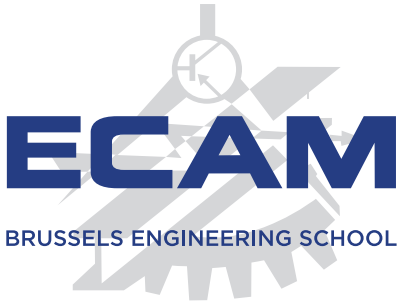


E301B C Programming

Session 3

Memory Structure and Pointer Manipulation

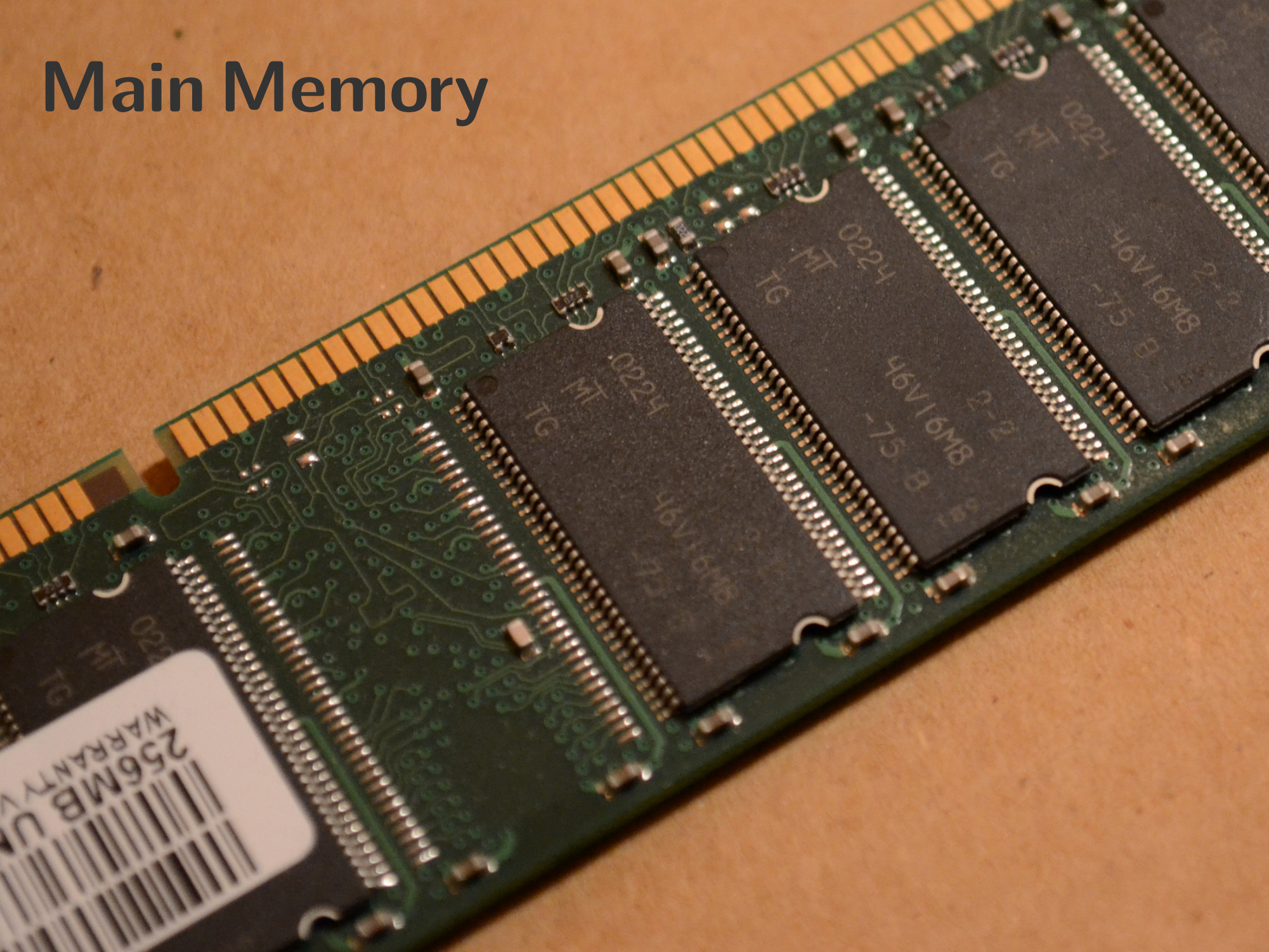
Sébastien Combéfis

Fall 2019

Objectives

- Processes and organisation of the main memory
  - Variable, addresses and pointers in the stack
  - Execution environment, procedure and function calls
  - Heap and creation of dynamic memory
- Pointer arithmetic and static/dynamic arrays
  - Pointer, variable address and dereferencing operator
  - Allocating memory with `malloc` and `free` functions

Main Memory



Main Memory (1)

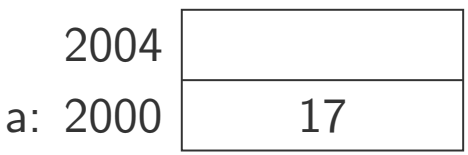
- A running program is called a process
  - A program is a sequence of bit stored on the hard drive
  - A process is in the main memory and running
- Memory space used by a process split in three parts
  - The text of the program with all the instructions
  - The stack with all the local variables
  - The heap with all the dynamically allocated memory
- Memory is just a huge bytes array

  Each byte has a unique address in memory

Main Memory (2)

- A variable is characterised by four elements
  - A memory address
  - The number of bytes used in memory
  - The value of the variable stored in memory
  - A symbolic name representing the variable

```
1  int a = 17;
```



Uninitialised Variable

- A variable just declared is said to be uninitialised

  Its value is undefined and depends on the external factors

- The main memory is not cleared upon process creation

```c
#include <stdio.h>

int main()
{
    int a;
    printf("%d\n", a);
    return 0;
}
```

```
277119030
```

Addressing

- Amount of addressable memory depends on several elements
  - Number of bits used to store an address
  - Addressing granularity (one for each bit, nibble, byte, etc.)
- Some common examples
  - 32 **bits** address for each byte: $2^{32} = 4$ *GiB*

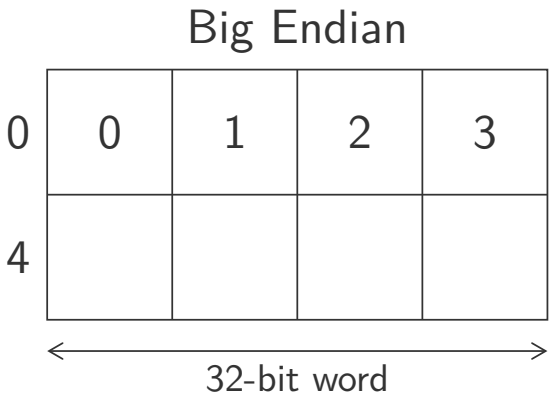
    Windows case that could not exceed 4 GiB of RAM
  - 64 **bits** address for each byte: $2^{64} = 16$ *EiB*

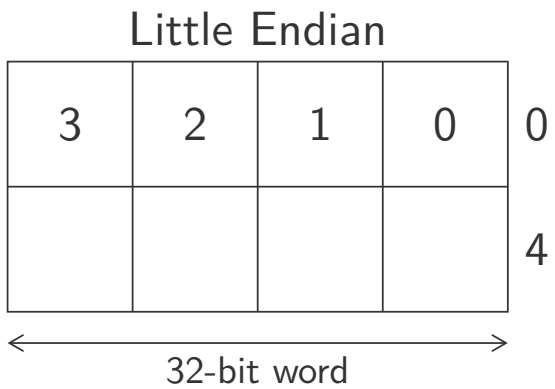
    We have time before seeing such RAM memory capacities

Endianness (1)

- Endianness defines the order in which bytes are organised

  For a given word that is stored in memory

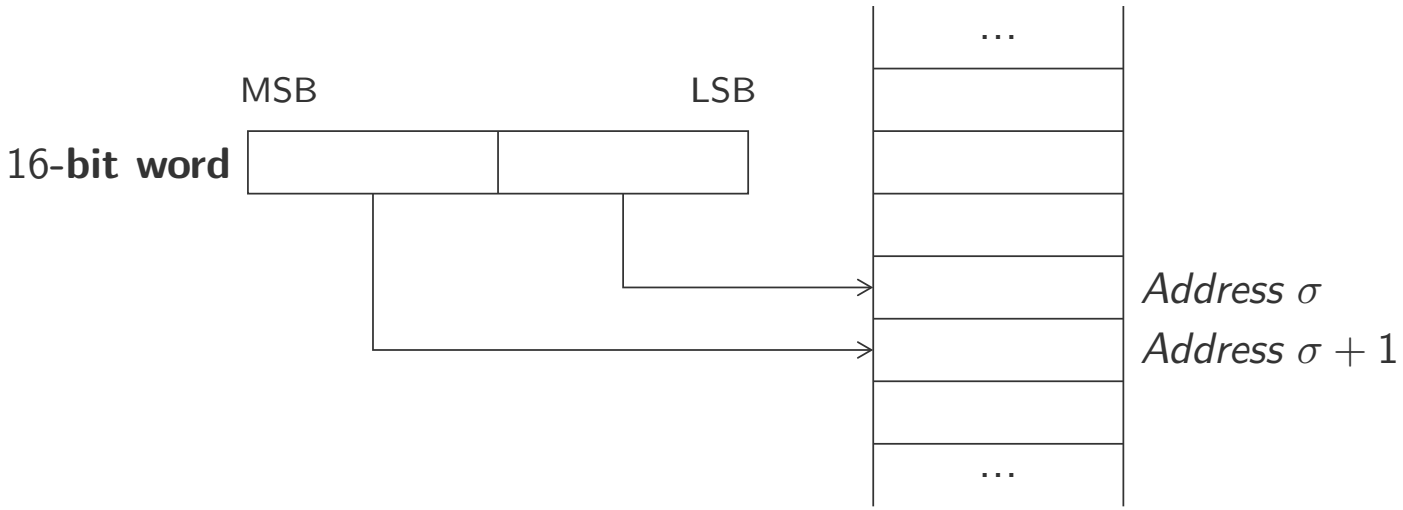




Endianness (2)

- Bytes are stored starting from LSB with little endian

  The bytes are stored "in reverse order"




Environment Stack

Execution Environment

- Running procedure/function has its own environment

  Contains memory space allocated for local variables

- Sizes known at compile time

  Compiler can compute this size for each procedure/function

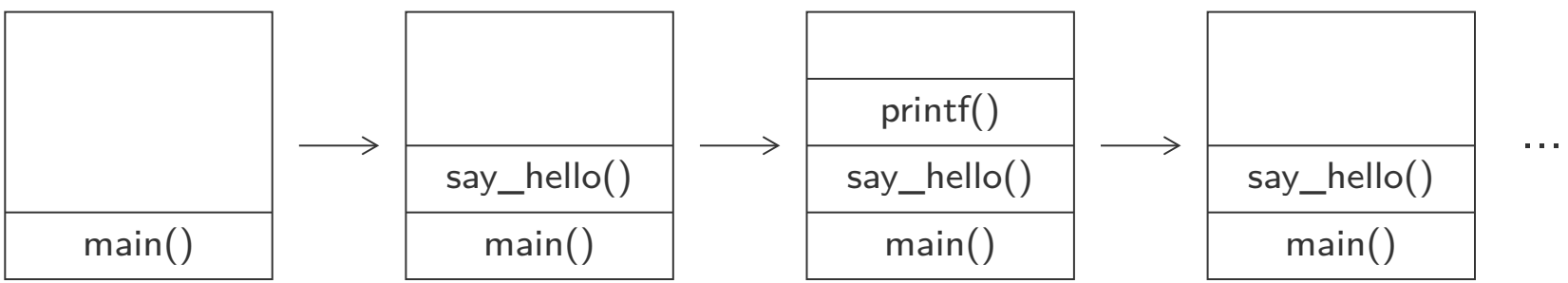
- Retain memory address where to put the return value

  Only used for functions by the `return` statement

Environment Stack

- Saving current environment at each procedure/function call
  - To not loose local variables from calling procedure/function
  - Restore environment after running called procedure/function
- Using an environment stack in memory

  Two operations on the stack: environment `push` *and* `pop`

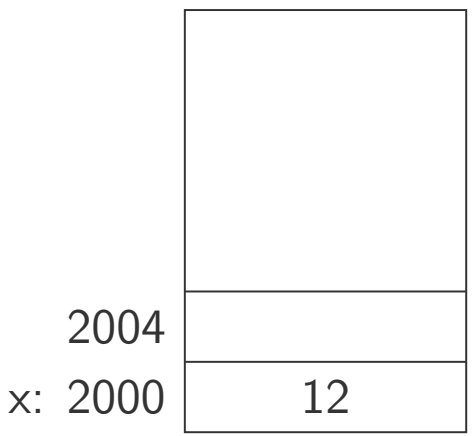


Procedure Call (1)

- The entry point of the program is the `main` function

  A local variable is required for this procedure

```
1  int main()
2  {
3      int x = 12;
4      print_sum(x, -2);
5
6      return 0;
7  }
```

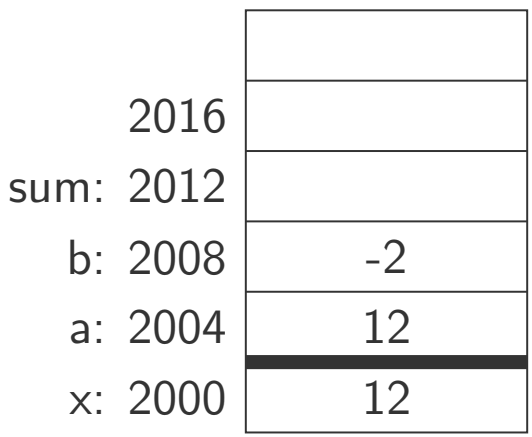


Procedure Call (2)

- Initialising parameters of `print_sum` during the call

  Value of effective parameters copied in formal parameters

```
1  void print_sum(int a, int b)
2  {
3      int sum = a + b;
4      printf("%d + %d = %d\n", a, b, sum);
5  }
```

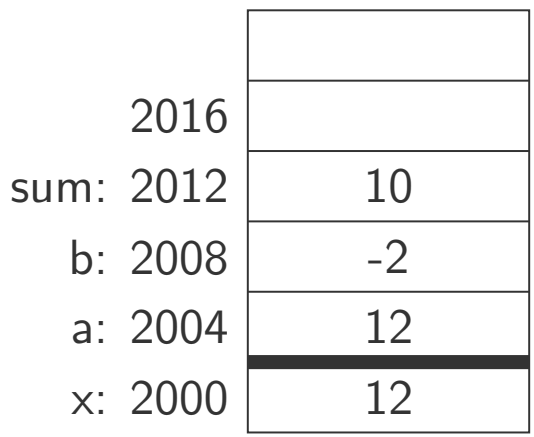


Procedure Call (3)

- Execution of the body of the `print_sum` procedure

  The only accessible memory is the local environment of the call

```
1  void print_sum(int a, int b)
2  {
3      int sum = a + b;
4      printf("%d + %d = %d\n", a, b, sum);
5  }
```

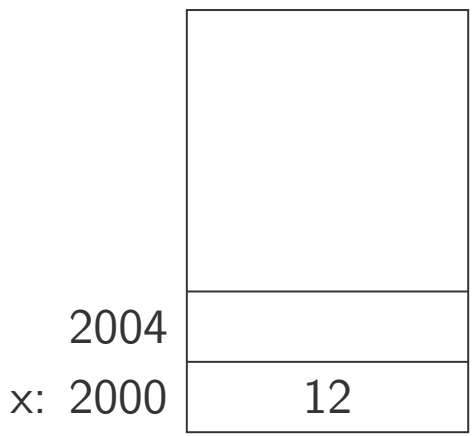


Procedure Call (4)

- After execution of `print_sum`, return to `main` function

  The environment of the `print_sum` call is deleted

```
1  int main()
2  {
3      int x = 12;
4      print_sum(x, -2);
5
6      return 0;
7  }
```



Stack Overflow

- Limitation of the memory area reserved for the stack

  The number of successive calls of procedures/functions is limited

- The precise limitation depends on the machine

  May depend on some physical constraints (register, memory, etc.)

```
1  int main()
2  {
3      main();
4      return 0;
5  }
```

```
[1]    70751 segmentation fault  ./a.out
```


TR
東洋緑化(株)
3M
Pointer

Pointer (1)

- A pointer is a variable containing an address
  - Unary operator `&` to get the address of a variable
  - Declaring a pointer with a `*` before the variable name
- A pointer is a variable pointing to another one

  It stores a reference to a memory cell

```
1  char c = 'M';
2  char *p = &c;
```

	2009	
p:	2001	2000
c:	2000	'M'

Dereferencing Operator (1)

- Dereference unary operator * (or indirection operator)

  Dereferencing a pointer means accessing to the pointed variable

- Dereferencing pointer type `xxx *` gives value with type `xxx`

  Otherwise, it produces a compile error

```
1  short a = 11;
2  short *p = &a;
3  short b = *p;
```

	2012	
b:	2010	11
p:	2002	2000
a:	2000	11

Pointer (2)

- It is possible to store the address of a pointer in a pointer

  Which gives two levels of indirection

```
1  short a = 11;
2  short *p = &a;
3  short b = *p;
4
5  short **pp = &p;
```

	2020	
pp:	2012	2002
b:	2010	11
p:	2002	2000
a:	2000	11

Pointer Type

- The * symbol indicates a pointer in the type of the variable

  Mandatory to indicate what type the pointer refers to

- The `value` variable is a pointer to an `int`

```
int *value;
```

- The `value` variable is a pointer to an `int`

```
int **value;
```

Dereferencing Operator (2)

- Modifying pointed variable value with dereferencing operator

  Just use it to the left of the assignment operator

```
1  short a = 11;
2  short *p = &a;
3  *p = 99;
```

	2010	
p:	2002	2000
a:	2000	99

Endianness (3)

- Example with a `short int` variable stored at address σ

  Accessing separately the two bytes with a cast

```c
#include <stdio.h>

int main()
{
    short s = 277; // 256 + 21
    char *c = (char*) &s;

    printf("%d\n", *c);
    printf("%d\n", *(c + 1));

    return 0;
}
```

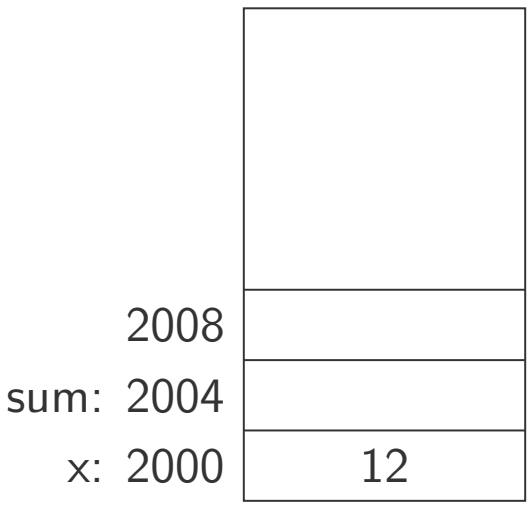
```
21
1
```

Function Call (1)

- The entry point of the program is the `main` function

  Two local variables is required for this function

```
1  int main()
2  {
3      int x = 12;
4      int sum = get_sum(x, -2);
5
6      return 0;
7  }
```

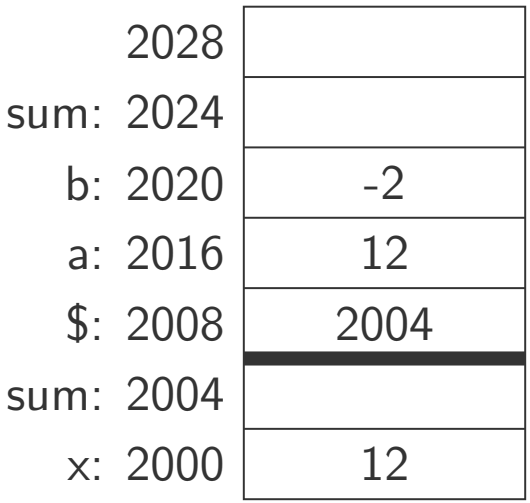


Function Call (2)

- Initialising parameters of `get_sum` during the call

  Storing return value address in a special variable $

```
1  int get_sum(int a, int b)
2  {
3      int sum = a + b;
4      return sum;
5  }
```

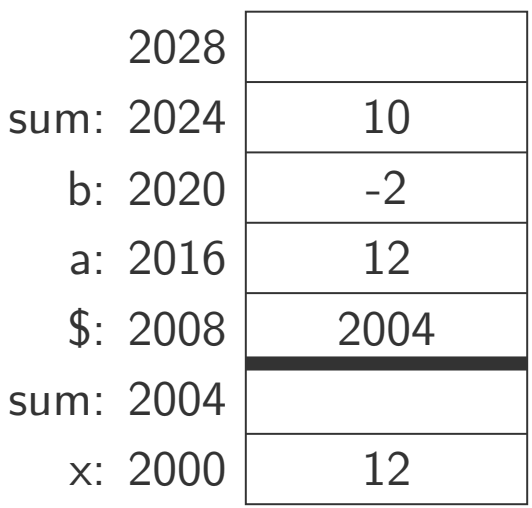


Function Call (3)

- Execution of the first instruction of the function `get_sum`

  The local variable `sum` is initialised with the result of `a + b`

```
1  int get_sum(int a, int b)
2  {
3      int sum = a + b;
4      return sum;
5  }
```

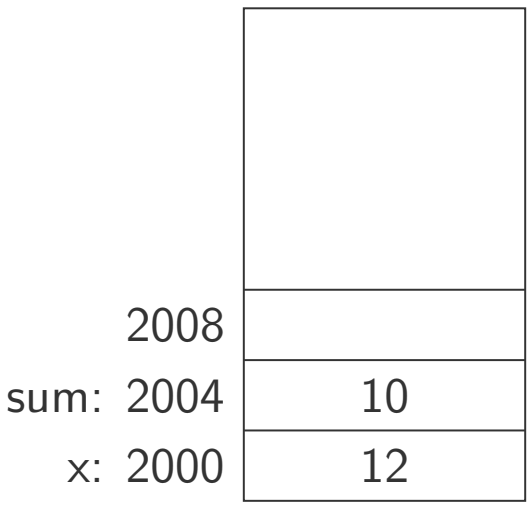


Function Call (4)

- After execution of `get_sum`, return to `main` function

  The `return` statement copies the return value ($ = sum)*

```
1  int main()
2  {
3      int x = 12;
4      int sum = get_sum(x, -2);
5
6      return 0;
7  }
```



Passing Parameter by Reference (1)

- Parameters passed by value when calling procedure/function

  Value of effective parameters copied in formal parameters

```
1  void swap(int a, int b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6  }
7
8  int main()
9  {
10     int x = 2, y = 15;
11     swap(x, y);
12     printf("x = %d et y = %d\n", x, y);
13
14     return 0;
15 }
```

Passing Parameter by Reference (2)

- Passing by reference if parameters are pointers

  Called procedure/function accesses values through addresses

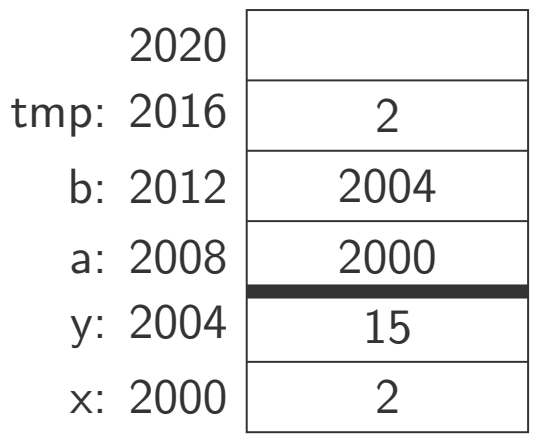
```
1  void swap(int *a, int *b)
2  {
3      int tmp = *a;
4      *a = *b;
5      *b = tmp;
6  }
7
8  int main()
9  {
10     int x = 2, y = 15;
11     swap(&x, &y);
12     printf("x = %d et y = %d\n", x, y);
13
14     return 0;
15 }
```

Passing Parameter by Reference (3)

- Execution of the first instruction of the `swap` procedure

  Local variable `tmp` initialised with value pointed by `a`

```
1  void swap(int *a, int *b)
2  {
3      int tmp = *a;
4      *a = *b;
5      *b = tmp;
6  }
```



Returning Pointer (1)

- Function that returns a pointer to a new variable

  Declaring and initialising a variable with a specified value

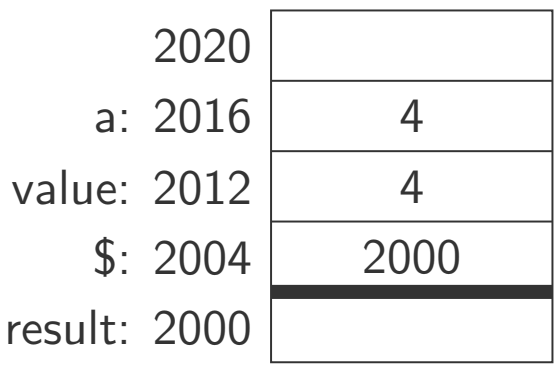
```
1  int* new_int(int value)
2  {
3      int a = value;
4      return &a;
5  }
6
7  int main()
8  {
9      int *result = new_int(4);
10     printf("%d\n", *result);
11     return 0;
12 }
```

Returning Pointer (2)

- Execution of the first instruction of the `swap` procedure

  Local variable `tmp` initialised with the value pointed by `a`

```
1  int* new_int(int value)
2  {
3      int a = value;
4      return &a;
5  }
```

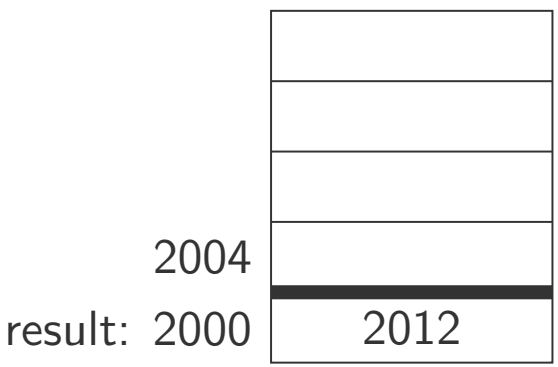


Returning Pointer (3)

- Return from the call and deletion of associated environment

The pointer `result` of `main` does not point anything

```
1  int main()
2  {
3      int *result = new_int(4);
4      printf("%d\n", *result);
5      return 0;
6  }
```

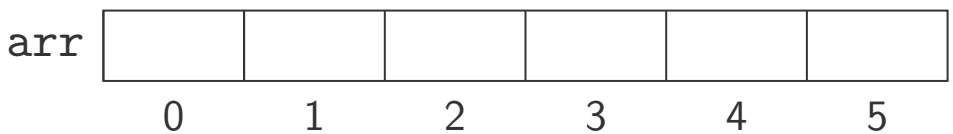



Array

Array

- An array can store a sequence of elements
  - All the elements have the same type (`int`, `double`, `char`, etc.)
  - A size N defining the number of elements
- Accessing the cell of the array with index `i` with `arr[i]`

  Indices of array cells goes from 0 *to* $N - 1$
- Example of an array `arr` with size 6



Array Operation

- Declaring an array with `int arr[N];`

  Specifying the type of elements and the size of the array

- Modification of an element with the assignment operator

  For example, `arr[i] = 3;` stores the value 3 in the index `i` cell

```
1  int N = 3;                    // Size of the array
2  int arr[N];                   // Declaration of an array of integers
3
4  int i;
5  for (i = 0; i < N; i++)
6     arr[i] = i + 1;            // Store values in the array
```

Array Initialisation

- Array cells values not initialised at declaration

  Manual initialisation or with an initialiser

- Direct initialisation of an array with initial values

  Only possible when declaring the array, not after

```
1  int a[3] = {3, 2, 1};          // Declaration and initialisation
2  a = {1, 2, 3, 4, 5}            // => FORBIDDEN
3
4  int b[10] = {1, 2};            // Specified size is an upper bound
5  int c[] = {1, 2, 3, 4, 5};     // Size can be automatically inferred
```

Displaying Array Elements

- Display elements of an array `arr` with a loop

  Aesthetic display in brackets, separating elements with commas

- Computing the size of the array with the `sizeof` operator

  `sizeof(type)` for type, `sizeof var` for variable, in bytes

```
1  int arr[] = {5, 4, 3, 2, 1};
2  int N = sizeof arr / sizeof(int);
3
4  if (N == 0)                          // Base case, empty array
5     printf("[]");
6  else {
7     printf("[%d", arr[0]);            // Display 1st element
8
9     int i;
10    for (i = 1; i < N; i++)
11       printf(", %d", arr[i]);        // If several elements
12    printf("]\n");
13 }
```

Array in Memory

- An array is a pointer to memory area of contiguous cells
  - Stored in the procedure/function environment in the stack
  - Constant pointer whose value cannot change

```
1  int a[3] = {3, 2, 1};
2
3  int *p = a;
4  printf("%d\n", p[0]);
5  a = p;                                  // => FORBIDDEN
```

	2020	
p:	2012	2000
	2008	1
	2004	2
a:	2000	3

Array Parameter (1)

- Pass an array parameter to a procedure/function

  Passing by reference, that is, as for a `int*`

```
1  void print_arr(int arr[])
2  {
3      int N = sizeof arr / sizeof(int);
4
5      // [...]
6  }
```

```
program.c:5:20: warning: sizeof on array function parameter will
return size of 'int *' instead of 'int []'
      [-Wsizeof-array-argument]
    int N = sizeof arr / sizeof(int);
                    ^
program.c:3:20: note: declared here
void print_arr(int arr[])
```

Array Parameter (2)

- The `sizeof` operator cannot access the stack of the caller

  `sizeof arr` returns the size of a `int*` (8 on a 64 bits machine)

- The `print_arr` function only displays the two first elements

  The value of `sizeof arr / sizeof(int)` is $8/4 = 2$

```
1  int main()
2  {
3      int a[3] = {3, 2, 1};
4      print_arr(a);
5
6      return 0;
7  }
```

```
[3, 2]
```

Array Parameter (3)

- Required to pass the size of the array in the parameters

  Two similar notations: `int arr[3]` *or* `int *arr`

```c
void print_arr(int arr[], int N)
{
    printf("%p\n", arr);
    // [...]
}

int main()
{
    int a[3] = {3, 2, 1};
    printf("%p\n", a);
    print_arr(&a, 3);

    return 0;
}
```

```
0x7fff5c96d84c
0x7fff5c96d84c
[3, 2, 1]
```

Array Parameter (4)

- Passing a pointer to a fixed size array as a parameter

  Access to array cells values by dereferencing the received pointer

```
1  void print_arr(int (*arr)[3])
2  {
3      int N = sizeof (*arr) / sizeof(int);    // Access to referenced array
4      if (N == 0)
5        printf("[]\n");
6      else {
7          printf("[%d", (*arr)[0]);           // Access to referenced array
8
9          int i;
10         for (i = 1; i < N; i++)
11             printf(", %d", (*arr)[i]);      // Access to referenced array
12         printf("]\n");
13     }
14 }
15
16 int main()
17 {
18     int a[3] = {3, 2, 1};
19     print_arr(&a);
20
21     return 0;
22 }
```

Careful with Types!

- Do not confuse pointers array and array pointeur
  - Pointers array: `int *p[2];`
  - Array pointer: `int (*q)[];`

```
1  int main()
2  {
3      int i = 12;
4
5      int *p[2];
6      p[0] = &i;
7      printf("%d\n", *p[0]);
8
9      int data[] = {99, 98, 97};
10
11     int (*q)[];
12     q = &data;
13     printf("%d\n", (*q)[0]);
14 }
```

Returning Array

- Impossible for a function to return an array

  Created on the stack and deleted when returning from function

- A function can return a subarray

  Returning the pointer to the start cell of the subarray

```
1  int* sub_arr(int *arr, int start)
2  {
3      return &arr[start];
4  }
5
6  int main()
7  {
8      int a[] = {1, 2, 3, 4, 5};
9      int *b = sub_arr(a, 2);
10     print_arr(b, 2);                    // Displays "[3, 4]"
11
12     return 0;
13 }
```

Pointer Arithmetic

Array and Pointer (1)

- Notions of pointer and array are very close

  Both are used to identify memory areas

```c
int main()
{
    int a = 1;
    int b = 2;
    int c = 3;

    int *p = &c;
    int i;
    for (i = 0; i < 3; i++)
        printf("%d\n", p[i]);

    return 0;
}
```

```
3
2
1
```

Array and Pointer (2)

- An array is a block of memory cells that are contiguous

An array variable is a pointer to the beginning of the block

```c
int arr[] = {1, 2};

printf("%p\n", arr);

printf("%d\n", arr[0]);
printf("%d\n", *arr);
```

```
0x7fff58e7d840
1
1
```

	2008	
	2004	2
arr:	2000	1

Notation Equivalence (1)

- Equivalent notations between array and pointers

  Using brackets and indices, or direct computation

- Example given the `int arr[];` declaration

arr[i]	*(arr + i)
&arr[i]	arr + i

Pointer Arithmetic (1)

- Pointer arithmetic proposes several operations

  The + operator is used to move in the memory

- Two main ways to move in an array
  - Navigation with the [] notation and an index
  - Navigation with the pointer and pointer arithmetic

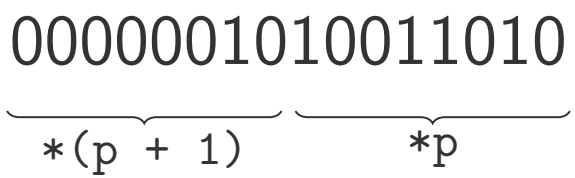
```
1 for (i = 0; i < N; i++)
2    printf("%d\n", tab[i]);
```

```
1 for (i = 0; i < N; i++)
2    printf("%d\n", *(tab + i));
```

+ Operator

- The + operator moves the pointer forward of one memory cell
  - Bytes increment to the address depends on the pointer type
  - No control on the value of the pointer

```
1  short s = 666;
2  char *p = (char*) &s;
3
4  printf("%d\n", *p);            // Display "-102"
5  printf("%d\n", *(p + 1));      // Display "2"
```



Moving in Memory (1)

- The + operator goes to the next "cell"

  Of the memory area indicated by the pointer

```
1  int arr[] = {1, 2, 3, 4};
2
3  printf("%d\n", *arr);            // arr is 2000
4  printf("%d\n", *(arr + 3));      // arr + 3 is 2012 (2000 + 3 * 4)
```

	2016	
	2012	4
	2008	3
	2004	2
arr:	2000	1

Moving in Memory (2)

- Possible to modify values in memory where we want

  Just use the dereferencing operator on a pointer

```
1  int arr[4];
2
3  int i;
4  for (i = 0; i < 4; i++)
5      *(arr + i) = 4 - i;
```

	2016	
	2012	1
	2008	2
	2004	3
arr:	2000	4

Pointer Arithmetic (2)

- Three categories of operations possible on pointers
  - Addition and subtraction on a pointer with + and -
  - Increment (decrement) a pointer with ++ (--)
  - Comparing pointers with ==, <, >, <= and >=
- Browse an array with start and end pointers

  End pointer can be computed given the size of the array

```
1  int tab[] = {1, 2, 3, 4, 5};
2  int *start = &tab[0];
3  int *end = start + 4;
4
5  while (start <= end)
6     printf("%d\n", *start++);
```

Segmentation Fault

- You must pay attention to illegal access to memory

  Each process has its own memory address space

- Memory accesses are controlled by the operating system

  Thanks to a hardware component, namely the MMU

```
1  int a = 999;
2  int b = 0;
3
4  printf("%d\n", *(&b + 1));    // Displays "999"
5  printf("%d\n", *(&b + 500)); // 9679 segmentation fault  ./a.out
```

Operator Priority

- You have to pay attention to the operator priorities

  In particular the priority of the dereferencing operator `*`

- Example given the declaration `int *p;`

`(*p)++`	`*(p++)`
`*p + 2`	`*(p + 2)`

NULL Pointer

- Only integer literal value that can be used for a pointer is 0
  - Indicates the absence of a pointed memory area
  - Represented by the `NULL` constant
- Possible to test the `NULL`ity of a pointer

  Using the == and != operators

```
1  int *p = NULL;
2
3  if (p != NULL)
4     printf("Valeur de p: %d\n", *p);
```

Dynamic Size Array (1)

- Using `NULL` to indicate the end of an array

  Array of pointers in which the last one is followed by a `NULL`

```
1  int main()
2  {
3      int a = 6;
4      int b = -2;
5
6      int *arr[10] = {&a, &b};
7      arr[2] = NULL;
8
9      return 0;
10 }
```

	2032	
	2024	0
	2016	2004
arr:	2008	2000
b:	2004	-2
a:	2000	6

Dynamic Size Array (2)

- Possible to create manipulation utility functions

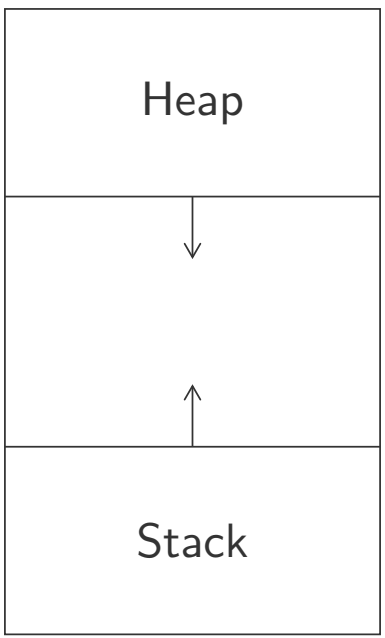
  For example, getting the size of the array by looking for the `NULL`

```
1  int length(int *arr[])
2  {
3      int i = 0;
4      while (arr[i++] != NULL);
5      return i;
6  }
7
8  void print_arr(int *arr[])
9  {
10     if (*arr == NULL)
11         printf("[]\n");
12     else {
13         printf("[%d", **arr);
14
15         while (*(++arr) != NULL)
16             printf(", %d", **arr);
17         printf("]\n");
18     }
19 }
```

Dynamic Memory

Dynamic Memory (1)

- There are two types of memory in a program
  - The **stack** contains local variables
  - The **heap** contains dynamic variables



Dynamic Memory (2)

- Memory management functions in `stdlib.h`
  - Creating a dynamic memory area with `malloc`
  - Freeing the allocated memory area with `free`

```
1  int *p = malloc(sizeof(int));
2
3  printf("%p\n", p);
4  *p = 12;
5  printf("%d\n", *p);
6
7  free(p);
```

```
0x7fdf58c03970
12
```

malloc Function

- The malloc function can be used to allocate memory space
  - Return the start address of the allocated memory block
  - Specifying the desired size in bytes
- The demand can be refused if the heap collides with the stack

  If unsuccessful, returns a NULL pointer

```
1  double *data = malloc(10 * sizeof(double));
2
3  if (data != NULL)
4  {
5     // ...using the allocated memory area...
6  }
```

`free` Function

- The `free` function can be used to deallocate memory space
  - Free space for future memory space allocation demands
  - Does not change the pointer, only invalidate the position
- Importance of freeing any memory area not used anymore

  Undefined behaviour if freeing a wrong pointer

```
1  double *data = malloc(10 * sizeof(double));
2
3  if (data != NULL) {
4     // ...using the allocated memory area...
5
6     free(data);
7  }
```

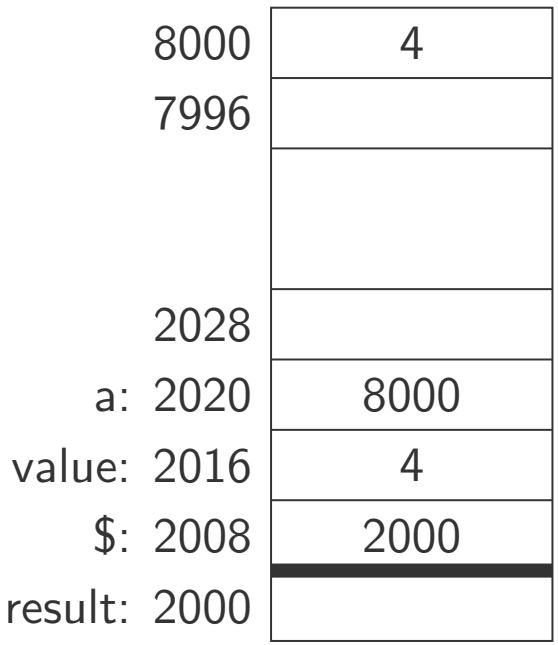
Returning Pointer (4)

- The dynamic memory can be used to share variables
  - Memory accessible from all the procedures/functions
  - Not deleted after the execution of procedures/functions
- Possibility to write a function returning a pointer

  Just point to a dynamic memory area

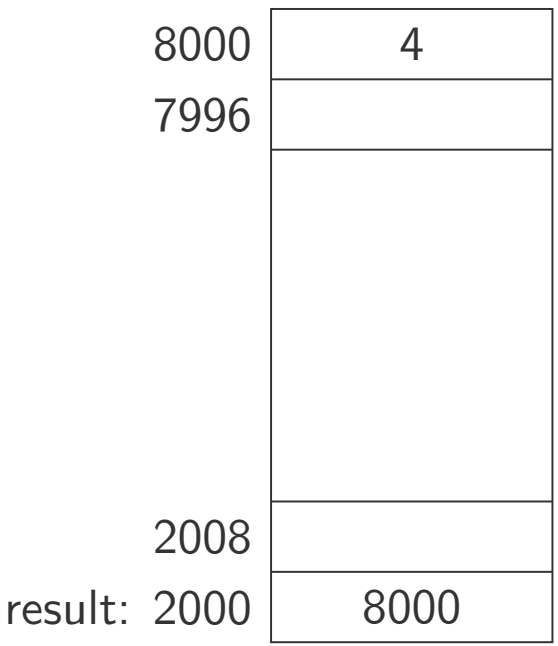
Returning Pointer (5)

```
1  int* new_int(int value)
2  {
3      int *a = malloc(sizeof(int));
4      *a = value;
5      return a;
6  }
```



Returning Pointer (6)

```
1  int main()
2  {
3      int *result = new_int(4);
4      printf("%d\n", *result);
5      free(result);
6
7      return 0;
8  }
```



calloc and realloc Functions

- The calloc function creates a memory area initialised to 0

```
1  double *data = calloc(5, sizeof(double));
2  if (data != NULL) {
3      // ...using the allocated memory area...
4      free(data);
5  }
```

- The realloc function modifies the size of a memory area

```
1  double *data = calloc(5, sizeof(double));
2  if (data != NULL)
3  {
4      data = realloc(data, 10 * sizeof(double));
5      // ...using the allocated memory area...
6      free(data);
7  }
```

Dynamic Array

- Memory allocated by `malloc` seen as a dynamic array

```
1  int N = 3;
2  int *data = malloc(N * sizeof(int));
3
4  int i;
5  for (i = 0; i < N; i++)
6     data[i] = i + 1;
7  print_arr(data, N);
```

	8008	3
	8004	2
	8000	1
	7996	
	2016	
i:	2012	3
data:	2004	8000
N:	2000	3

Returning Array

- Returning an array must be done with dynamic memory

  Do not forget to free the memory after use

```
1  int* new_tab(int size)
2  {
3      return calloc(size, sizeof(int));
4  }
5
6  int main()
7  {
8      int *data = new_tab(3);
9      // ...using the allocated memory area...
10     free(data);
11
12     return 0;
13 }
```

References

- Shohei Yokoyama, *Understanding Memory Layout*, November 10, 2018.
  https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709
- Neotam, *Endianness. Little Endian vs Big Endian*, April 6, 2019.
  https://getkt.com/blog/endianness-little-endian-vs-big-endian
- Colin Walls, *Endianness*, October 16, 2015.
  https://www.embedded.com/design/mcus-processors-and-socs/4440613/Endianness
- Puneet Sapra, *No more fear of pointers: Coding Allusion*, February 4, 2018.
  https://medium.com/the-mighty-programmer/variable-and-pointer-fb637566bfd9

Credits

- Brian Turner, February 21, 2013, https://www.flickr.com/photos/60588258@N00/8552885919.
- Colin, September 9, 2018, https://www.flickr.com/photos/colinsd40/43917455784.
- Jen, July 27, 2013, https://www.flickr.com/photos/nstop/9406739132.
- kishjar?, November 8, 2013, https://www.flickr.com/photos/kishjar/10747531785.
- Marc Kjerland, January 12, 2010, https://www.flickr.com/photos/marckjerland/4285433391.
- Orin Zebest, May 27, 2006, https://www.flickr.com/photos/orinrobertjohn/154616556.