Memory

...

Axel Buendia
(axel.buendia@cnam.fr)

Different memories?

- RAM
  - the stack: used for local variables with limited scope, very fast
  - the heap: used for persistent variables (created with `new`), much slower
  - the registers: used for very local and basic typed variables, fastest

The STACK

```
void Function () {
    Vector x;
}
```

- x is allocated on the stack
- memory allocation is very fast (because of the allocator)
- the size of the stack is limited (stack overflow error)
- at the end of the variable scope, the memory is freed automatically
- stack is unique to each thread (not shared)

STACK allocator

- Process as a stack
- Next free space?
  - address of the top of the stack
  - shift the top of the stack according to the needed size
- Delete
  - just free the memory attached to the pointer address
  - do not shift the top of the stack
- Pros
  - very fast
- Cons
  - limited size
  - does not manage defragmention of the memory

The HEAP

```
void Function () {
    Vector* x = new Vector();
}
```

- x is allocated on the heap
- memory allocation is much slower (because of the allocator)
- the size of the heap is almost unlimited
- at the end of the variable scope, the memory is kept, the desallocation must be made manually
- heap is shared among the threads

HEAP allocator

- Complex allocator
- Next free space?
  - complex management of holes in memory
  - bookkeeping of free spaces
- Delete
  - just free the memory attached to the pointer address
  - register the new hole in the book keeping
- Pros
  - unlimited size
  - manage defragmention of the memory
- Cons
  - much slower

The REGISTERS

```
void Function () {
    register int i = 0;
}
```

- 'i' may be stored in a register (compiler choice)
- read and write are the fastest (faster than in RAM)
- very limited (depend on the processor), only basic types
- Registers have no memory address
- NB: 'register' keyword is deprecated

REGISTERS

- How to use them?
  - only basic types (int then float)
  - variable usage in the scope
  - no access to variable address
- To check look at the assembly
- Use inline assembly asm{}

Memory Alignment

- To enhance performance
- RAM <-> BUS <-> CPU
- Example
  - Bus 4 octets (32 bits)
  - Data 4 octets on address % 4 !=0
    - alignment violation
    - read 2 times 4 octets and rebuild the value
- Alignment: address % size = 0
- Be careful with bit fields
- Subtleties:
  - CPU cache size
  - virtual memory pagination
- Solutions
  - look at compiler options
  - add manual padding

Bit Field

```
struct Date {
    unsigned int weekDay  : 3;  //0..7 (3 bits)
    unsigned int monthDay : 6;  //0..31 (6 bits)
    unsigned int month    : 5;  //0..12 (5 bits)
    unsigned int year     : 8;  //0..100 (8 bits)
}
```

- Bit field regroup several members which sizes are less than an int
- each bit field member is declared with its size in bits
- type of a bit field can only be char, short, int, long, long long or enum
- the size of the bit field member cannot exceed the underlying type (other bits are used as padding)
- bit field members have no address
- bit field are signed or unsigned by default depending on the compiler
- passing the range of a bit field generates strange behaviors depending on the compiler

Alignment & Bit Fields

```
struct Date {
    unsigned int weekDay  : 3;  //0..7 (3 bits)
    unsigned int monthDay : 6;  //0..31 (6 bits)
    unsigned int month    : 5;  //0..12 (5 bits)
    unsigned int year     : 8;  //0..100 (8 bits)
    // total              : 22 bits
    unsigned int padding  : 10; //(10 bits to match 32 bits)
}
```

- to force memory alignment, padding is added to match the next CPU meaningful size (here 32 bits)
- an unnamed 0 sized bit field member can be used to force padding