CASS: Exercise session 1

An introduction to C

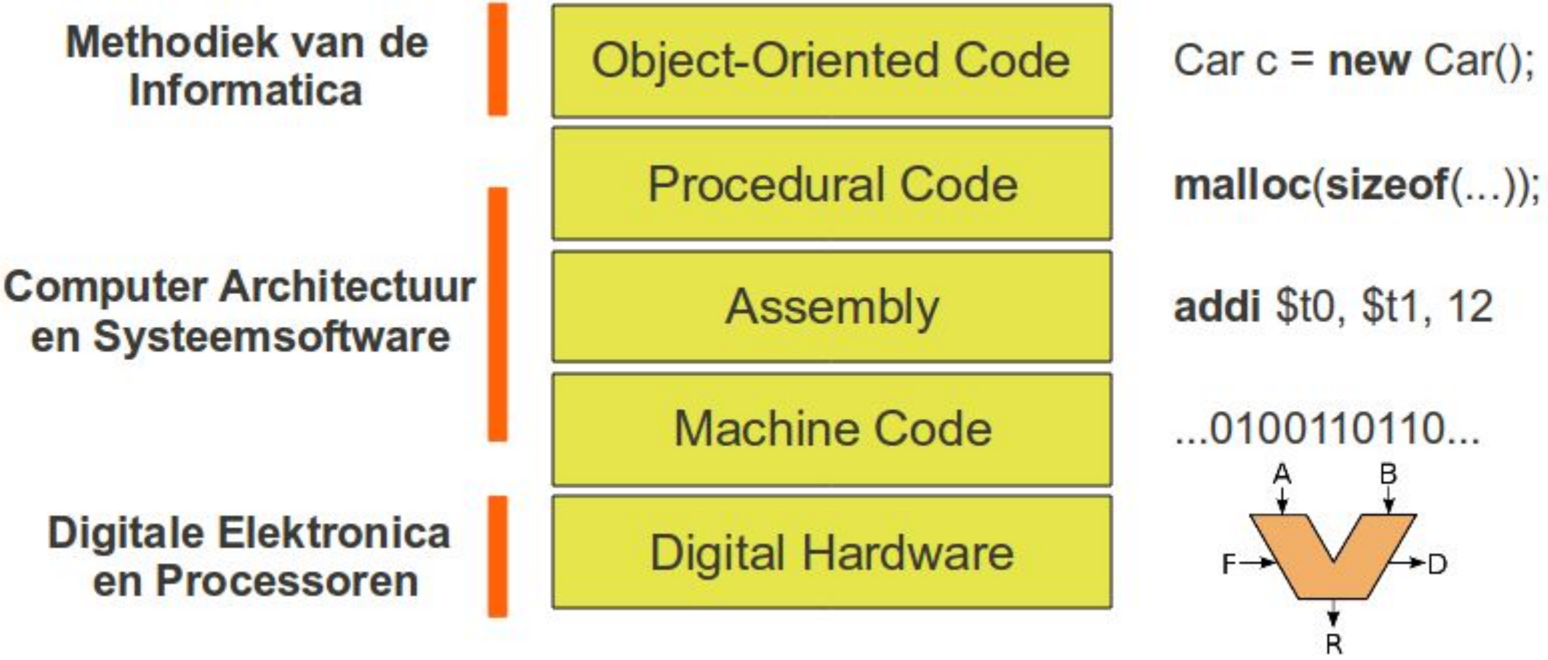
Exercise sessions: practical

- 9 sessions of 2.5 hours
- Bring laptop
- Solutions to exercises on Toledo (end of week)

Content

- Low-level programming: C and Risc-V
- Planning
    - Session 1: Introduction to C
    - Session 2: Introduction to Risc-V
    - Session 3: Stack & Recursion
    - Session 4: Pointers and heap
    - Session 5: Linked list (Risc-V and C review)
    - Session 6: Cache
    - Session 7: Performance
    - Session 8: Syscalls, I/O and OS
    - Session 9: Review

Perspective



Methodiek van de Informatica

Computer Architectuur en Systeemsoftware

Digitale Elektronica en Processoren

Object-Oriented Code

Procedural Code

Assembly

Machine Code

Digital Hardware

Car c = **new** Car();

malloc(**sizeof**(...));

addi $t0, $t1, 12

...0100110110...

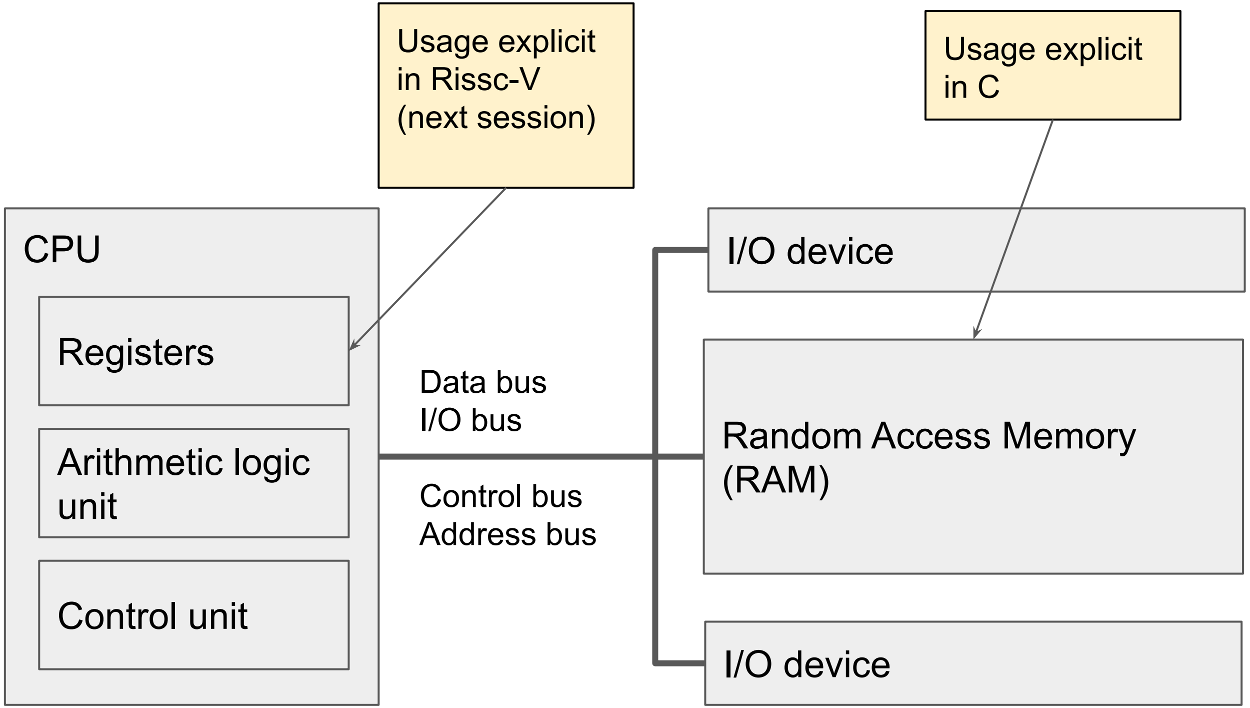
Goals of today

- Knowing what C is
- Understanding basic C data structures
- Understanding static memory allocation
- Understanding basic IO
- Write, compile and run simple C programs

C versus Java

- “Medium-level” programming language
  - Manual memory management
    - Pointers
  - Procedural (vs object-oriented)
    - No classes or objects
  - Compiled language (vs interpreted)
  - No error handling
    - Error leads to crash or undefined behaviour
  - …
- More work for a C programmer
  - More room for mistakes
  - But also more freedom

Basic processor architecture



Hello world

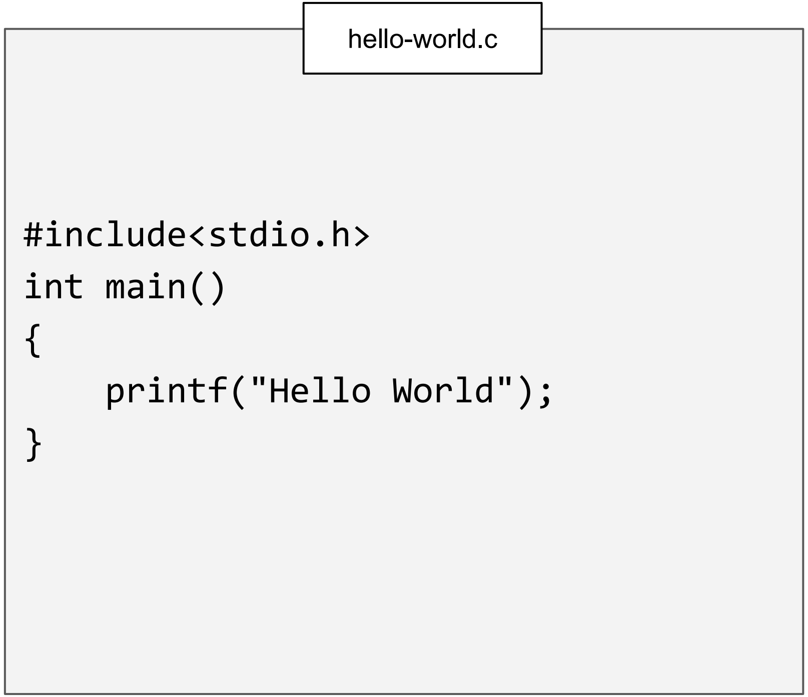
hello-world.c

```c
#include<stdio.h>
int main()
{
    printf("Hello World");
}
```

Console

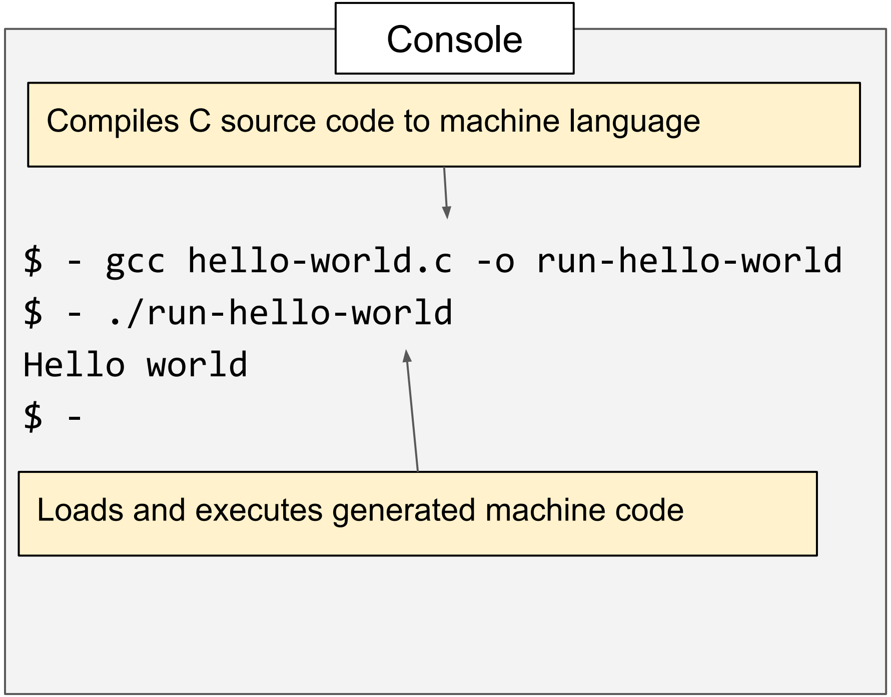
```
$ - gcc hello-world.c -o run-hello-world
$ - ./run-hello-world
Hello world
$ -
```

Hello world



hello-world.c

```c
#include<stdio.h>
int main()
{
    printf("Hello World");
}
```



Console

Compiles C source code to machine language

```
$ - gcc hello-world.c -o run-hello-world
$ - ./run-hello-world
Hello world
$ -
```

Loads and executes generated machine code

Integers

integers.c

```
#include <stdio.h>
int main()
{
   int i = 5;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
}
```

Console

```
$ - gcc integers.c -o run-integers
$ - ./run-integers
  Value of i: 5
Address of i: 0x7fd1dfe4
$ -
```

Location of the variable i in memory

Integers

integers.c

```
#include <stdio.h>
int main()
{
   int i = 5;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
}
```

Memory (32-bit)

Address	**Value**
...	...
0x7ff1dfe0	?
0x7ff1dfe4	0x00..05
0x7ff1dfe8	?
...	...

Byte addressing

4 bytes per word* (32 bits)

*machine specific

Pointers

pointers.c

```c
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
   printf("  Value of j: %p\n", j);
   printf("Address of j: %p\n", &j);
   printf("  Value of k: %i\n", k);
   printf("Address of k: %p\n", &k);
}
```

Console

```
$ - gcc pointers.c -o run-pointers
$ - ./run-pointers
  Value of i: 5
Address of i: 0x7f698878
  Value of j: 0x7f698878
Address of j: 0x7f698880
  Value of k: 5
Address of k: 0x7f69887c
$ -
```

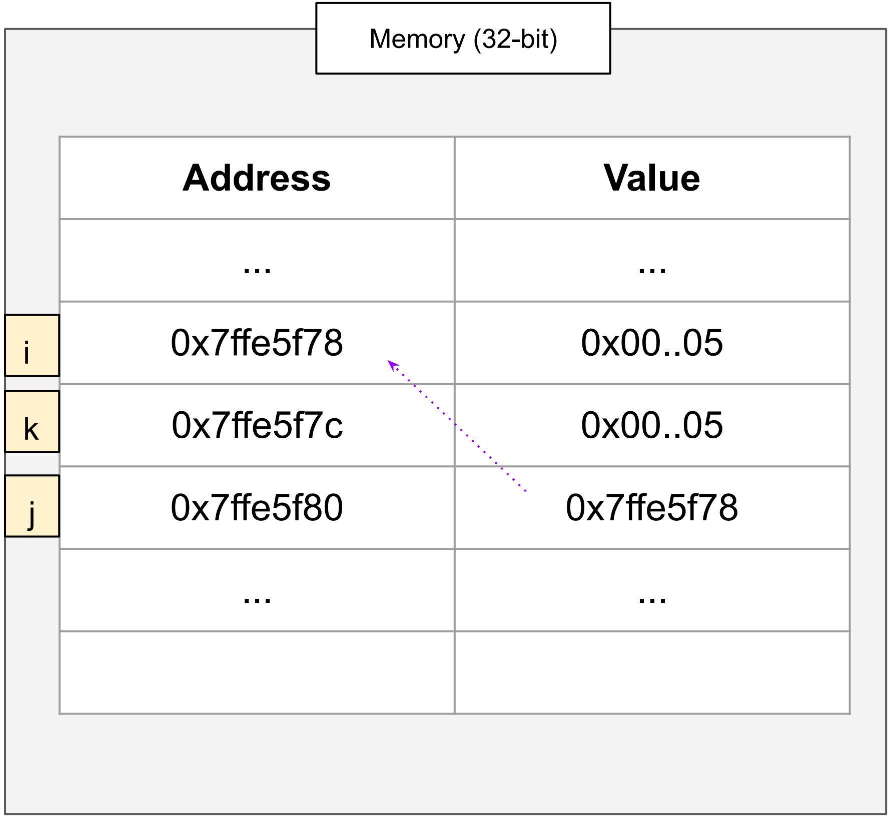
Pointers

pointers.c

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
   printf("  Value of j: %p\n", j);
   printf("Address of j: %p\n", &j);
   printf("  Value of k: %i\n", k);
   printf("Address of k: %p\n", &k);
}
```

Pointer declaration

*var – value stored under the address in var
&var – address of var



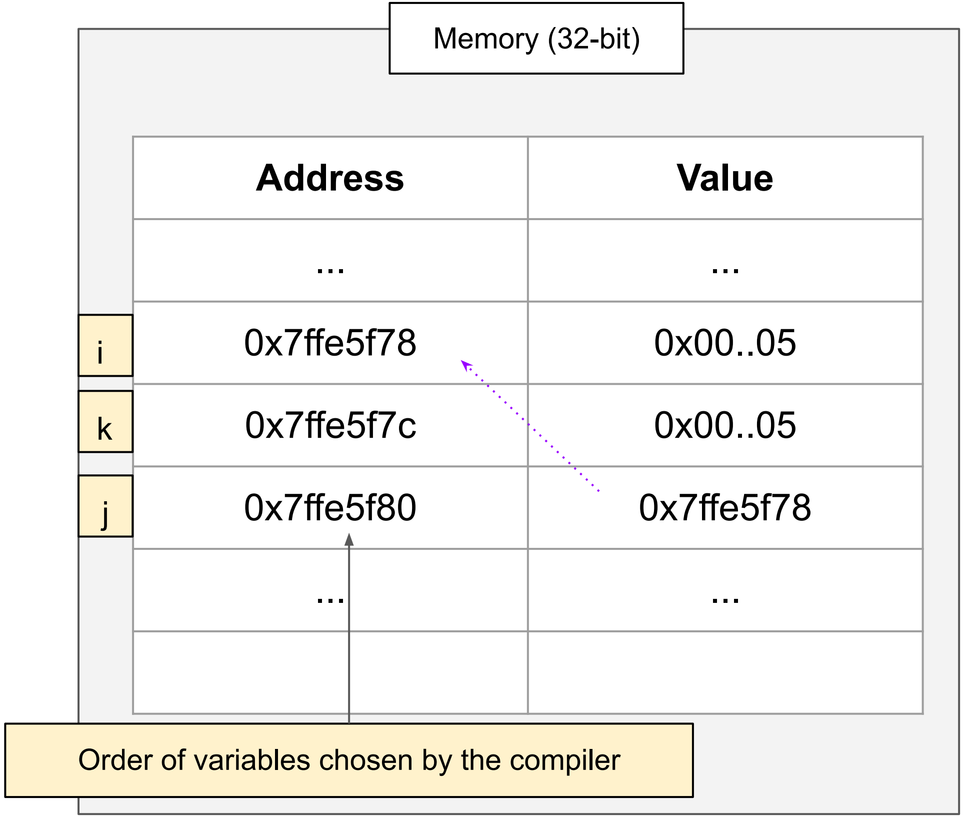
Memory (32-bit)

	Address	Value
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

Pointers

pointers.c

```c
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
   printf("  Value of j: %p\n", j);
   printf("Address of j: %p\n", &j);
   printf("  Value of k: %i\n", k);
   printf("Address of k: %p\n", &k);
}
```



Memory (32-bit)

	Address	**Value**
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

Order of variables chosen by the compiler

Question 1: what changes in memory?

tmp

```
#include <stdio.h>
int main()
{
    int i = 5;
    int* j = &i;
    int k = *j;
    //initial memory
    k = 3;
    //new memory
}
```

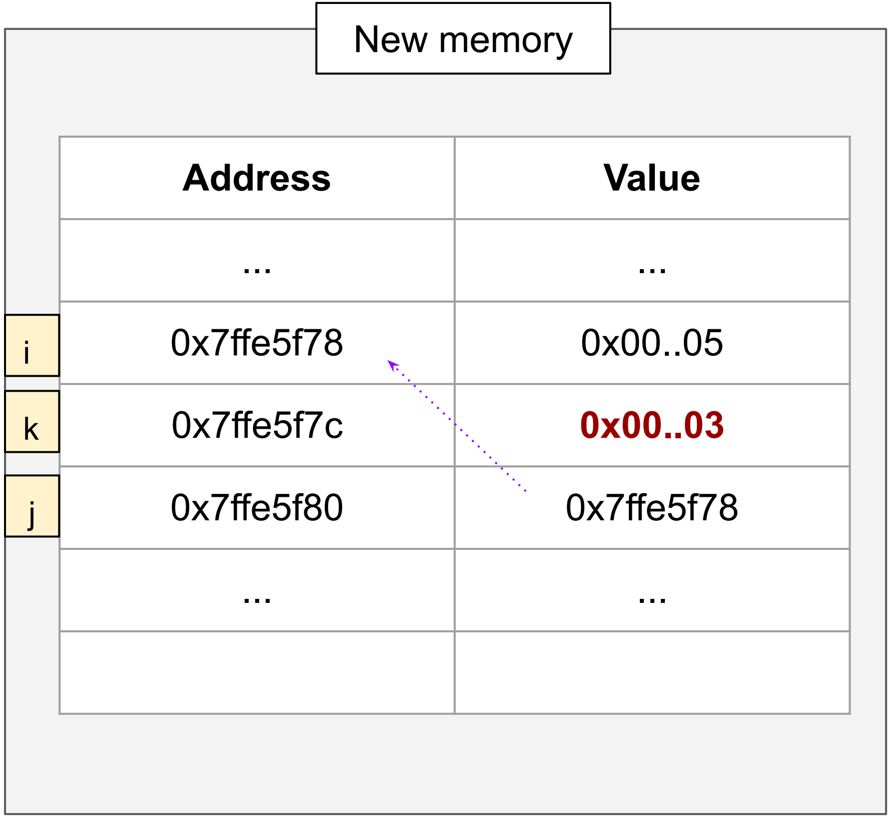
Initial memory

	Address	**Value**
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

Answer 1

tmp

```
#include <stdio.h>
int main()
{
    int i = 5;
    int* j = &i;
    int k = *j;
    //initial memory
    k = 3;
    //new memory
}
```



New memory

	Address	**Value**
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	**0x00..03**
j	0x7ffe5f80	0x7ffe5f78
	...	...

Question 2: what changes in memory?

tmp

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   //initial memory
   j = 9;
   //new memory
}
```

Initial memory

	Address	Value
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

Answer 2

tmp

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   //initial memory
   j = 9; //Undefined behavior!
   //new memory
}
```

Don't do this!!! Variable `j` is a pointer and stores addresses, not integers

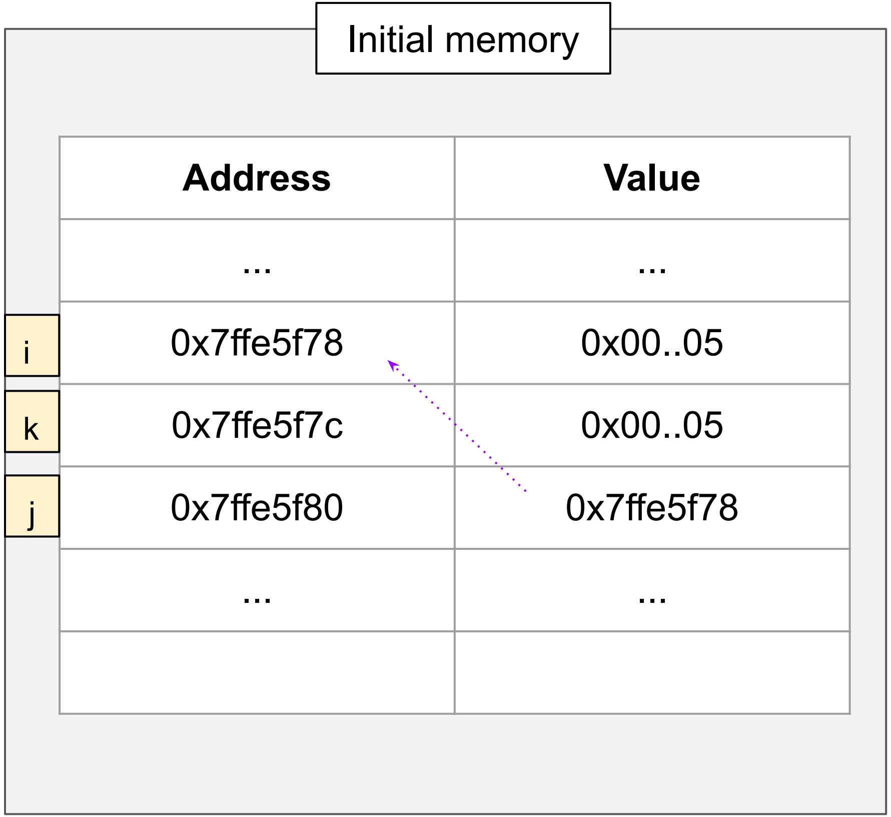
New memory

	Address	Value
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	**0x00..09**
	...	...

Question 3: what changes in memory?

tmp

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   //initial memory
   j = &k;
   //new memory
}
```



Initial memory

	Address	**Value**
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

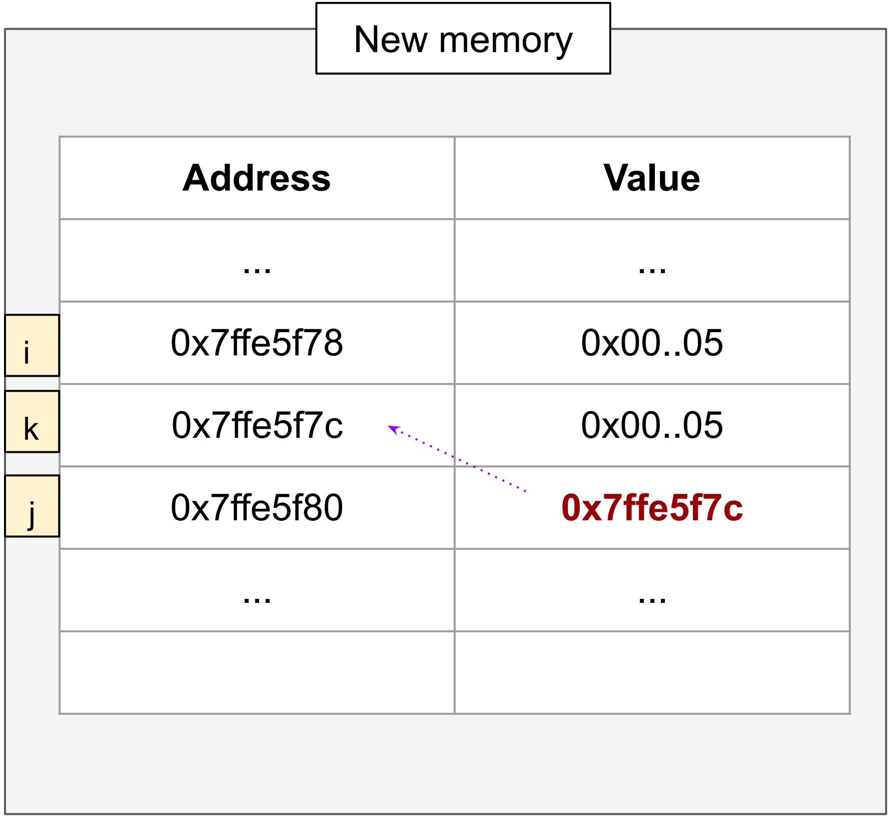
Answer 3

tmp

```
#include <stdio.h>
int main()
{
    int i = 5;
    int* j = &i;
    int k = *j;
    //initial memory
    j = &k;
    //new memory
}
```

Correct as &k is a valid pointer value*

- j = k would be the same mistake as the last example



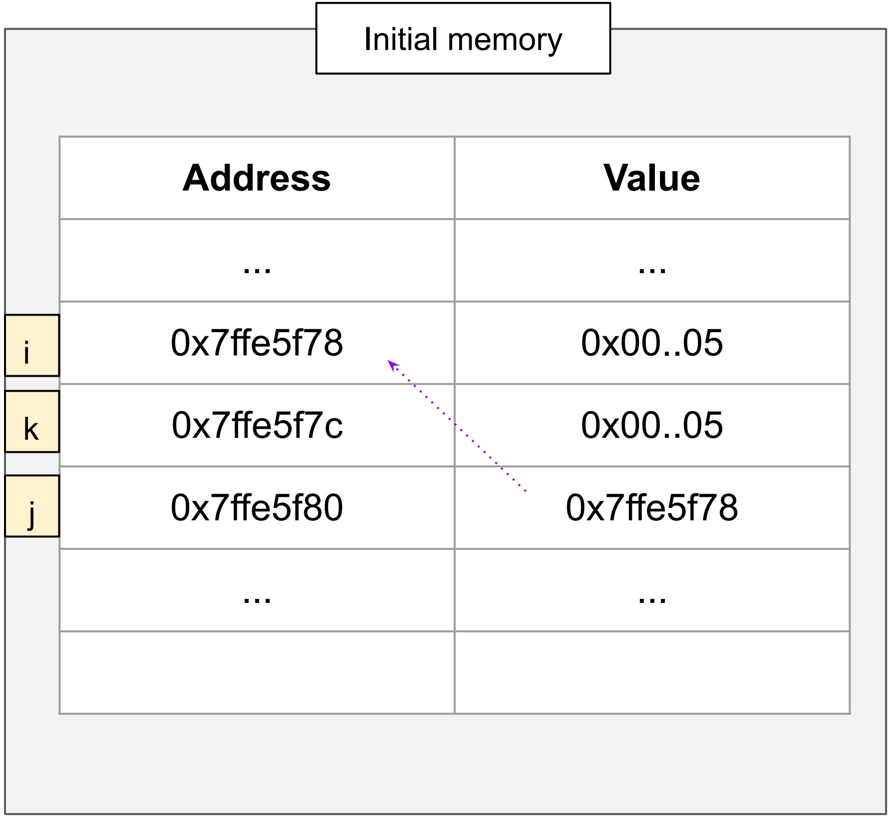
New memory

	Address	Value
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	**0x7ffe5f7c**
	...	...

Question 4: what changes in memory?

tmp

```
#include <stdio.h>
int main()
{
    int i = 5;
    int* j = &i;
    int k = *j;
    //initial memory
    *j = 9;
    //new memory
}
```



Initial memory

	Address	Value
	...	...
i	0x7ffe5f78	0x00..05
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

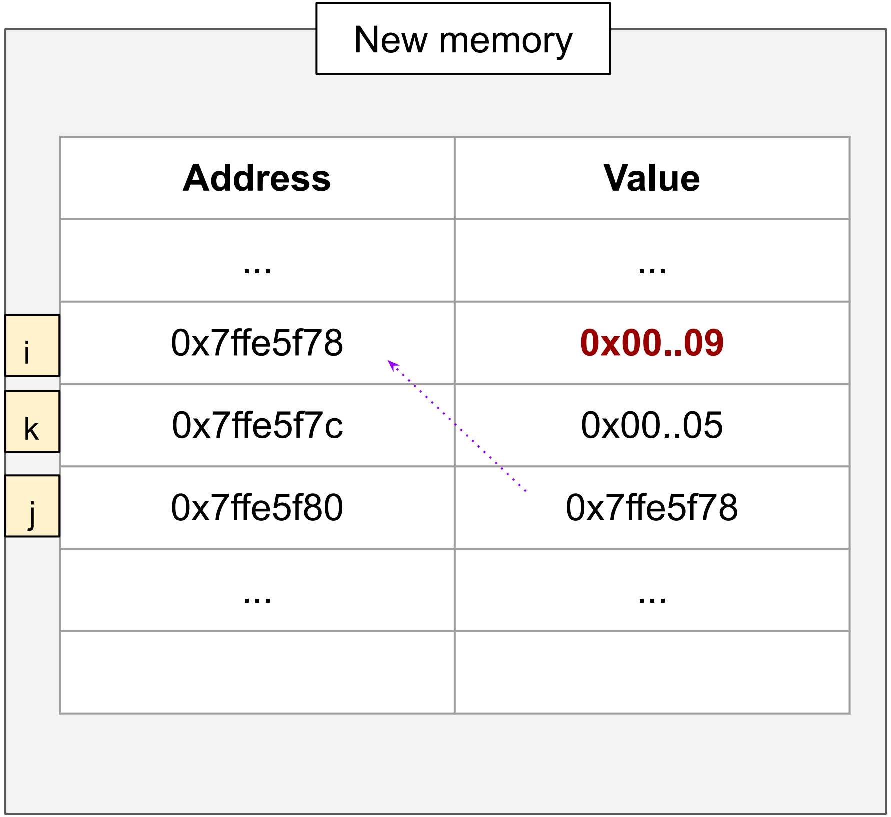
Answer 4

tmp

```
#include <stdio.h>
int main()
{
    int i = 5;
    int* j = &i;
    int k = *j;
    //initial memory
    *j = 9;
    //new memory
}
```

Writing *j is the same as writing i because j points to i

New memory



	Address	Value
	...	...
i	0x7ffe5f78	**0x00..09**
k	0x7ffe5f7c	0x00..05
j	0x7ffe5f80	0x7ffe5f78
	...	...

Other data types

- Basic types
    - char – stores a single character of a string
        - Defined as smallest addressable unit of a machine, typically 1 byte (most machines are byte addressable)
    - int – integer number
        - Guaranteed to be **at least** 16 bits, typically 32 bits
    - long – large integer number
        - Guaranteed to be **at least** 32 bits, typically 64 bits
    - float – floating point number
        - Usually 23 bits of significand, 8 bits of exponent, and 1 sign bit
    - double – double precision floating point number
        - Usually 52 bits of significand, 11 bits of exponent, and 1 sign bit

Other data types

- Data type modifier
    - unsigned
        - Stores only positive numbers
          (E.g. unsigned int: only positive integers)
- Exhaustive list and their format strings
    - https://en.wikipedia.org/wiki/C_data_types
    - <list some of the format specifiers for convenience?>

Size of data type - sizeof

sizeof.c

```
#include <stdio.h>
int main()
{
   printf("  Bytes in char: %lu\n",
          sizeof(char));
   printf(" Bytes in char*: %lu\n",
          sizeof(char*));
   printf("   Bytes in int: %lu\n",
          sizeof(int));
   printf("  Bytes in uint: %lu\n",
          sizeof(unsigned int));
   //rest of main analogous
}
```

Console

```
$ - gcc sizeof.c -o run-sizeof
$ - ./run-sizeof
  Bytes in char: 1
 Bytes in char*: 4
   Bytes in int: 4
  Bytes in uint: 4
  Bytes in long: 8
 Bytes in ulong: 8
Bytes in double: 8
 Bytes in float: 4
$ -
```

Arrays

arrays.c

```
#include <stdio.h>
int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  int i;
  printf("   Address of arr: %p\n", arr);
  for(i = 0 ; i < 5 ; i++){
     printf("  Value of arr[%i]: %i\n",
            i, arr[i]);
     printf("Address of arr[%i]: %p\n",
            i, &arr[i]);
  }
  printf("    Size of arr: %lu\n",
         sizeof(arr));
}
```

Console

```
$ - gcc arrays.c -o run-arrays
$ - ./run-arrays
   Address of arr: 0x7f60a6d0
  Value of arr[0]: 1
Address of arr[0]: 0x7f60a6d0
  Value of arr[1]: 2
Address of arr[1]: 0x7f60a6d4
  Value of arr[2]: 3
Address of arr[2]: 0x7f60a6d8
  Value of arr[3]: 4
Address of arr[3]: 0x7f60a6dc
  Value of arr[4]: 5
Address of arr[4]: 0x7f60a6e0
      Size of arr: 20
$ -
```

Arrays

arrays.c

```
#include <stdio.h>
int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  int i;
  printf("   Address of arr: %p\n", arr);
  for(i = 0 ; i < 5 ; i++){
     printf("  Value of arr[%i]: %i\n",
            i, arr[i]);
     printf("Address of arr[%i]: %p\n",
            i, &arr[i]);
  }
  printf("    Size of arr: %lu\n",
         sizeof(arr));
}
```

Memory

Address	Value
...	...
0x7f60a6d0	**0x00..01**
0x7f60a6d4	0x00..02
0x7f60a6d8	0x00..03
0x7f60a6dc	0x00..04
0x7f60a6e0	0x00..05
...	...

Question: can we calculate size of array?

arrays.c

```
#include <stdio.h>
int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  int i;
  printf("   Address of arr: %p\n", arr);
  for(i = 0 ; i < 5 ; i++){
    printf("  Value of arr[%i]: %i\n",
            i, arr[i]);
    printf("Address of arr[%i]: %p\n",
            i, &arr[i]);
  }
  printf("    Size of arr: %lu\n",
         sizeof(arr));
}
```

Console

```
$ - gcc arrays.c -o run-arrays
$ - ./run-arrays
   Address of arr: 0x7f60a6d0
  Value of arr[0]: 1
Address of arr[0]: 0x7f60a6d0
  Value of arr[1]: 2
Address of arr[1]: 0x7f60a6d4
  Value of arr[2]: 3
Address of arr[2]: 0x7f60a6d8
  Value of arr[3]: 4
Address of arr[3]: 0x7f60a6dc
  Value of arr[4]: 5
Address of arr[4]: 0x7f60a6e0
      Size of arr: 20
$ -
```

Answer

arrays.c

```c
#include <stdio.h>
int main()
{
  int arr[] = {1, 2, 3, 4, 5};
  int i;
  printf("   Address of arr: %p\n", arr);
  for(i = 0 ; i < sizeof(arr)/sizeof(int)
      ; i++){
    //loop body

  }
  printf("    Size of arr: %lu\n",
         sizeof(arr));
}
```

Console

```
$ - gcc arrays.c -o run-arrays
$ - ./run-arrays
   Address of arr: 0x7f60a6d0
  Value of arr[0]: 1
Address of arr[0]: 0x7f60a6d0
  Value of arr[1]: 2
Address of arr[1]: 0x7f60a6d4
  Value of arr[2]: 3
Address of arr[2]: 0x7f60a6d8
  Value of arr[3]: 4
Address of arr[3]: 0x7f60a6dc
  Value of arr[4]: 5
Address of arr[4]: 0x7f60a6e0
      Size of arr: 20
$ -
```

Strings

strings.c

```
#include <stdio.h>
int main(){
  char abc_1[] = {'a', 'b', 'c'};
  char abc_2[] = {'a', 'b', 'c', '\0'};
  char abc_3[] = "abc";
  puts(abc_1);
  puts(abc_2);
  puts(abc_3);
  printf("Size 1: %lu\n", sizeof(abc_1));
  printf("Size 2: %lu\n", sizeof(abc_2));
  printf("Size 3: %lu\n", sizeof(abc_3));
  printf("Addr 1: %p\n", abc_1);
  printf("Addr 2: %p\n", abc_2);
  printf("Addr 3: %p\n", abc_3);
}
```

Console

```
$ - gcc arrays.c -o run-arrays
$ - ./run-arrays
abcabc
abc
abc
Size 1: 3
Size 2: 4
Size 3: 4
Addr 1: 0x7fec70ad
Addr 2: 0x7fec70b0
Addr 3: 0x7fec70b4
$ -
```

Strings

strings.c

```c
#include <stdio.h>
int main(){
  char abc_1[] = {'a', 'b', 'c'};
  char abc_2[] = {'a', 'b', 'c', '\0'};
  char abc_3[] = "abc";
  puts(abc_1);
  puts(abc_2);
  puts(abc_3);
  printf("Size 1: %lu\n", sizeof(abc_1));
  printf("Size 2: %lu\n", sizeof(abc_2));
  printf("Size 3: %lu\n", sizeof(abc_3));
  printf("Addr 1: %p\n", abc_1);
  printf("Addr 2: %p\n", abc_2);
  printf("Addr 3: %p\n", abc_3);
}
```

Memory

Address	**Value**	
...	...	
0x7fec70ac	0x616263??	abc?
0x7fec70b0	0x61626300	abc\0
0x7fec70b4	0x61626300	abc\0
...	...	

Little endian representation:
Least significant byte first (right to left)

Structs

structs.c

```
#include <stdio.h>
struct person {
  int age;
  char* first_name;
  char* last_name;
};
int main()
{
  struct person p;
  printf("Size: %lu\n", sizeof(struct person));
  p.age = 54;
  p.first_name = "James";
  p.last_name = "May";
  printf("Size: %lu\n", sizeof(p));
  printf(" Address: %p\n", &p);
  printf("Age addr: %p\n", &p.age);
  printf(" FN addr: %p\n", p.first_name);
  printf(" LN addr: %p\n", p.last_name);
}
```

Console

```
$ - gcc structs.c -o run-structs
$ - ./run-structs
Size: 12
Size: 12
 Address: 0x7f2bba10
Age addr: 0x7f2bba10
 FN addr: 0x5670182f
 LN addr: 0x56701835
$ -
```

Structs

structs.c

```
#include <stdio.h>
struct person {
  int age;
  char* first_name;
  char* last_name;
};
int main()
{
  struct person p;
  printf("Size: %lu\n", sizeof(struct person));
  p.age = 54;
  p.first_name = "James";
  p.last_name = "May";
  printf("Size: %lu\n", sizeof(p));
  printf(" Address: %p\n", &p);
  printf("Age addr: %p\n", &p.age);
  printf(" FN addr: %p\n", p.first_name);
  printf(" LN addr: %p\n", p.last_name);
}
```

Declaring structure variable

Initializing structure members

* we can assign pointer to string,
it will point to the string in memory

Console

```
$ - gcc structs.c -o run-structs
$ - ./run-structs
Size: 12
Size: 12
 Address: 0x7f2bba10
Age addr: 0x7f2bba10
 FN addr: 0x5670182f
 LN addr: 0x56701835
$ -
```

Structs

structs.c

```c
#include <stdio.h>
struct person {
  int age;
  char* first_name;
  char* last_name;
};
int main()
{
  struct person p;
  printf("Size: %lu\n", sizeof(struct person));
  p.age = 54;
  p.first_name = "James";
  p.last_name = "May";
  printf("Size: %lu\n", sizeof(p));
  printf(" Address: %p\n", &p);
  printf("Age addr: %p\n", &p.age);
  printf(" FN addr: %p\n", p.first_name);
  printf(" LN addr: %p\n", p.last_name);
}
```

Console

```
$ - gcc structs.c -o run-structs
$ - ./run-structs
Size: 12
Size: 12
 Address: 0x7f2bba10
Age addr: 0x7f2bba10
 FN addr: 0x5670182f
 LN addr: 0x56701835
$ -
```

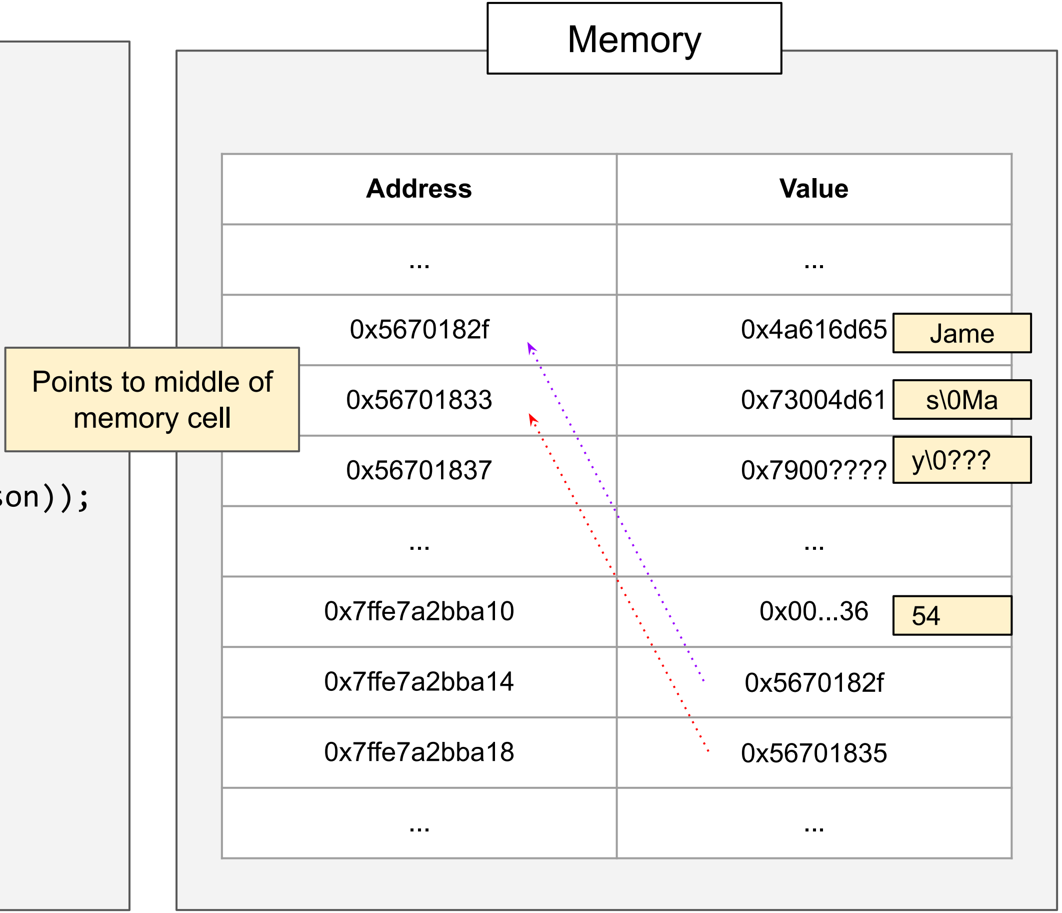
Where does 12 come from?

Why the same size after initialization?

Structs

structs.c

```
#include <stdio.h>
struct person {
  int age;
  char* first_name;
  char* last_name;
};
int main()
{
  struct person p;
  printf("Size: %lu\n", sizeof(struct person));
  p.age = 54;
  p.first_name = "James";
  p.last_name = "May";
  printf("Size: %lu\n", sizeof(p));
  printf(" Address: %p\n", &p);
  printf("Age addr: %p\n", &p.age);
  printf(" FN addr: %p\n", p.first_name);
  printf(" LN addr: %p\n", p.last_name);
}
```



Memory

Address	Value	
...	...	
0x5670182f	0x4a616d65	Jame
0x56701833	0x73004d61	s\0Ma
0x56701837	0x7900????	y\0???
...	...	
0x7ffe7a2bba10	0x00...36	54
0x7ffe7a2bba14	0x5670182f	
0x7ffe7a2bba18	0x56701835	
...	...	

Points to middle of memory cell

Functions and IO

factorial.c

```c
#include <stdio.h>
unsigned long fac(int n)
{
  if (n == 0){
     return 1;
  }else{
     return(n * fac(n-1));
  }
}

int main(){
  int n;
  printf("Enter a number:\n");
  scanf("%d", &n);
  if(n < 0) return -1;
  printf("Result: %lu\n", fac(n));
  return 0;
}
```

Console

```
$ - gcc factorial.c -o factorial
$ - ./factorial
Enter a number:
5
Result: 120
$ -
```

Functions and IO

factorial.c

```c
#include <stdio.h>
unsigned long fac(int n)
{
  if (n == 0){
      return 1;
  }else{
      return(n * fac(n-1));
  }
}

int main(){
  int n;
  printf("Enter a number:\n");
  scanf("%d", &n);
  if(n < 0) return -1;
  printf("Result: %lu\n", fac(n));
  return 0;
}
```

Takes a pointer to the variable because needs the address to place the value read from the console

Main function returns status codes.
0 means successful execution

Console

```
$ - gcc factorial.c -o factorial
$ - ./factorial
Enter a number:
5
Result: 120
$ -
```

Exercises

- 6 exercises on Toledo
- Every 15 minutes new solution is discussed
- Use slides as a first resource
  - Google is a great resource as well, try to understand found solutions

Self test (after session)

- After this session/before next session you should
  - Know what C is
  - Understand basic C data structures
  - Understand static memory allocation
  - Understand basic I/O
  - Write, compile and run simple C programs