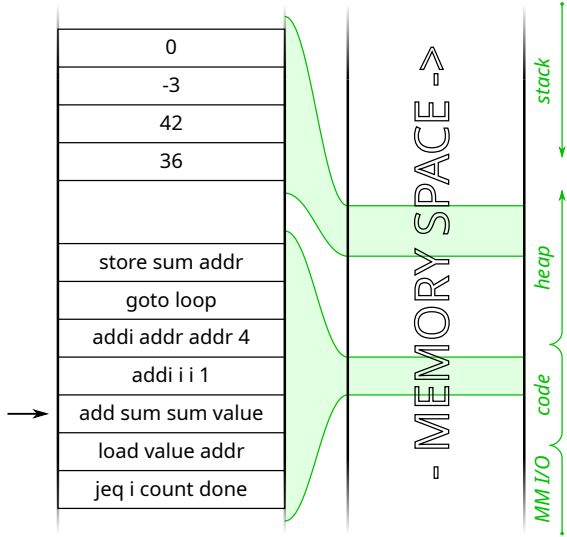


Processor Model

registers



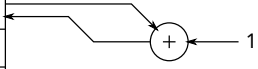

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



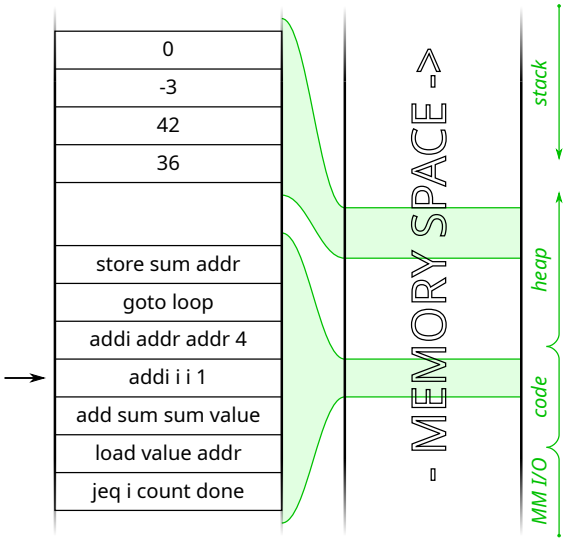
Processor Model

registers

r0:	0
r1:	0
value:	42
addr:	84
sum:	78
i:	3
count:	4
pc:	22



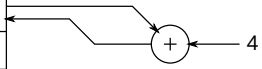

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



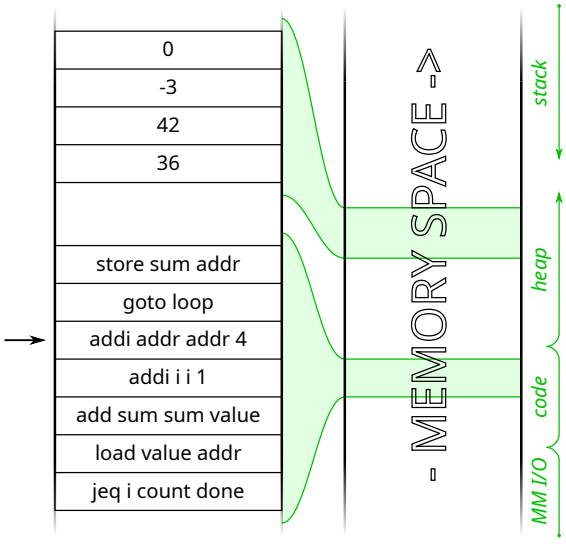
Processor Model

registers

r0:	0
r1:	0
value:	42
addr:	88
sum:	78
i:	3
count:	4
pc:	26




```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

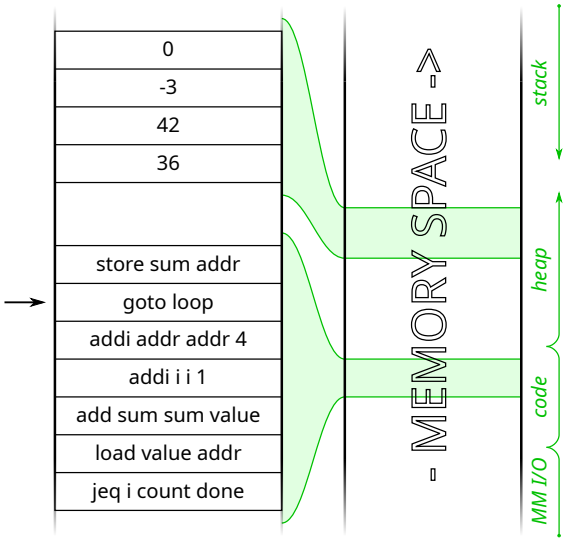


Processor Model

registers

r0:	0
r1:	0
value:	42
addr:	88
sum:	78
i:	3
count:	4
pc:	10


```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

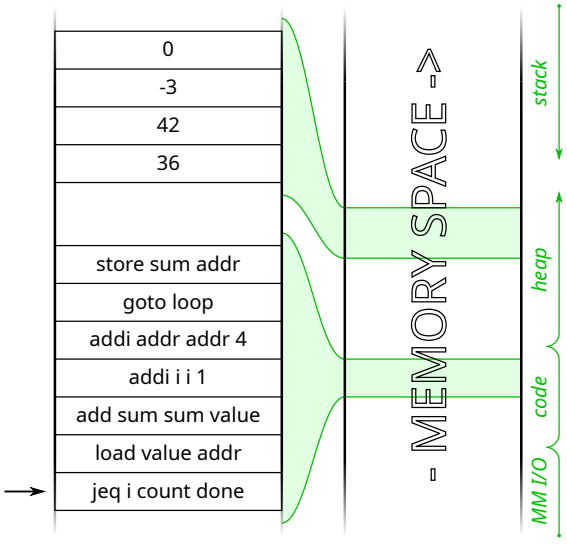


Processor Model

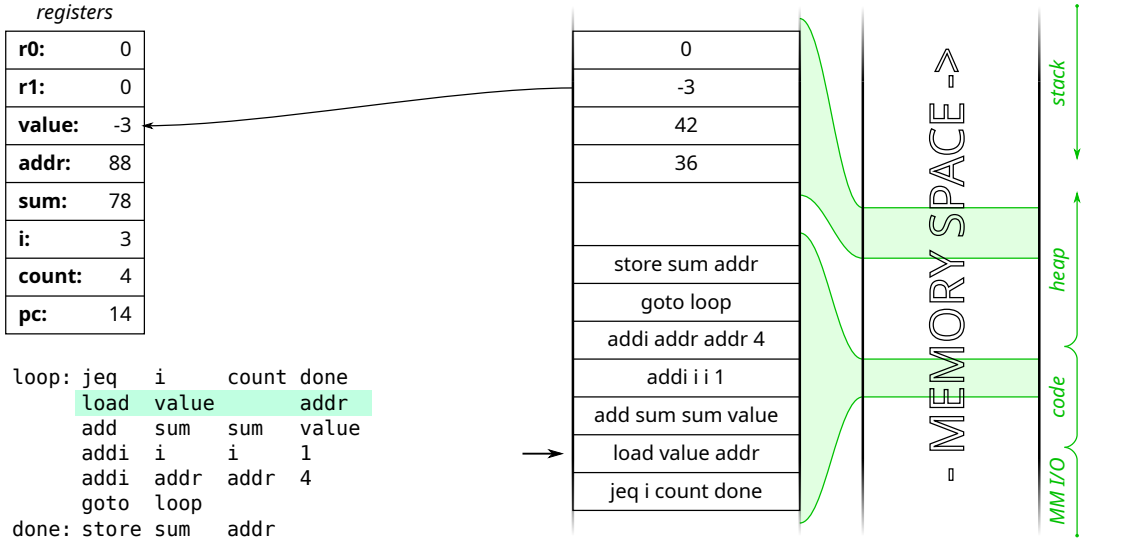
registers

r0:	0
r1:	0
value:	42
addr:	88
sum:	78
i:	3
count:	4
pc:	10


```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

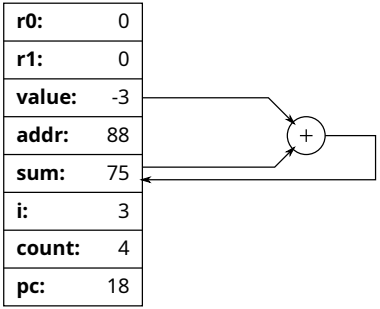


Processor Model

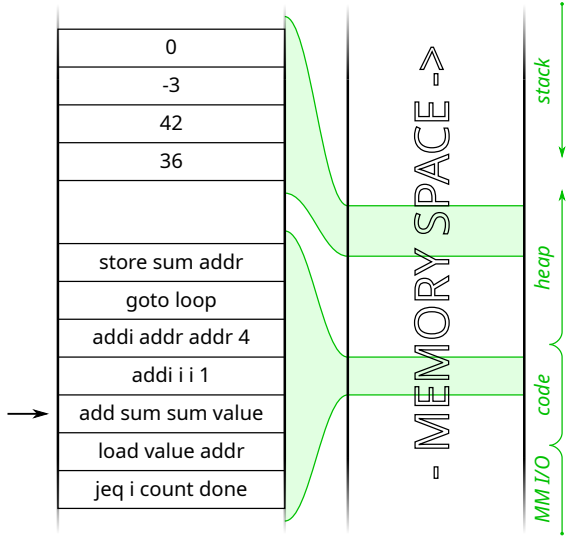


Processor Model

registers



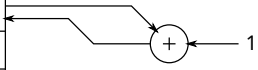

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



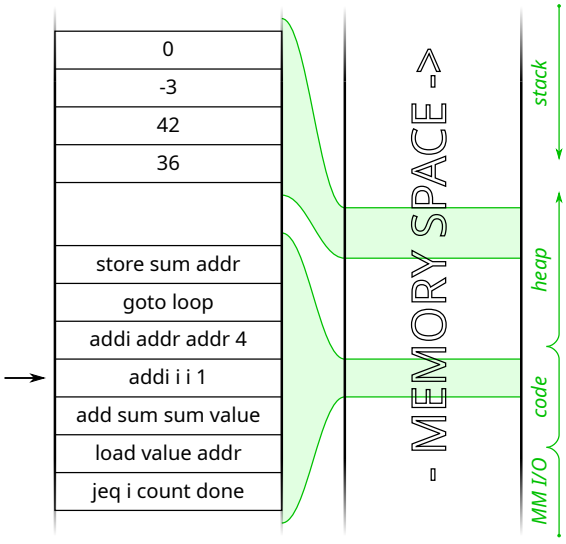
Processor Model

registers

r0:	0
r1:	0
value:	-3
addr:	88
sum:	75
i:	4
count:	4
pc:	22



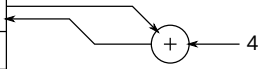

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```



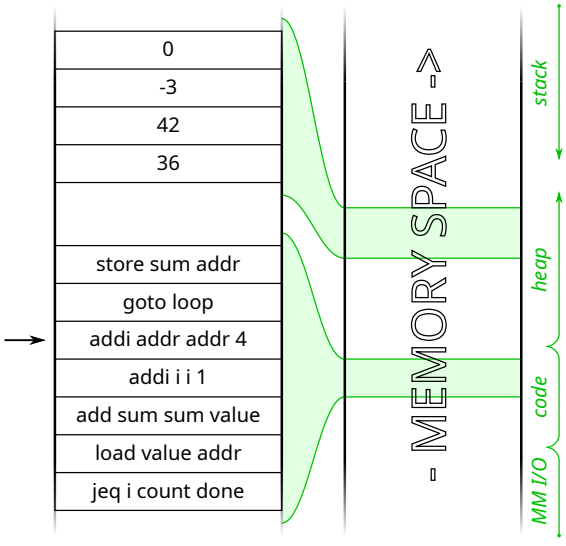
Processor Model

registers

r0:	0
r1:	0
value:	-3
addr:	92
sum:	75
i:	4
count:	4
pc:	26




```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

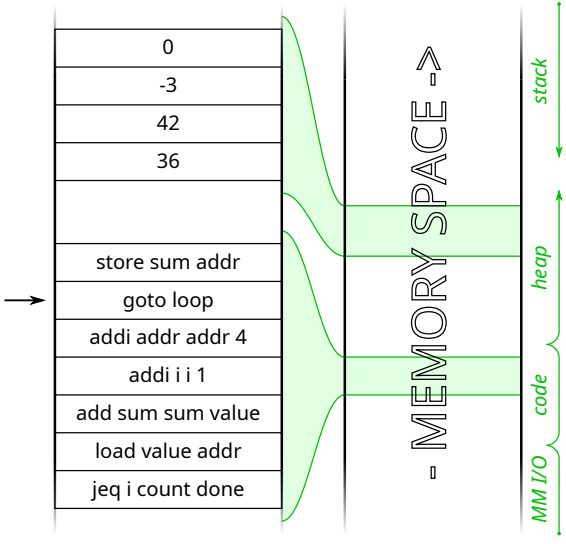


Processor Model

registers

r0:	0
r1:	0
value:	-3
addr:	92
sum:	75
i:	4
count:	4
pc:	10


```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

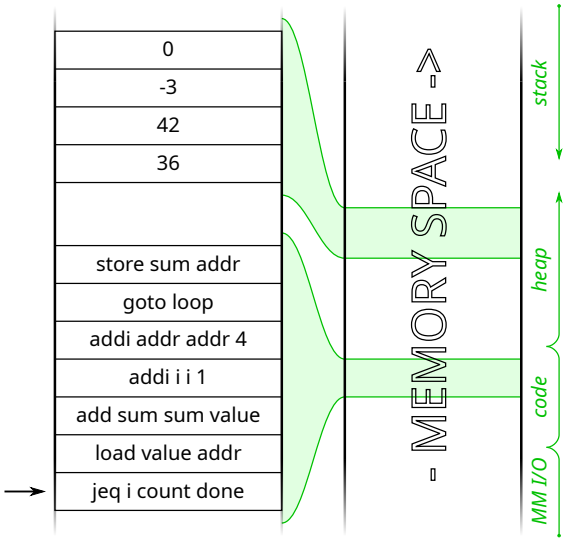


Processor Model

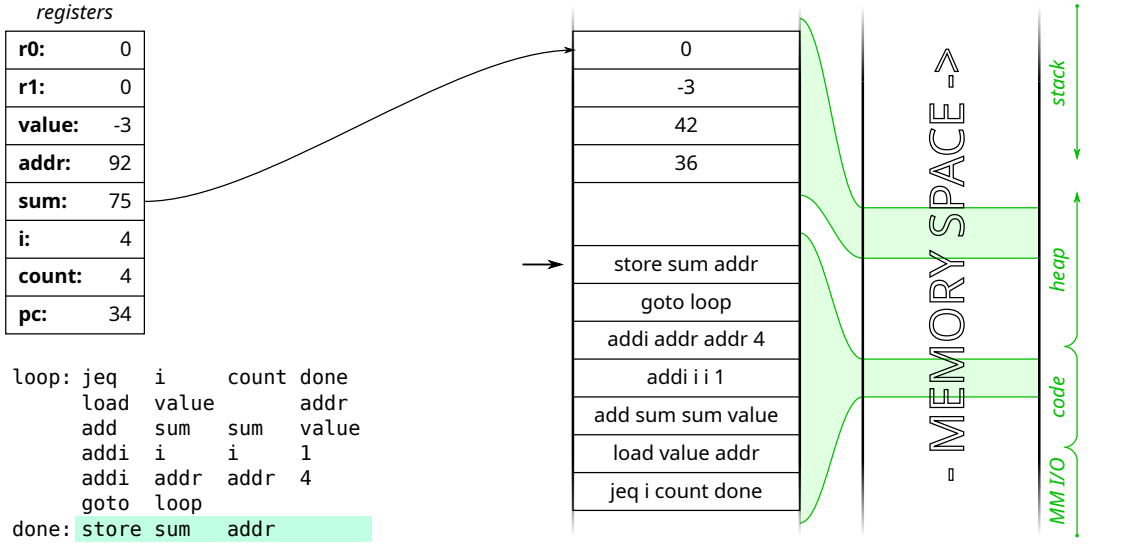
registers

r0:	0
r1:	0
value:	-3
addr:	92
sum:	75
i:	4
count:	4
pc:	10


```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

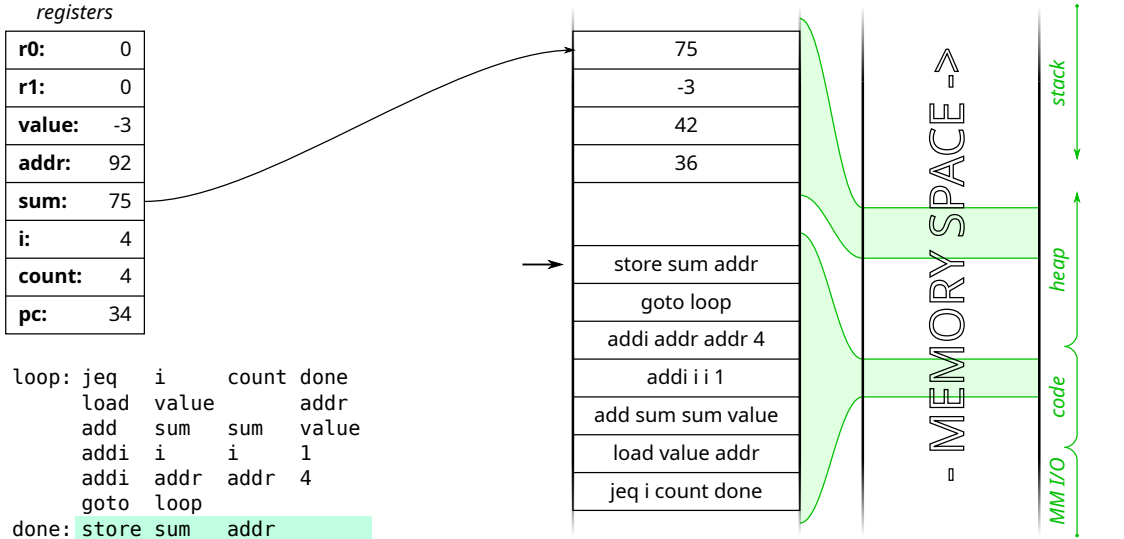


Processor Model




```
loop: jeq   i      count done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

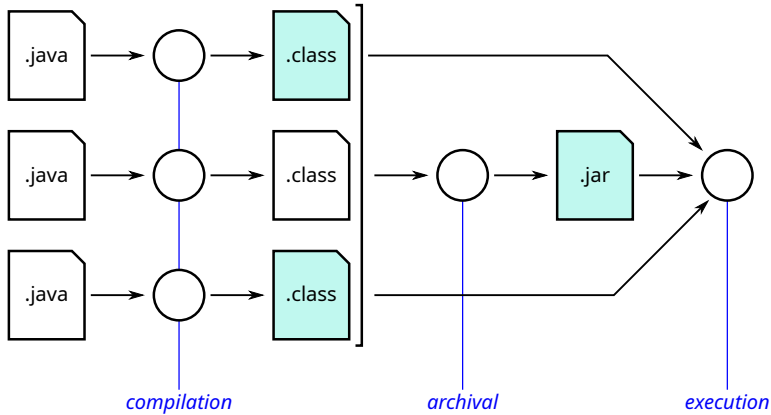
Processor Model



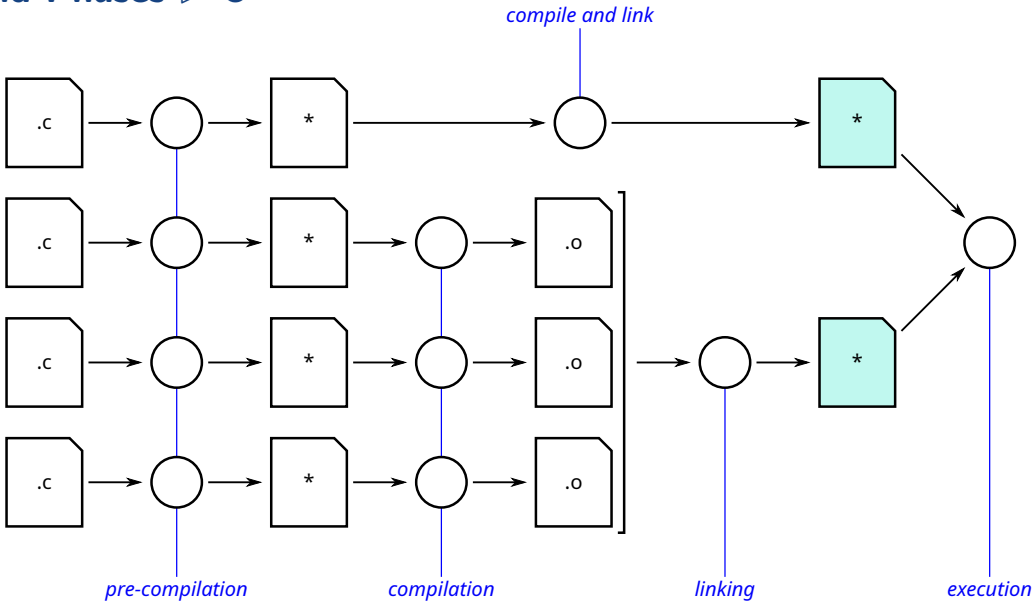

```
loop: jeq   i      count  done
      load  value         addr
      add   sum    sum    value
      addi  i      i      1
      addi  addr   addr   4
      goto  loop
done: store sum    addr
```

Build Phases

Build Phases ▷ Java



Build Phases ▷ C



Basic Abstractions

Property	Data	Instruction
Repetition:	array	while + do-while + for
Sequence:	struct	; + function
Choice:	union	if + switch
Indirection:	pointer	pointer

Simple types:

- ***Integers:*** **sizeof(`char`) $\leq$ sizeof(`short`) $\leq$ sizeof(`int`) $\leq$ sizeof(`long`)**
- ***Floats:*** **sizeof(`float`) $\leq$ sizeof(`double`)**
- ***Booleans:*** Just integers.

Complex types:

- ***Strings:*** Just zero-terminated arrays of `char`.

Formatted Printouts

We will be relying heavily on `printf` for formatted printouts:

- It uses the ellipsis operator to support an arbitrary number of arguments.
- The first argument is a `char*` template for how to format the remaining arguments.
- Variants exists. For instance, `sprintf` does the same, except that it outputs to a `char*`.

Documentation of the template format:

`https://www.gnu.org/software/libc/manual/html_node/Table-of-Output-Conversions.html`

Part 2:
Constructs in Logic

Branching

```
if (i%2==0) {
  printf("even\n");
} else {
  printf("odd\n");
}

int even = i%2==0;
printf((even ? "even" : "odd"));
```

Switch Statements

```
switch (input) {
case NORTH:
  printf("going north\n");
  break;
case SOUTH:
  printf("going south\n");
  break;
case EAST:
  printf("going east\n");
  break;
case WEST:
  printf("going west\n");
  break;
default:
  printf("feeling confused\n");
}
```

Loops

```c
for (int i=0 ; i<10 ; i++) {
  printf("%d\n", i);
}

int i = 0;
while (i++ < 10) {
  printf("%d\n", i);
}

i = 0;
do {
  printf("%d\n", i);
} while (i++ < 10);
```

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Goto Statements Considered Useful

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUF_SIZE (16)
#define INDEX    (12)

int main (int argc, char* argv[]) {
  char* filename = argv[1];
  char* buf = (char*) malloc(BUF_SIZE*sizeof(char));
  if (!buf) goto failure;

  FILE* f = fopen(filename, "r");
  if (!f) goto failure;

  size_t actual = fread(buf, sizeof(char), BUF_SIZE, f);
  if (actual!=BUF_SIZE) goto failure;

//  for (int i=0 ; i<BUF_SIZE ; i++) {
//    printf("%2d -> '%c'\n", i, buf[i]);
//  }

  printf("'%s'[%d] = '%c'\n", filename, INDEX, buf[INDEX]);

  if (fclose(f)) goto failure;

  free(buf);
  return 0;

failure:
  printf("Oops, something went wrong with error %d.\n", errno);
  return 1;
}
```

'goto_rop.c'[12] = 'd'

Part 3:
Constructs in Data

Defining Types

```c
#include <stdio.h>

typedef unsigned short uint16_t;

int main (int argc, char* argv[]) {
  uint16_t var = 42;
  printf("sizeof(uint16_t)=%ld\n", sizeof(uint16_t));

  printf("42 -> %d\n", var);
  var = 0;
  printf("0  -> %d\n", var);
  var--;
  printf("-1 -> %d\n", var);

  return 0;
}
```

```
sizeof(uint16_t)=2
42 -> 42
0  -> 0
-1 -> 65535
```

Enums

```c
#include <stdio.h>

typedef enum {
  CAT_UNKNOWN,
  CAT_STRING,
  CAT_INTEGER = 10,
  CAT_FLOAT,
} category_t;

category_t get_category (char* input) {
  category_t state = CAT_UNKNOWN;

  for (char c ; c = *input ; input++) {
    switch (state) {
    case CAT_UNKNOWN:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (c<='9' && c>='0') {
        state = CAT_INTEGER;
      } else {
        state = CAT_STRING;
      }
      break;
    case CAT_INTEGER:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (!(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_FLOAT:
      if (c=='.' || !(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_STRING:
      break;
    default:
      printf("Error, unknown state: %d\n", state);
    }
  }

  return state;
}

int main (int argc, char* argv[]) {
  char* inputs[] = {
    "10.h",
    "10.7",
    "10",
    "Hello, world"
  };

  printf("UNK=%d STR=%d INT=%d FLOAT=%d\n",
         CAT_UNKNOWN, CAT_STRING, CAT_INTEGER, CAT_FLOAT);
  for (int i=0 ; i<4 ; i++) {
    char* input = inputs[i];
    category_t category = get_category(input);
    printf("category(\"%s\") -> %d\n", input, category);
  }

  return 0;
}
```

Enums

```c
#include <stdio.h>

typedef enum {
  CAT_UNKNOWN,
  CAT_STRING,
  CAT_INTEGER = 10,
  CAT_FLOAT,
} category_t;

category_t get_category (char* input) {
  category_t state = CAT_UNKNOWN;

  for (char c ; c = *input ; input++) {
    switch (state) {
    case CAT_UNKNOWN:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (c<='9' && c>='0') {
        state = CAT_INTEGER;
      } else {
        state = CAT_STRING;
      }
      break;
    case CAT_INTEGER:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (!(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_FLOAT:
      if (c=='.' || !(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_STRING:
      break;
    default:
      printf("Error, unknown state: %d\n", state);
    }
  }

  return state;
}

int main (int argc, char* argv[]) {
  char* inputs[] = {
    "10.h",
    "10.7",
    "10",
    "Hello, world"
  };

  printf("UNK=%d STR=%d INT=%d FLOAT=%d\n",
         CAT_UNKNOWN, CAT_STRING, CAT_INTEGER, CAT_FLOAT);
  for (int i=0 ; i<4 ; i++) {
    char* input = inputs[i];
    category_t category = get_category(input);
    printf("category(\"%s\") -> %d\n", input, category);
  }

  return 0;
}
```

Enums

```
UNK=0 STR=1 INT=10 FLOAT=11
category("10.h") -> 1
category("10.7") -> 11
category("10") -> 10
category("Hello, world") -> 1
```

```c
#include <stdio.h>

typedef enum {
  CAT_UNKNOWN,
  CAT_STRING,
  CAT_INTEGER = 10,
  CAT_FLOAT,
} category_t;

category_t get_category (char* input) {
  category_t state = CAT_UNKNOWN;

  for (char c ; c = *input ; input++) {
    switch (state) {
    case CAT_UNKNOWN:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (c<='9' && c>='0') {
        state = CAT_INTEGER;
      } else {
        state = CAT_STRING;
      }
      break;
    case CAT_INTEGER:
      if (c=='.') {
        state = CAT_FLOAT;
      } else if (!(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_FLOAT:
      if (c=='.' || !(c<='9' && c>='0')) {
        state = CAT_STRING;
      }
      break;
    case CAT_STRING:
      break;
    default:
      printf("Error, unknown state: %d\n", state);
    }
  }

  return state;
}

int main (int argc, char* argv[]) {
  char* inputs[] = {
    "10.h",
    "10.7",
    "10",
    "Hello, world"
  };

  printf("UNK=%d STR=%d INT=%d FLOAT=%d\n",
         CAT_UNKNOWN, CAT_STRING, CAT_INTEGER, CAT_FLOAT);
  for (int i=0 ; i<4 ; i++) {
    char* input = inputs[i];
    category_t category = get_category(input);
    printf("category(\"%s\") -> %d\n", input, category);
  }

  return 0;
}
```

Unions

```c
#include <stdio.h>

typedef union {
  double reading;
  int    count;
} payload_t;

void print (payload_t payload) {
  printf("payload ");
  for (unsigned long i=1 ; i!=0 ; i<<=1) {
    printf("%d", !!(*((long*)&payload) & i));
  }
  printf(" reading=%lf count=%10u\n", payload.reading, payload.count);
}

int main (int argc, char* argv[]) {
  payload_t p = {.reading = 3.14};
  printf("sizeof(payload_t)=%ld sizeof(double)=%ld sizeof(int)=%ld\n", sizeof(payload_t), sizeof(double), sizeof(int));

  print(p);
  p.count = 42;
  print(p);
  p.reading = 2.718;
  print(p);

  return 0;
}
```

Unions

```c
#include <stdio.h>

typedef union {
  double reading;
  int    count;
} payload_t;

void print (payload_t payload) {
  printf("payload ");
  for (unsigned long i=1 ; i!=0 ; i<<=1) {
    printf("%d", !!(*((long*)&payload) & i));
  }
  printf(" reading=%lf count=%10u\n", payload.reading, payload.count);
}

int main (int argc, char* argv[]) {
  payload_t p = {.reading = 3.14};
  printf("sizeof(payload_t)=%ld sizeof(double)=%ld sizeof(int)=%ld\n", sizeof(payload_t), sizeof(double), sizeof(int));

  print(p);
  p.count = 42;
  print(p);
  p.reading = 2.718;
  print(p);

  return 0;
}
```

```
sizeof(payload_t)=8 sizeof(double)=8 sizeof(int)=4
payload 11111000101000011101011110001010000111010111100010010000000010 reading=3.140000 count=1374389535
payload 01010100000000000000000000000000000111010111100010010000000010 reading=3.139999 count=        42
payload 00011010100111000010110100010011011011100111101101000000000010 reading=2.718000 count=3367254360
```

Structs

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LENGTH (10)

typedef struct {
  double x;
  double y;
} point_t;

double dist (point_t a, point_t b) {
  double xdiff = b.x - a.x;
  double ydiff = b.y - a.y;
  return sqrt(xdiff*xdiff + ydiff*ydiff);
}

int main (int argc, char* argv[]) {
  point_t  p0 = {0,0};
  point_t  p1 = {1,1};
  point_t* ps = malloc(LENGTH*sizeof(point_t));

  for (int i=0 ; i<LENGTH ; i++) {
    ps[i].x = i;
    ps[i].y = 1;
  }

  printf("distance(p0, p1) = %5.3f\n", dist(p0, p1));
  for (int i=0 ; i<LENGTH ; i++)
    printf("distance(p0, ps[%u]) = %5.3f\n", i, dist(p0, ps[i]));

  return 0;
}
```

Structs

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LENGTH (10)

typedef struct {
  double x;
  double y;
} point_t;

double dist (point_t a, point_t b) {
  double xdiff = b.x - a.x;
  double ydiff = b.y - a.y;
  return sqrt(xdiff*xdiff + ydiff*ydiff);
}

int main (int argc, char* argv[]) {
  point_t  p0 = {0,0};
  point_t  p1 = {1,1};
  point_t* ps = malloc(LENGTH*sizeof(point_t));

  for (int i=0 ; i<LENGTH ; i++) {
    ps[i].x = i;
    ps[i].y = 1;
  }

  printf("distance(p0, p1) = %5.3f\n", dist(p0, p1));
  for (int i=0 ; i<LENGTH ; i++)
    printf("distance(p0, ps[%u]) = %5.3f\n", i, dist(p0, ps[i]));

  return 0;
}
```

Structs

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LENGTH (10)

typedef struct {
  double x;
  double y;
} point_t;

double dist (point_t a, point_t b) {
  double xdiff = b.x - a.x;
  double ydiff = b.y - a.y;
  return sqrt(xdiff*xdiff + ydiff*ydiff);
}

int main (int argc, char* argv[]) {
  point_t  p0 = {0,0};
  point_t  p1 = {1,1};
  point_t* ps = malloc(LENGTH*sizeof(point_t));

  for (int i=0 ; i<LENGTH ; i++) {
    ps[i].x = i;
    ps[i].y = 1;
  }

  printf("distance(p0, p1) = %5.3f\n", dist(p0, p1));
  for (int i=0 ; i<LENGTH ; i++)
    printf("distance(p0, ps[%u]) = %5.3f\n", i, dist(p0, ps[i]));

  return 0;
}
```

Structs

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LENGTH (10)

typedef struct {
  double x;
  double y;
} point_t;

double dist (point_t a, point_t b) {
  double xdiff = b.x - a.x;
  double ydiff = b.y - a.y;
  return sqrt(xdiff*xdiff + ydiff*ydiff);
}

int main (int argc, char* argv[]) {
  point_t  p0 = {0,0};
  point_t  p1 = {1,1};
  point_t* ps = malloc(LENGTH*sizeof(point_t));

  for (int i=0 ; i<LENGTH ; i++) {
    ps[i].x = i;
    ps[i].y = 1;
  }

  printf("distance(p0, p1) = %5.3f\n", dist(p0, p1));
  for (int i=0 ; i<LENGTH ; i++)
    printf("distance(p0, ps[%u]) = %5.3f\n", i, dist(p0, ps[i]));

  return 0;
}
```

```
distance(p0, p1) = 1.414
distance(p0, ps[0]) = 1.000
distance(p0, ps[1]) = 1.414
distance(p0, ps[2]) = 2.236
distance(p0, ps[3]) = 3.162
distance(p0, ps[4]) = 4.123
distance(p0, ps[5]) = 5.099
distance(p0, ps[6]) = 6.083
distance(p0, ps[7]) = 7.071
distance(p0, ps[8]) = 8.062
distance(p0, ps[9]) = 9.055
```

Dynamic Memory Allocation

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

typedef struct {
  double x;
  double y;
} point_t;

point_t* point_new (double x, double y) {
  point_t* point = (point_t*) malloc(sizeof(*point));
  point->x = x;
  point->y = y;
  return point;
}

void point_destroy (point_t* point) {
  free(point);
}

void point_print (point_t* point) {
  printf("(point x=%f y=%f)\n", point->x, point->y);
}

int main (int argc, char* argv[]) {
  point_t* point = point_new(3.14, 2.7);
  point_print(point);
  point_destroy(point);

  return 0;
}
```

Dynamic Memory Allocation

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

typedef struct {
  double x;
  double y;
} point_t;

point_t* point_new (double x, double y) {
  point_t* point = (point_t*) malloc(sizeof(*point));
  point->x = x;
  point->y = y;
  return point;
}

void point_destroy (point_t* point) {
  free(point);
}

void point_print (point_t* point) {
  printf("(point x=%f y=%f)\n", point->x, point->y);
}

int main (int argc, char* argv[]) {
  point_t* point = point_new(3.14, 2.7);
  point_print(point);
  point_destroy(point);

  return 0;
}
```

```
(point x=3.140000 y=2.700000)
```

Arrays

```c
#include <stdio.h>

#define LENGTH (10)

int main (int argc, char* argv[]) {
  int values[LENGTH];

  for (int i=0 ; i<LENGTH ; i++) {
    printf("%u: %d\n", i, values[i]);
  }

  return 0;
}
```

```
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 88076336
7: 32755
8: 0
9: 0
```

Pointers

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

#define LENGTH (10)

int main (int argc, char* argv[]) {
  int* values = (int*) malloc(LENGTH*sizeof(int));

  for (int i=0 ; i<LENGTH ; i++) {
//    printf("%u: %d\n", i, *(values+i));
//    printf("%u: %d\n", i, values[i]);
    printf("%u: %d\n", i, *(values++));
  }

  return 0;
}
```

```
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
```

Pointers to Functions

```c
#include <stdio.h>

typedef int (*int2int_function)(int);

int incr (int i) { return i+1; }

void map (int data[], int length, int2int_function fun) {
  for (int i=0 ; i<length ; i++) {
    data[i] = fun(data[i]);
  }
}

void print(int* data, int length) {
  for (int i=0 ; i<length ; i++) {
    printf("%2d: %2d\n", i, data[i]);
  }
}

int main (int argc, char* argv[]) {
  int data[] = {0,1,2,3,4,5,6,7,8,9,10};

  printf("Initial data:\n");
  print(data, 11);

  map(data, 11, &incr);

  printf("\nMapped data:\n");
  print(data, 11);

  return 0;
}
```

Pointers to Functions

```c
#include <stdio.h>

typedef int (*int2int_function)(int);

int incr (int i) { return i+1; }

void map (int data[], int length, int2int_function fun) {
  for (int i=0 ; i<length ; i++) {
    data[i] = fun(data[i]);
  }
}

void print(int* data, int length) {
  for (int i=0 ; i<length ; i++) {
    printf("%2d: %2d\n", i, data[i]);
  }
}

int main (int argc, char* argv[]) {
  int data[] = {0,1,2,3,4,5,6,7,8,9,10};

  printf("Initial data:\n");
  print(data, 11);

  map(data, 11, &incr);

  printf("\nMapped data:\n");
  print(data, 11);

  return 0;
}
```

Pointers to Functions

```c
#include <stdio.h>

typedef int (*int2int_function)(int);

int incr (int i) { return i+1; }

void map (int data[], int length, int2int_function fun) {
  for (int i=0 ; i<length ; i++) {
    data[i] = fun(data[i]);
  }
}

void print(int* data, int length) {
  for (int i=0 ; i<length ; i++) {
    printf("%2d: %2d\n", i, data[i]);
  }
}

int main (int argc, char* argv[]) {
  int data[] = {0,1,2,3,4,5,6,7,8,9,10};

  printf("Initial data:\n");
  print(data, 11);

  map(data, 11, &incr);

  printf("\nMapped data:\n");
  print(data, 11);

  return 0;
}
```

```
Initial data:
 0:  0
 1:  1
 2:  2
 3:  3
 4:  4
 5:  5
 6:  6
 7:  7
 8:  8
 9:  9
10: 10

Mapped data:
 0:  1
 1:  2
 2:  3
 3:  4
 4:  5
 5:  6
 6:  7
 7:  8
 8:  9
 9: 10
10: 11
```

Pointer Overview

- A pointer variable contains only an address.
- An address is an integer.
- The address of something can be obtained using the `&` operator.
- The data at some address can be obtained using the `*` operator (we say that we follow or *"dereference"* the pointer).
- A field of a struct or union type can be accessed through the `.` operator.
- A field of a struct or union pointer type can be accessed through the `->` operator.
- A pointer type has a size (in bytes). Incrementing the pointer by n advances it by n steps of this size.
- Any pointer type can be converted to any other pointer type by going over (i.e., casting) the `void*` type. This does not change the value.
- Following a pointer to a location in memory outside your allocations will trigger a segmentation fault (aka *segfault*)[†].

Part 4:
Precompiler Directives

Defines

```c
#include <stdio.h>

#define DEFAULT (12)
#define RADIUS2DIAMETER(r) ((r)*2)

int main (int argc, char* argv[]) {
  int radius = DEFAULT*MULTIPLIER;
  int diameter = RADIUS2DIAMETER(radius);
  printf("r=%d => d=%d\n", radius, diameter);

  return 0;
}
```

Defines

```c
#include <stdio.h>

#define DEFAULT (12)
#define RADIUS2DIAMETER(r) ((r)*2)

int main (int argc, char* argv[]) {
  int radius = DEFAULT*MULTIPLIER;
  int diameter = RADIUS2DIAMETER(radius);
  printf("r=%d => d=%d\n", radius, diameter);

  return 0;
}
```

Defines

```c
#include <stdio.h>

#define DEFAULT (12)
#define RADIUS2DIAMETER(r) ((r)*2)

int main (int argc, char* argv[]) {
  int radius = DEFAULT*MULTIPLIER;
  int diameter = RADIUS2DIAMETER(radius);
  printf("r=%d => d=%d\n", radius, diameter);

  return 0;
}
```

```
$ gcc -O0 -Wpedantic -DMULTIPLIER=3 define.c -o define
$ ./define
r=36 => d=72
$
```

Ifdefs

```c
#include <stdio.h>
#include <sys/time.h>

#define ITERATION_COUNT (1<<30)

double time_diff (struct timeval t0, struct timeval t1)
{
  double t0_us = (double)t0.tv_sec * 1000000 + (double)t0.tv_usec;
  double t1_us = (double)t1.tv_sec * 1000000 + (double)t1.tv_usec;
  return (t1_us - t0_us)/1000000;
}

int main (int argc, char* argv[]) {
#ifdef BENCHMARK
  struct timeval t0, t1;
  gettimeofday(&t0, NULL);
#endif

  for (long i=0 ; i<ITERATION_COUNT ; i++) ;

#ifdef BENCHMARK
  gettimeofday(&t1, NULL);
  printf("Performed %u iterations in %fs\n", ITERATION_COUNT, time_diff(t0, t1));
#endif

  return 0;
}
```

```
$ gcc -O0 -Wpedantic ifdef.c -o ifdef
$ ./ifdef
$ gcc -O0 -Wpedantic -DBENCHMARK ifdef.c -o ifdef_test
$ ./ifdef_test
Performed 1073741824 iterations in 2.163524s
$
```

Includes

```c
#ifndef __POINT_H
#define __POINT_H

typedef struct {
  double x;
  double y;
} point_t;

point_t* point_new (double x, double y);
void point_destroy (point_t* point);
void point_print (point_t* point);

#endif
```

```c
#include "point.h"

int main (int argc, char* argv[]) {
  point_t* point = point_new(3.14, 2.7);
  point_print(point);
  point_destroy(point);

  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "point.h"

point_t* point_new (double x, double y) {
  point_t* point = (point_t*) malloc(sizeof(*point));
  point->x = x;
  point->y = y;
  return point;
}

void point_destroy (point_t* point) {
  free(point);
}

void point_print (point_t* point) {
  printf("(point x=%f y=%f)\n", point->x, point->y);
}
```

```
$ gcc -O0 -Wpedantic point_prog.c point.c -o point_prog
$ ./point_prog
(point x=3.140000 y=2.700000)
$
```

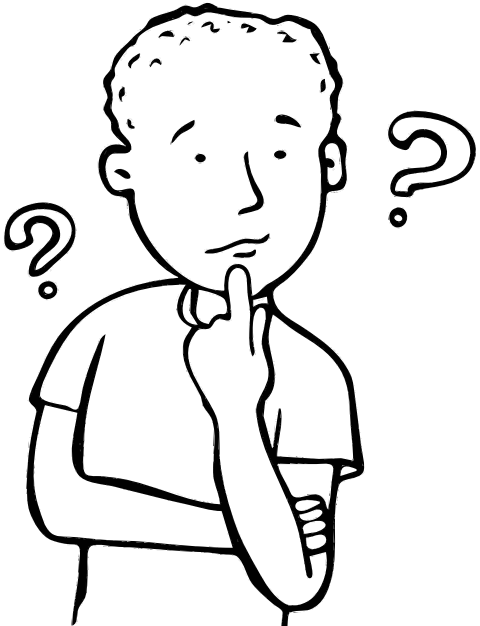
Part 5:
Parting Words

The Ten Commandments

1. Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.
2. Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.
3. Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.
4. If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.
5. Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".
6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.
7. Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.
8. Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.
9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.
10. Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.

`https://www.lysator.liu.se/c/ten-commandments.html`

Questions?



https://openclipart.org/detail/238087/boy-thinking-of-question