Lecture 2: C Memory Model

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

C vs Java

A lot of the syntax is the same, but there are two major differences:

1. C is not object-oriented; and
2. C does not have a garbage collector.

An actual computer's memory

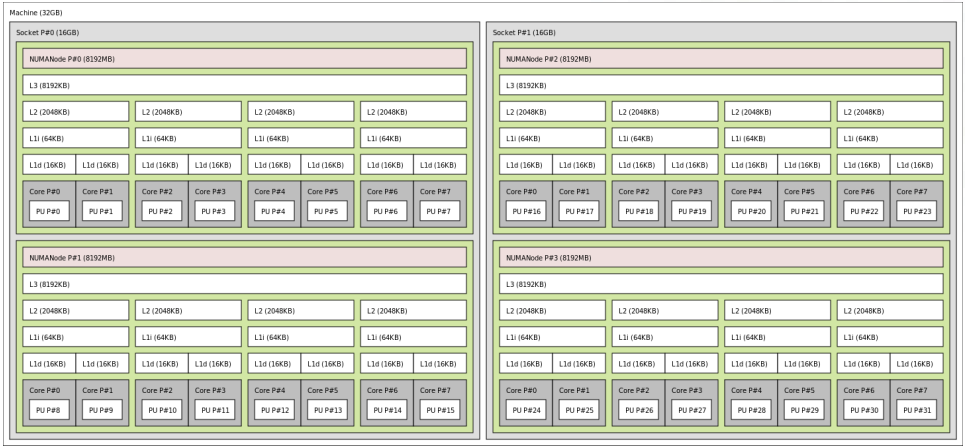
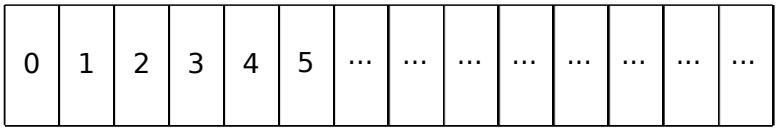


Figure: Copyright: The Portable Hardware Locality Project

A programmer's model of computer memory



What are variables?

Consider the following C code:

```c
#include <stdio.h>

int main(void) {
  int variable;

  variable = 5;

  printf("%d\n", variable);
}
```

Assembly:

```
Lcfi2:
.cfi_def_cfa_register %rbp
subq $16, %rsp
leaq L_.str(%rip), %rdi
movl $5, -4(%rbp)
movl -4(%rbp), %esi
movb $0, %al
callq _printf
xorl %esi, %esi
movl %eax, -8(%rbp)
movl %esi, %eax
addq $16, %rsp
popq %rbp
retq
.cfi_endproc
```


Variables are memory addresses

This code ...

```c
#include <stdio.h>

int main(void) {
  int variable;
  /* this is a pointer! */
  int *address;
  variable = 5;
  address = &variable;
  printf("value = %d\n", variable);
  printf("address = %p\n", address);
}
```

Produces this on my machine:

```
value = 5
address = 0x7ffee7f4442c
```


Copy-by-value

This code ...

```
#include <stdio.h>
void change_param(int aparam) {
  aparam = 7;
  printf("%d\n", aparam);
}
int main(void) {
  int variable;
  variable = 5;
  printf("value = %d\n", variable);
  change_param(variable);
  printf("value = %d\n", variable);
}
```

Produces this:

```
value = 5
7
value = 5
```


Copy-by-value and pointers

This code ...

```
#include <stdio.h>
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
int main(void) {
  int v1, v2;
  v1 = 5; v2 = 7;
  printf("v1 = %d, v2 = %d\n", v1, v2);
  swap(&v1, &v2);
  printf("v1 = %d, v2 = %d\n", v1, v2);
}
```

Produces:

```
v1 = 5, v2 = 7
v1 = 7, v2 = 5
```


Arrays

This code ...

```
#include <stdio.h>
int main(void) {
  /* declares an array of 5 ints */
  int array[5];
  int i;
  for (i=0; i<5; i++)
    array[i] = i;
  for (i=0; i<6; i++) /* oh oh */
    printf("%d ", array[i]);
  printf("\n");
}
```

Produces:

```
0 1 2 3 4 32766
```


Arrays are pointers too ...

This code produces the same thing as the previous slide ...

```
#include <stdio.h>
int main(void) {
  int array[5];
  int i;
  int *ptr;
  ptr = array;
  for (i=0; i<5; i++)
    *ptr++ = i;
  for (i=0; i<5; i++)
    printf("%d ", array[i]);
  printf("\n");
}
```

And so does this ...

```
#include <stdio.h>
int main(void) {
  int array[5];
  int i;
  int *ptr;
  ptr = array;
  for (i=0; i<5; i++)
    *ptr++ = i;
  ptr = array;
  for (i=0; i<5; i++)
    printf("%d ", *ptr++);
  printf("\n");
}
```


Array indices ...

In fact, the compiler converts:

`array[3]`

to

`*(array+3)`

`array` is a memory address, and we add 3 to the address to get the memory address of the third element of the array.

This is the reason most progamming languages start indexing arrays at 0. The index is the amount to add to the base memory address.

Dynamic arrays

If we don't know the size of an array or data structure at runtime, then we have to use dynamic memory allocation.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int *array;
  int i;
  array = (int*)malloc(5*sizeof(int)); /* allocates memory */
  for (i=0; i<5; i++)
    array[i] = i; /* access an element as usual */
  for (i=0; i<5; i++)
    printf("%d ", array[i]);
  printf("\n");
  free(array);/* deallocates the array */
}
```


Why manual memory management?

- if you want robust and secure code, then avoid manual memory management if you can
- if you want code to go as fast as it can and use as little memory as it can, then manual memory management is probably needed.
- it is better to de-allocate memory in the same scope as it was allocated. So try to organise your code so that if you allocate something in a function, you also de-allocate it in the same function.
- can't always achieve that with dynamic structures like lists and trees, in that case, try to hide allocation and deallocation behind an API