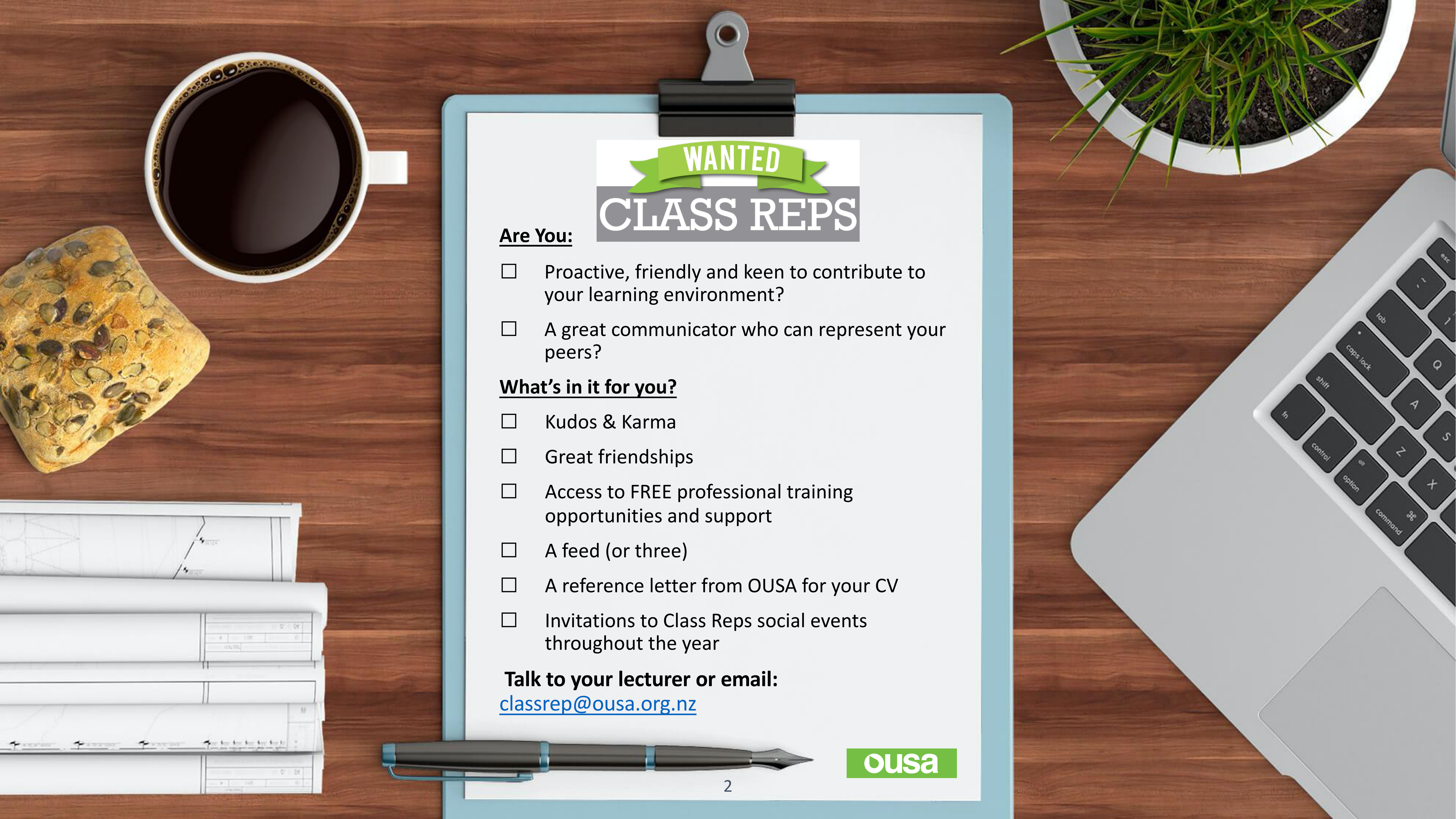
C programming
Lecture 2

COSC 242 – Algorithms and Data Structures



WANTED

CLASS REPS

Are You:

- ☐ Proactive, friendly and keen to contribute to your learning environment?
- ☐ A great communicator who can represent your peers?

What's in it for you?

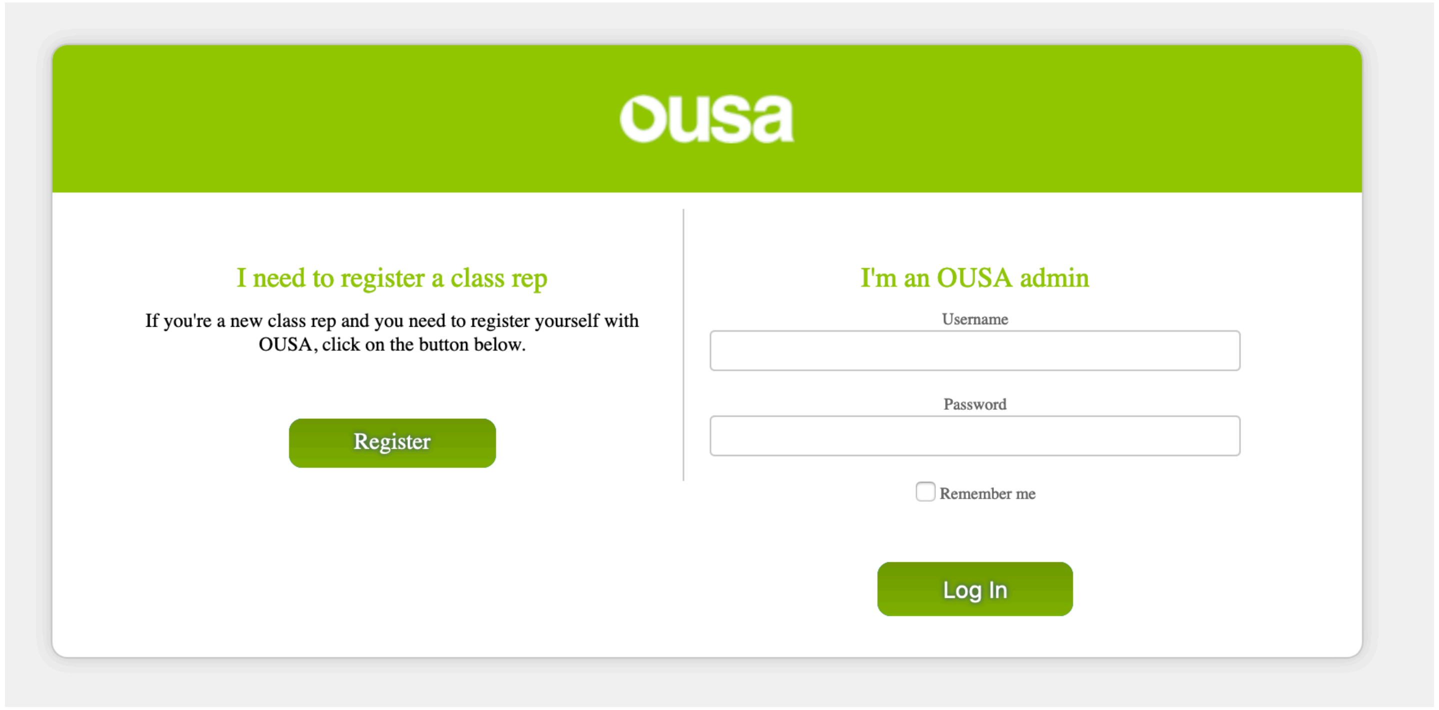
- ☐ Kudos & Karma
- ☐ Great friendships
- ☐ Access to FREE professional training opportunities and support
- ☐ A feed (or three)
- ☐ A reference letter from OUSA for your CV
- ☐ Invitations to Class Reps social events throughout the year

Talk to your lecturer or email:
classrep@ousa.org.nz

ousa

Class reps

- https://www.ousa.org.nz/support/class-reps



I need to register a class rep

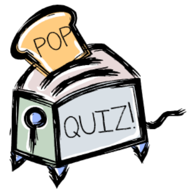
If you're a new class rep and you need to register yourself with OUSA, click on the button below.

I'm an OUSA admin

Lecture conventions

Solo activity (by yourself)

Group activity (2-3 people)



Class question

Important slide

Today's outline

1. C and memory
2. Pointers
3. Arrays
4. Memory management

Today’s outline

1. C and memory
2. Pointers
3. Arrays
4. Memory management

C and Java

C and Java share a lot of the same syntax.

This is intentional, as Java was designed to be familiar to programmers who were already experienced in C and C++

Thankfully, this 'learning boost' is symmetric: knowing Java will facilitate your learning of C.

But the two languages differ in some important, and not so important ways…

Comparing C and Java

C	Java
A procedural programming language.	An Object-Orientated language.
Developed by Dennis M. Ritchie in 1972. It was a successor to B, developed earlier by Ken Thompson and Ritchie in 1969.	Developed by James Gosling in 1995
Memory allocation can be done by *malloc*	Memory allocation can be done by the *new* keyword.
Memory allocated to variable is freed using *free*	A compiler will free up the memory by automatically calling the garbage collector.
Is a middle-to-lower level language. It is closer to the machine level and its architecture.	A high-level language. The JVM translates Java bytecode into machine language through either a just-in-time compiler (JIT), or at runtime with the interpreter.
Uses pointers	Uses references

Many more differences [1, 2]

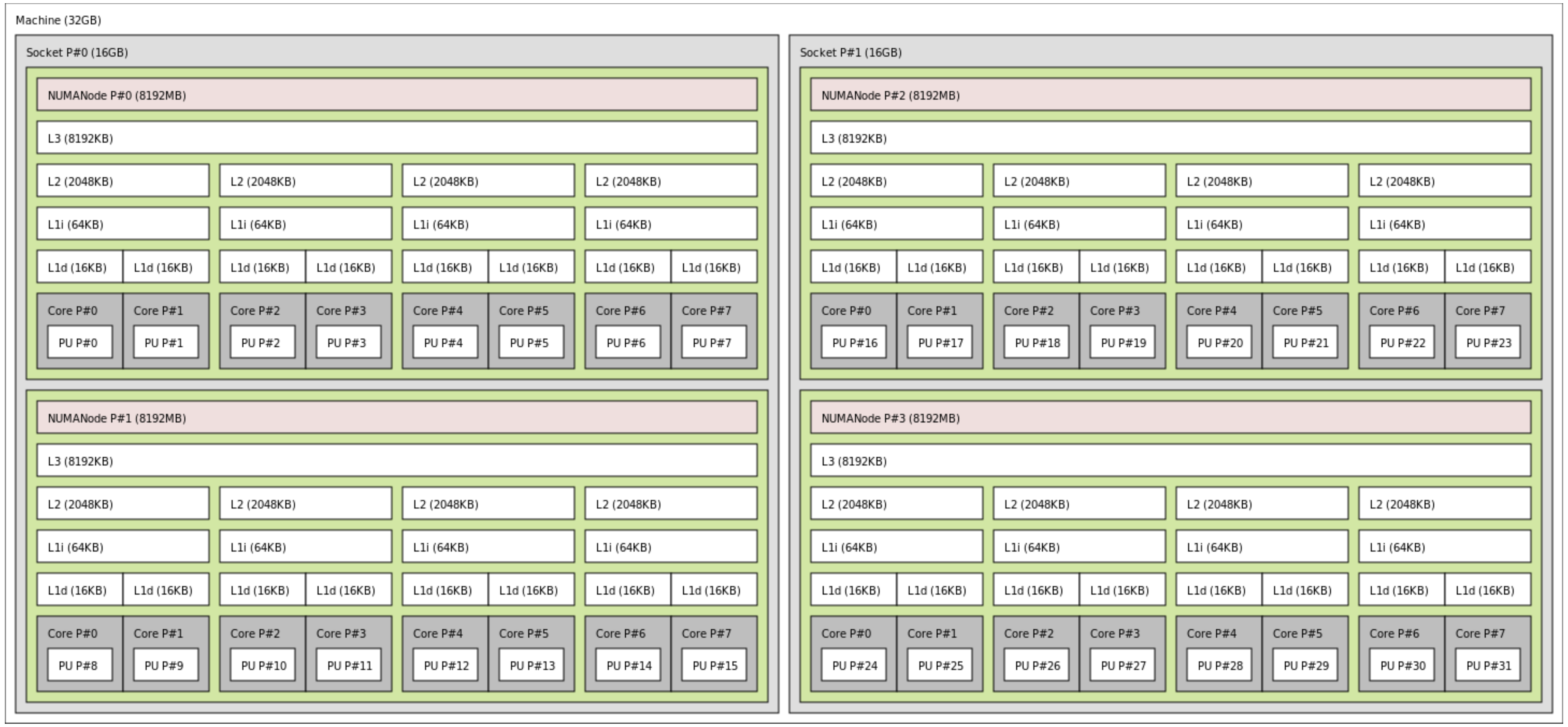
Memory management

An important difference between C and Java is how memory is managed. This includes allocation and deallocation.

In Java, memory is largely managed automatically. This means fewer bugs, but entails a performance penalty.

In C, memory is managed manually (i.e. by the programmer). This leads to more bugs, but increased performance.

Topology and memory of an actual computer



https://www.open-mpi.org/projects/hwloc/

A programmer’s view of memory

As programmers, we don’t worry about the low-level details of specific machine architectures.

Instead, we view memory as an array. In most modern computers, main memory (RAM) is divided into **bytes**. Each byte holds 8 bits:

1	1	1	1	0	0	1	0

Question: what number do we have?

Bytes and addresses

Each byte of memory has a unique **address** to distinguish it from other other bytes in memory.

With *n* bytes of memory, addresses range from 0.. *n*-1.

Address	**Contents**
0	10010011
1	01010101
2	00001111
3	11000011
…	…
n-1	11001100

What are variables?

Consider the following **C code:**

```
#include <stdio.h>

int main()
{
    int my_var;

    my_var = 9098;
    printf("%d\n", my_var);
    return 0;
}
```

What are variables?

Consider the following **C code:**

```c
#include <stdio.h>

int main()
{
    int my_var;

    my_var = 9098;
    printf("%d\n", my_var);
    return 0;
}
```

Assembly

gcc -S -c my_var.c

```
.cfi_def_cfa_register %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
movl    $9098, -8(%rbp)     ## imm = 0x238A
movl    -8(%rbp), %esi
leaq    L_.str(%rip), %rdi
movb    $0, %al
callq   _printf
xorl    %esi, %esi
movl    %eax, -12(%rbp)     ## 4-byte Spill
movl    %esi, %eax
addq    $16, %rsp
popq    %rbp
retq
.cfi_endproc
```

Variables have a memory address

This code…

```
#include <stdio.h>

int main()
{
    int my_var;
    // This is a pointer
    int *p_var;

    my_var = 5;
    // Store mem address of my_var in p_var
    p_var = &my_var;

    printf("Value: %d\n", my_var);
    printf("Address: %p\n", p_var);
}
```

Variables have a memory address

This code…

```c
#include <stdio.h>

int main()
{
    int my_var;
    // This is a pointer
    int *p_var;

    my_var = 5;
    // Store mem address of my_var in p_var
    p_var = &my_var;

    printf("Value: %d\n", my_var);
    printf("Address: %p\n", p_var);
}
```

Produces this on my machine…

```
Value: 5
Address: 0x7ffee6fd659c
```

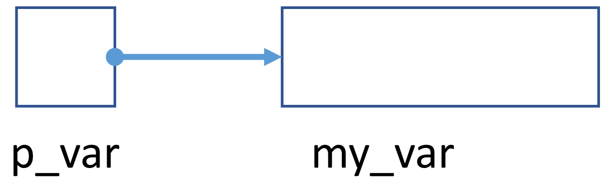
Pointers

Memory addresses are represented by numbers[1].

Their range of values may differ from integers, so we can't store those addresses in an integer type variable.

Instead, we store memory addresses in **pointer variables**.

When we store the address of `my_var` in the pointer `p_var`, we say that `p_var` "points to" `my_var`.



Declaring a pointer

We declare a pointer variable in much the same way as any other variable.

The only difference is that the name of the variable must be preceded by an **asterisk '*'**:

```
int *p_var; /* can point to an integer variable */
```

Pointer type

Every pointer variable can only point to objects of a particular type. This is called the **referenced type**.

```
int *i;     /* Points only to integers */
char *j;    /* Points only to characters */
double *k; /* Points only to doubles */
int *l[n]; /* Array of pointers, each points to ints */
```

Address and indirection operators

& (address) operator - Find the address of an initialized variable.

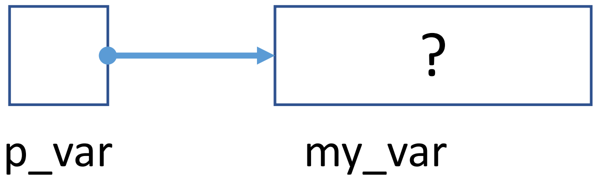
If `my_var` is a variable, then `&my_var` is the address of `my_var` in memory.

***** (indirection) operator - Access the object that a pointer points to.

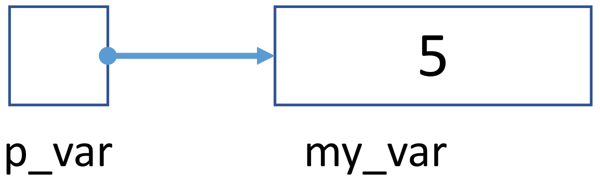
If `*p_var` points to `my_var`, then `*p_var` represents the object `my_var`

Address and indirection operators

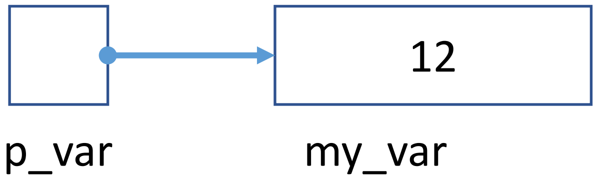
```
int my_var, *p_var;    /* Declare variables */
p_var = &my_var;       /* p_var now points to my_var */
```



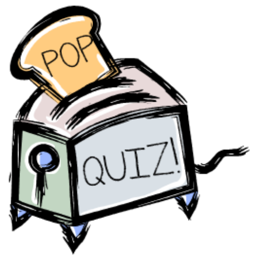
```
*p_var = 5;            /* my_var now holds 5 */
```



```
my_var = 12;           /* my_var now holds 12 */
```



Pop quiz 1



- What will be output by the following:

```
char my_var, *p_var;
p_var = &my_var;
my_var = 'x’;
*p_var = 'a’;
printf("%c\n", my_var);
printf("%c\n", *p_var);
```

Answers

- x then a
- a then a
- x then x

Reminder: Pointer to a memory address

```c
#include <stdio.h>

int main()
{
    int my_var;
    // This is a pointer
    int *p_var;

    my_var = 5;
    // Store mem address of my_var in p_var
    p_var = &my_var;

    printf("Value: %d\n", my_var);
    printf("Address: %p\n", p_var);
}
```

Output

```
Value: 5
Address: 0x7ffee6fd659c
```

Address and indirection operators

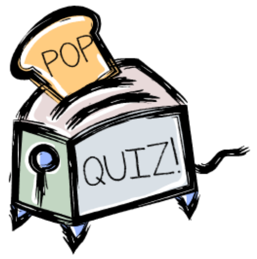
You can think of ***** as the inverse of **&**.

Applying **&** to a variable produces a pointer to the variable.

Applying ***** to the pointer takes us back to the original variable:

```
int x;
int y = *&x;      /* same as y = x */
```

Pop quiz 2



What might the following code do?

```
int *my_var;
*my_var = 55;
printf("%d\n", *my_var);
```

A common bug

This is a common error in C programming.

Declaring a pointer sets aside space for a pointer, but it doesn’t make it point to an object.

```
int *i;  /* points nowhere in particular */
*i = 7; /* error. Where should I put this? */
```

Arguments are pass-by-value

In C, function arguments are **passed by value**.

When a function is called, each argument is evaluated. The argument’s value is then assigned (copied) to the corresponding parameter in the callee function.

As the parameter contains a copy of the argument’s value, any changes to the parameter in the callee function doesn’t affect the argument in the caller function.

This is the same behavior in Java.

Example: Pass by value

This code...

```
#include <stdio.h>

void change_param(int aparam) {
  aparam = 7;
  printf("2. %d\n", aparam);
}
int main(void) {
  int variable;
  variable = 5;
  printf("1. value = %d\n", variable);
  change_param(variable);
  printf("3. value = %d\n", variable);
}
```

Produces:

```
1. Value = 5
2. 7
3. Value = 5
```

Pass by value with pointers

Pass by value also operates when pointers are used as function arguments.

With a pointer, the “value” is the memory address (what’s being pointed to).

That is, the ‘value’ that is passed (copied) from the caller’s argument into the callee’s parameter is the memory address.

Example: Swapping numbers

Say we have two variables, a and b. We want to swap their contents.

This swap operation needs to happen frequently, as it is used in a sorting algorithm.

We decide to implement this naively using primitives.

Example: Swapping numbers (naïve)

```
void swap(int n, int m)
{
    int temp = n;
    n = m;
    m = temp;
}

int main()
{
    int n1 = 55;
    int n2 = 77;
    printf("1. n1 = %d | n2 = %d\n", n1, n2);
    swap(n1, n2);
    printf("2. n1 = %d | n2 = %d\n", n1, n2);
    return 0;
}
```

Output

```
1. n1 = 55 | n2 = 77
2. n1 = 55 | n2 = 77
```

Not what we wanted…

Example: Swapping with pointers

```c
void swap(int *n, int *m)
{
    int temp = *n;
    *n = *m;
    *m = temp;
}

int main()
{
    int n1 = 55;
    int n2 = 77;
    printf("1. n1 = %d | n2 = %d\n", n1, n2);
    swap(&n1, &n2);
    printf("2. n1 = %d | n2 = %d\n", n1, n2);
    return 0;
}
```

Output

```
1. n1 = 55 | n2 = 77
2. n1 = 77 | n2 = 55
```

Success!

C pointers vs Java References

Question: Aren’t C pointers the same thing as references in Java?

Answer: No, they are not the same, but they are conceptually related.

C pointers vs Java References

Java reference is a variable that refers to something else, and can be used as an alias for that thing.

When we pass a reference (object) as an argument, the refence to that object is copied into the parameter, not the object itself.

A **pointer** is a variable that stores a memory address. It acts as an alias to what is stored at that memory address.

When we pass a pointer as an argument, the address is copied into the parameter.

C pointers vs Java References

"They sound the same to me. Are they really different?"

Yes. A pointer is a reference, but a reference is not necessarily a pointer. Pointers can do more things than references. Such as:

Pointer arithmetic – As pointers are variables, we can do clever arithmetic to move around memory. This also allows us to navigate around arrays with concise syntax.

Speaking of arrays...

Pointers can do more than this, such as typecasting, but we won't go into that…

Arrays in Java

This code…

```
public class ArrayBounds {
    public static void main(String[] args) {
        int array[] = new int[5];
        int i;
        for (i = 0; i < 5; i++)
            array[i] = i;
        for (i = 0; i < 6; i++)
            System.out.println(array[i] + ", ");
        System.out.println();
    }
}
```

Produces:

```
Exception in thread "main" 0,
java.lang.ArrayIndexOutOfBoundsException:
Index 5 out of bounds for length 5
at mic.ArrayBounds.main(ArrayBounds.java:13)
1,
2,
3,
4,
```

Arrays in C

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++)
        printf("%d ", array[i]);
    printf("\n");
  return 0;
}
```

Produces:

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

?

There are no compile checks to make sure you don't access anything past the end of the array.

Generally there are no runtime checks either. It might cause a runtime error (seg fault or similar), but it may not.

Accessing outside of array bounds is defined as "undefined behavior".

Segmentation fault

This code...

```c
#include <stdio.h>

int main(void) {
    int array[] = {0, 1, 2, 3, 4};
    printf("array[0] is %d\n", array[0]);
    /* Read access beyond bounds */
    printf("array[10] is %d\n", array[10]);

    /* Write access beyond bounds */
    array[10] = 11;
    printf("array[10] is %d\n", array[10]);
    return 0;
}
```

Produces:

```
array[0] is 0
array[10] is 1893035209
array[10] is 11
[1]    17659 segmentation fault
```

A segmentation fault is caused by the program attempting to access memory it is not allowed to access.

Arrays are pointers too in C

This code produces the same output..

```c
#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");
    return 0;
}
```

Produces:

```
0 1 2 3 4
```

Arrays are pointers too in C

Likewise with this code…

```c
#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");
    return 0;
}
```

Produces:

```
0 1 2 3 4
```

Syntactic sugar

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 programming 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 i, *array;
    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 */
    return 0;
}
```

Produces:

```
0 1 2 3 4
```

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.

Why manual memory management?

It is better to de-allocate memory in the same scope as it was allocated.

Try to organise your code so that if you allocate something in a function, you also de-allocate it in the same function.

You can't always achieve that with dynamic structures like lists and trees.

In that case, try to hide allocation and deallocation behind an API.

Real-time C to Assembly

Would you like to see C compiled to Assembly in real-time?

Visit https://gcc.godbolt.org

Helpful resources on proofs

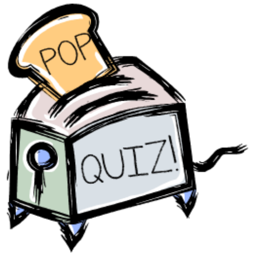
C books

- The C Programming Language, by Kernighan and Ritchie, Prentice Hall, 2nd Edition, 1988.
- C Programming: A modern approach, by K. N. King, W. W. Norton & Company, 2nd Edition, 2008.

If you find K&R's style too dense or brief, then King's book is another excellent option.

Solutions

Pop quiz 1



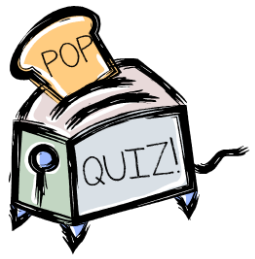
- What will be output by the following:

```
char my_var, *p_var;
p_var = &my_var;
my_var = 'x’;
*p_var = 'a’;
printf("%c\n", my_var);
printf("%c\n", *p_var);
```

Answers

- x then a
- a then a ✓
- x then x

Pop quiz 2



- What might the following code do?

```
int *my_var;
*my_var = 55;
printf("%d\n", *my_var);
```

Output – crashes!

```
[1]    83447 segmentation fault  ./simple
```

References and attributions

1. C Programming: A modern approach, by K. N. King, W. W. Norton & Company, 2nd Edition, 2008.

Image attributions