Pointers And Addressing

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

UNIVERSITY OF GREENWICH

Download as a PDF

Principle of Locality

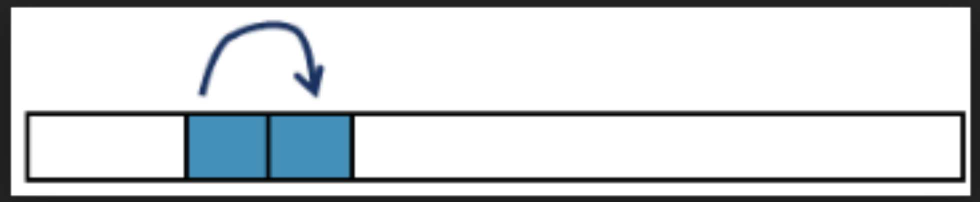
Programs tend to use data and instructions with addresses near or equal to those they have used recently

Temporal

- Recently referenced items are likely to be referenced again in the near future

Spatial

- Items with nearby addresses tend to be referenced close together in time



Example

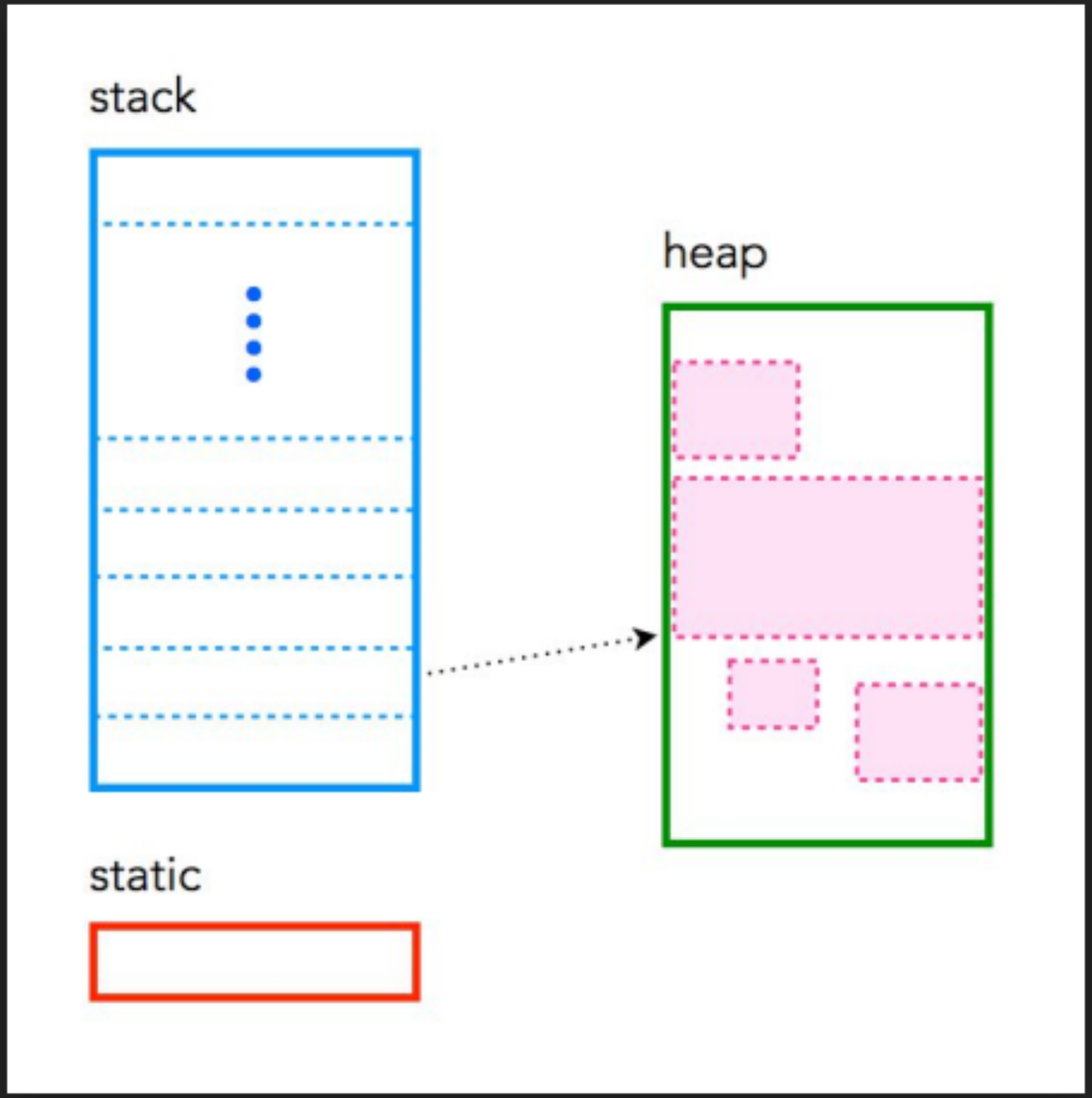
- **Spatial**
  - Access array elements `a[i]` in succession – Data
  - Reference instructions in sequence – Instruction
- **Temporal**
  - Reference `sum` each iteration – Data
  - Cycle through loop repeatedly – Instruction

```c
int main(){

  int sum = 0;
  int a[5];

  for ( int i = 0; i < n; i++ )
  {
    sum += a[i];
  }

  return 0;
}
```

Stack, Static and Heap

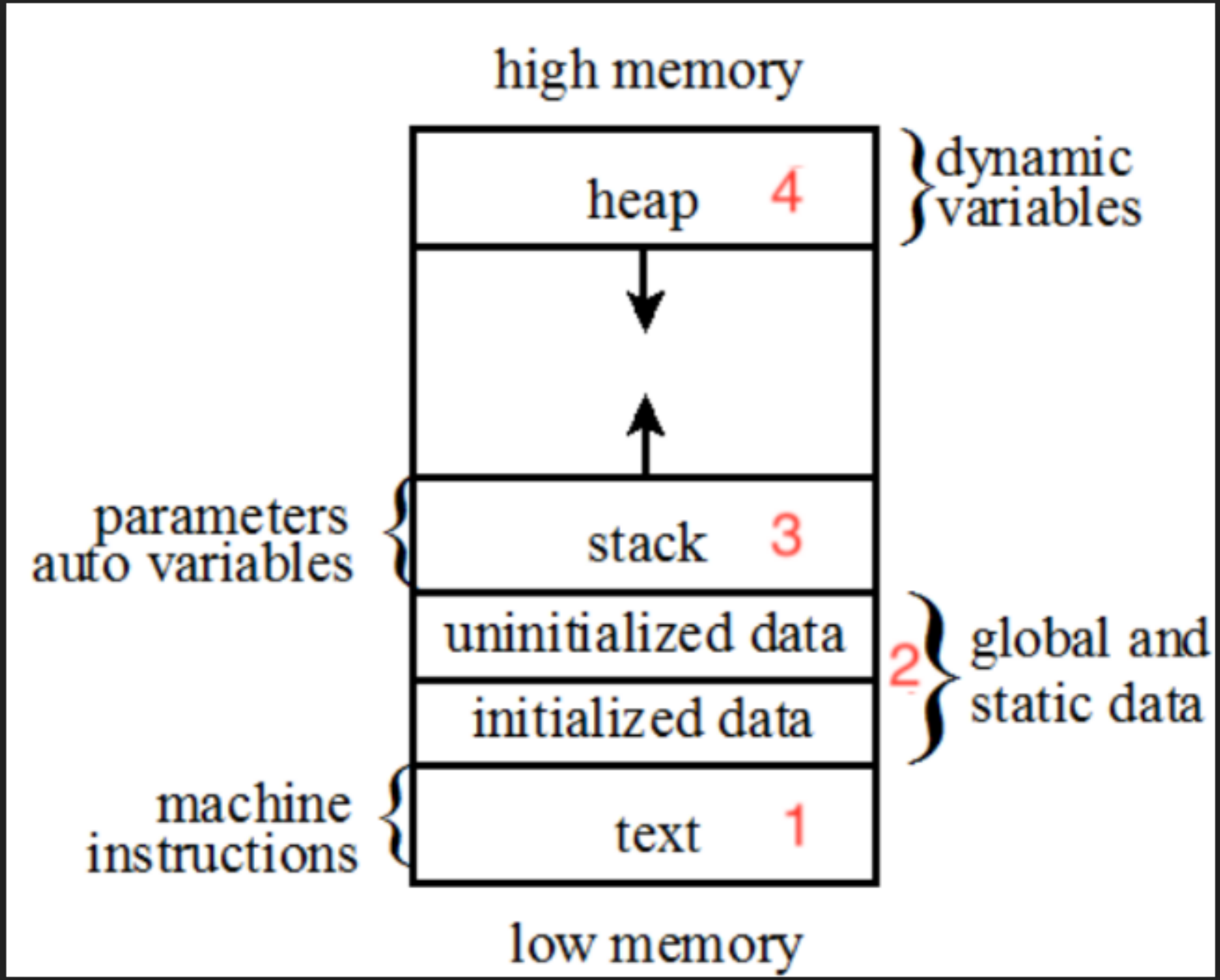
The great thing about C is that it is so intertwined with memory – and by that I mean that the programmer has quite a good understanding of “what goes where”. C has three different pools of memory.

- **static**: global variable storage, permanent for the entire run of the program.
- **stack**: local variable storage (automatic, continuous memory).
- **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).



General Memory Layout

- [1] **text:** stores the code being executed
- [2] **data:** stores global variables, separated into initialised and uninitialised
- [3] **stack:** stores local variables
- [4] **heap:** dynamic memory for programmer to allocate



Static

- Static memory persists throughout the entire life of the program, and is usually used to store things like global variables, or variables created with the static clause.

- On many systems this variable uses 4 bytes of memory. This memory can come from one of two places. If a variable is declared outside of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are static,

```c
#include <stdio.h>

// Global variable
int globalVar = 10;

void demoFunction() {
    int localVar = 5;

    static int staticLocalVar = 5;

    // Incrementing static local variable
    localVar++;
    staticLocalVar++;

    // Printing values and memory addresses
    printf("Local Variable: %d, Address: %p\n", localVar, &localVar);
    printf("Global Variable: %d, Address: %p\n", globalVar, &globalVar);
    printf("Static Local Variable: %d, Address: %p\n", staticLocalVar, &staticLocalVar);
}

int main() {
    demoFunction();
    return 0;
}
```

```
❯ ./global.exe
Global Variable: 10, Address:       00007FF6F3403000
Static Local Variable: 6, Address:  00007FF6F3403004
Local Variable: 6, Address:         000000CA637FF88C
```

Static

- Static memory persists throughout the entire life of the program, and is usually used to store things like global variables, or variables created with the static clause.

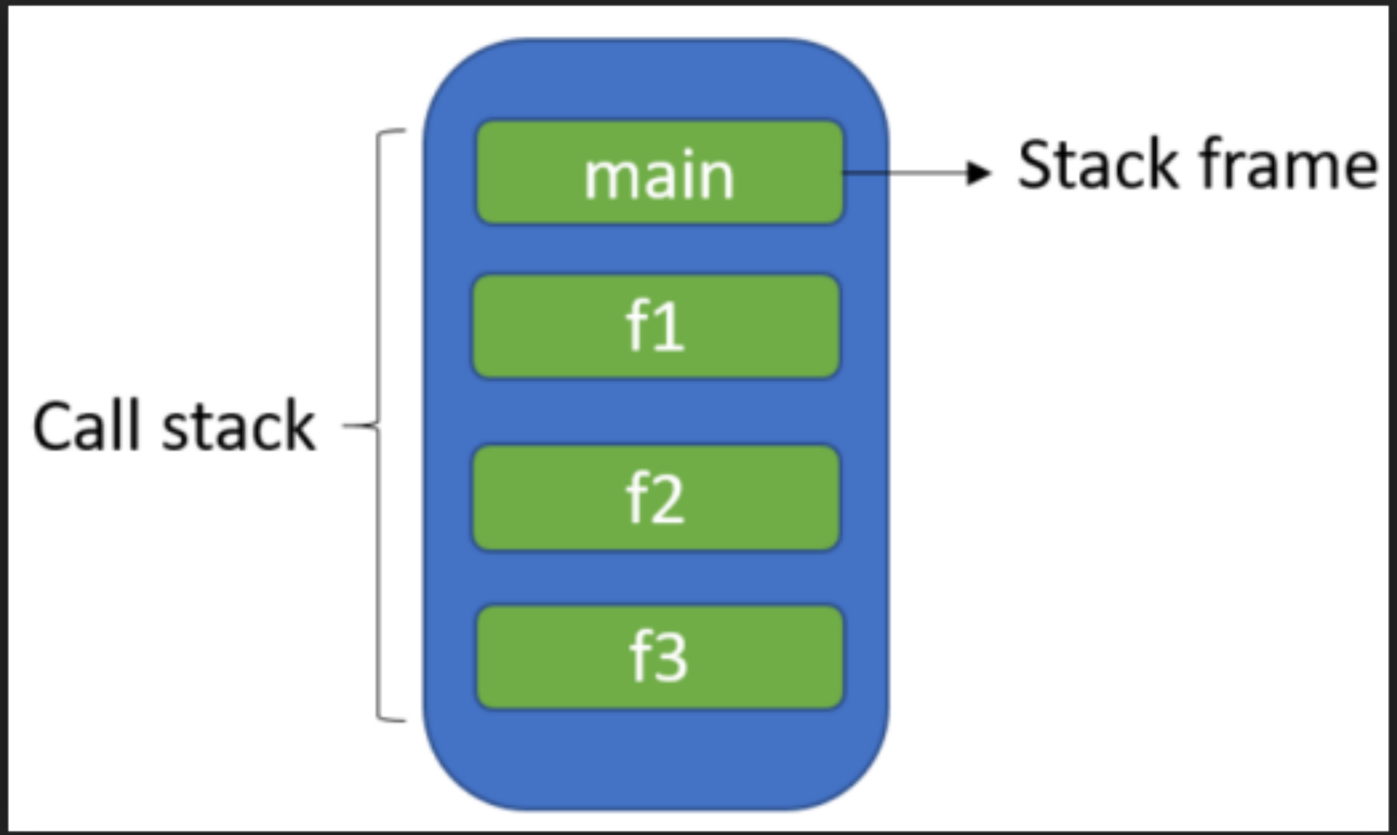
- On many systems this variable uses 4 bytes of memory. This memory can come from one of two places. If a variable is declared outside of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are static,

```c
#include <stdio.h>

// Global variable
int globalVar = 10;

void demoFunction() {
    int localVar = 5;

    static int staticLocalVar = 5;

    // Incrementing static local variable
    localVar++;
    staticLocalVar++;

    // Printing values and memory addresses
    printf("Local Variable: %d, Address: %p\n", localVar, &localVar);
    printf("Global Variable: %d, Address: %p\n", globalVar, &globalVar);
    printf("Static Local Variable: %d, Address: %p\n", staticLocalVar, &staticLocalVar);
}

int main() {
    demoFunction();
    return 0;
}
```

```
❯ ./global.exe
Global Variable: 10, Address:       00007FF6F3403000
Static Local Variable: 6, Address: 00007FF6F3403004
Local Variable: 6, Address:         000000CA637FF88C
```

Stack

- The stack is managed by the CPU, there is no ability to modify it
- Variables are allocated and freed automatically
- The stack it not limitless - most have an upper bound
- The stack grows and shrinks as variables are created and destroyed
- Stack variables only exist whilst the function that created them exists



Stack

- It’s a `LIFO`, “Last-In,-First-Out”, structure. Every time a function declares a new variable it is “pushed” onto the stack.
- The stack is managed by the CPU, there is `no ability` to modify it
- Variables are allocated and freed `automatically`
- The stack it not limitless – most have an `upper bound`
- The stack `grows and shrinks` as variables are created and destroyed
- Stack variables only exist `whilst` the `function` that created them `exists`

Stack Overflow

- A stack overflow occurs if the call stack pointer exceeds the stack bound.

- The call stack may consist of a limited amount of address space, often determined at the start of the program.

Heap

The heap is the diametrically `opposite of the stack`.

- The heap is **managed** by the **programmer**, the ability to modify it is somewhat boundless
- The heap is large, and is usually **limited** by the **physical memory** available in an embedded environment and in a PC it is stored within paging files on main memory (SSD)
- This is memory that is not automatically managed
  – you have to explicitly allocate (using functions such as `malloc()`, `calloc()`, `realloc()`), and deallocate (`free()`) the memory.
- The heap **requires pointers** to access it

```c
#include <windows.h>
#include <stdio.h>
#include <malloc.h>

int main() {
    _HEAPINFO hinfo;
    int heapstatus;
    hinfo._pentry = NULL;

    size_t total_allocated = 0;

    while ((heapstatus = _heapwalk(&hinfo)) == _HEAPOK) {
        total_allocated += hinfo._size;
    }

    if (heapstatus == _HEAPEND) {
        printf("Total heap space allocated: %zu bytes\n", total_allocated);
    }

    return 0;
}
```

```
❯ ./heap.exe
Total heap space allocated: 84098 bytes
```

Example

- `char *str;`

  This declares a pointer to a character.

  At this point, str points nowhere useful — it's uninitialised.

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

int main(void)
{
    int y = 4; char *str;

    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    for(int i =0; i< 100; i++)
    {
        printf("heap memory: %c\n", str[i]);
    }
    free(str);
    printf("heap memory: %c\n", str[0]);
    return 0;
}
```

UNIVERSITY OF GREENWICH

Example

- `malloc(100 * sizeof(char))`

  malloc allocates heap memory dynamically.

- `sizeof(char)` is always `1`, so this is just `malloc(100)`.

  So we are allocating 100 bytes of memory on the heap for `str` to point to.

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

int main(void)
{
    int y = 4; char *str;

    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    for(int i =0; i< 100; i++)
    {
        printf("heap memory: %c\n", str[i]);
    }
    free(str);
    printf("heap memory: %c\n", str[0]);
    return 0;
}
```

Example

- `free()` is a standard C library function used to deallocate heap memory that was previously allocated with

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

int main(void)
{
    int y = 4; char *str;

    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    for(int i =0; i< 100; i++)
    {
        printf("heap memory: %c\n", str[i]);
    }
    free(str);
    printf("heap memory: %c\n", str[0]);
    return 0;
}
```

UNIVERSITY OF GREENWICH

Memory Allocation

- a `char`acter, 1 byte of memory which is:

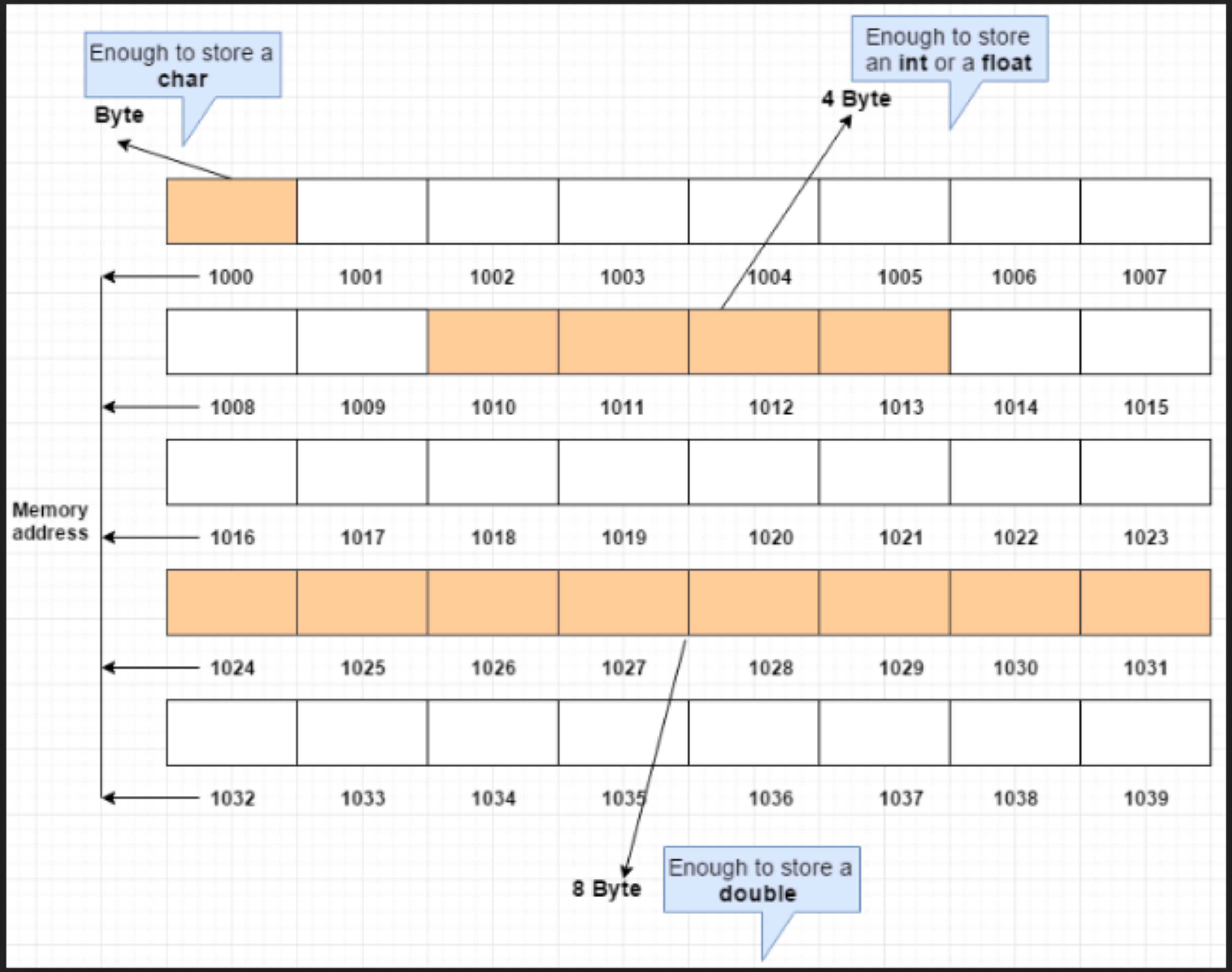
$$8\,bit = 1 * 8$$

- an `int`eger or a `float`, 4 byte of memory which is:

$$32\,bit = 4 * 8$$

- a `double` value, 8 byte of memory which is :

$$64\,bit = 8 * 8$$



Memory Allocation: Pointers and Addresssing

- In C/C++/C# you can access a variables address using the `&` and `*` symbol.
- With ‘address of’ `&` we can reference the variable’s address when used with itself.
- A ‘pointer’ `*` is a variable that stores the address of another variable.
- Be warned, playing with unprotected memory is dangerous and can cause systems to crash and even become unrecoverable.

Memory Allocation: C

```
int main ()
{ // The variable has its own address (unknown to us now)
  int n = 11;
  // this variable stores the address of the other variable
  int *ptrToN = n;
  printf("n's address: %d and %d ptrToN value \n", &n, ptrToN);
  printf("n's value: %d and ptrToN points to value %d \n", n, *ptrToN);
  return 0;
}
```

```
n's address: 0x7fff20494e4c and 0x7fff20494e4c ptrToN value
n's value: 11 and ptrToN points to value 11
```

Memory Allocation Array: C

```
int main ()
{
  int n = 11, i;
  char ptr[11] = "hello world";

  for (i = 0; i < n; i++)
  {
    printf ("\t%p       ||    ptr[%d]     =     %c\n", &ptr[i],i,ptr[i]);
  }

  printf("\t%p       ||    ptr[]      =  %c \n", &ptr,*ptr);

  return 0;
}
```