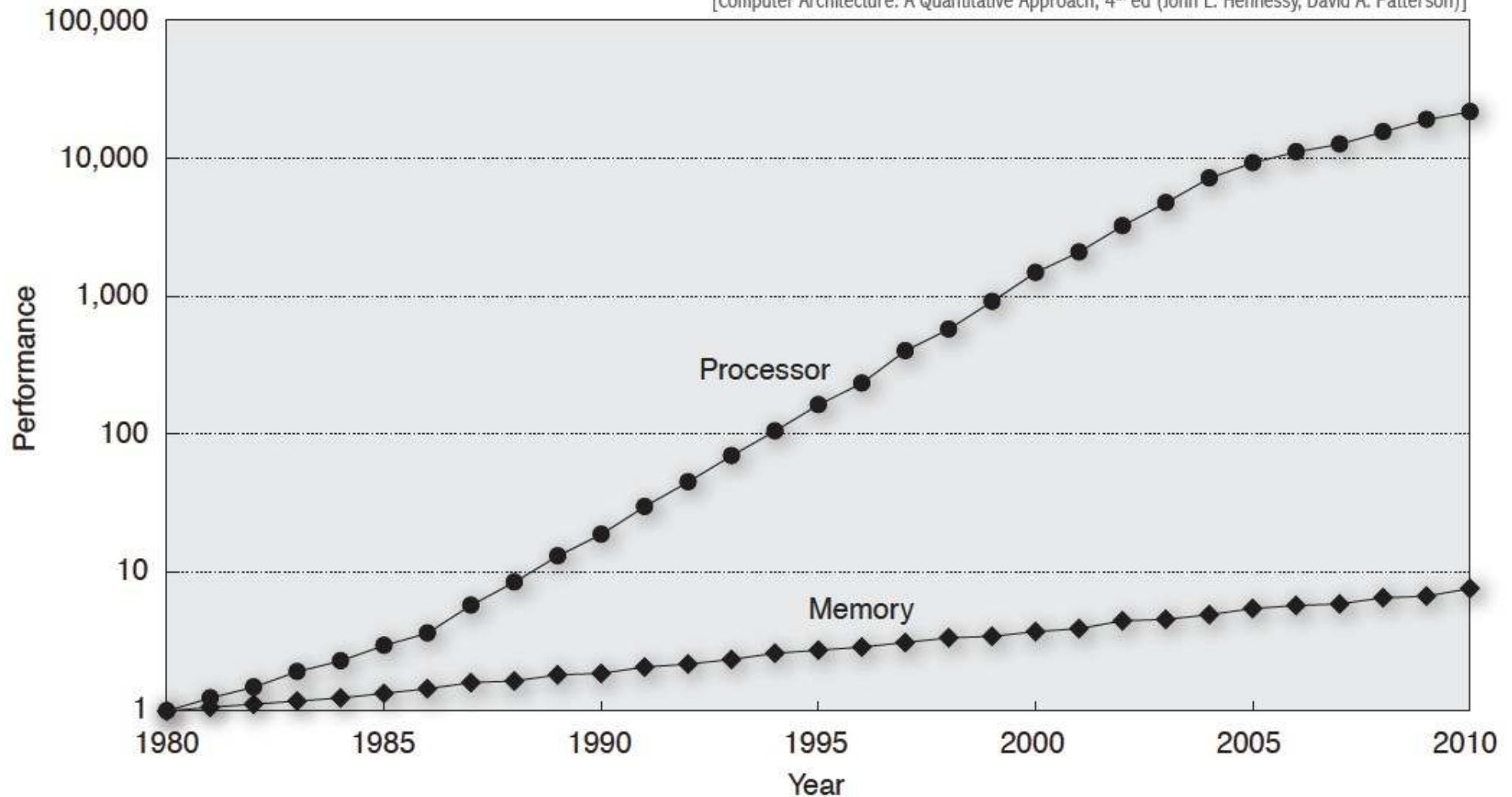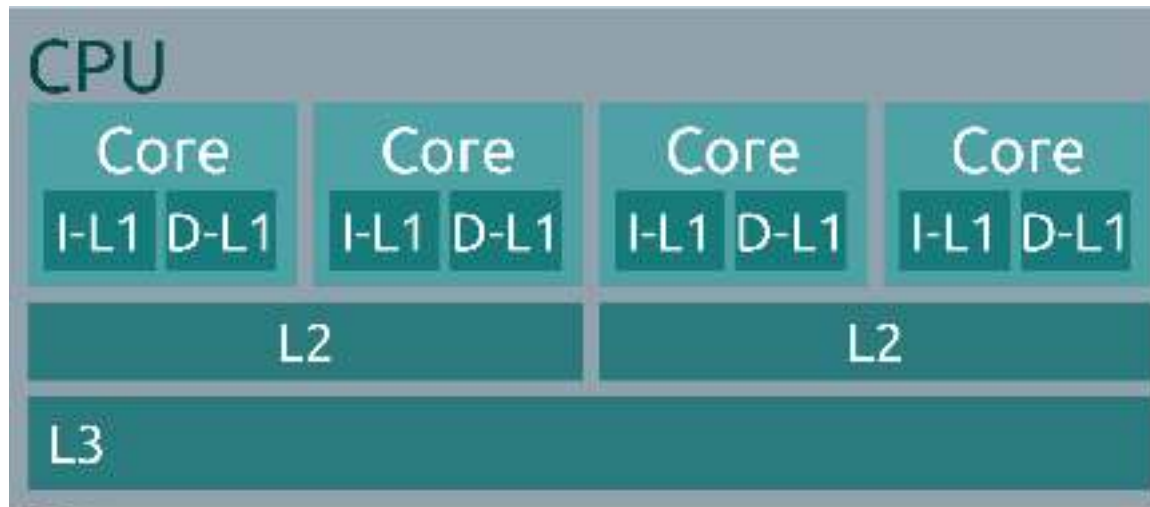# Why memory management matters
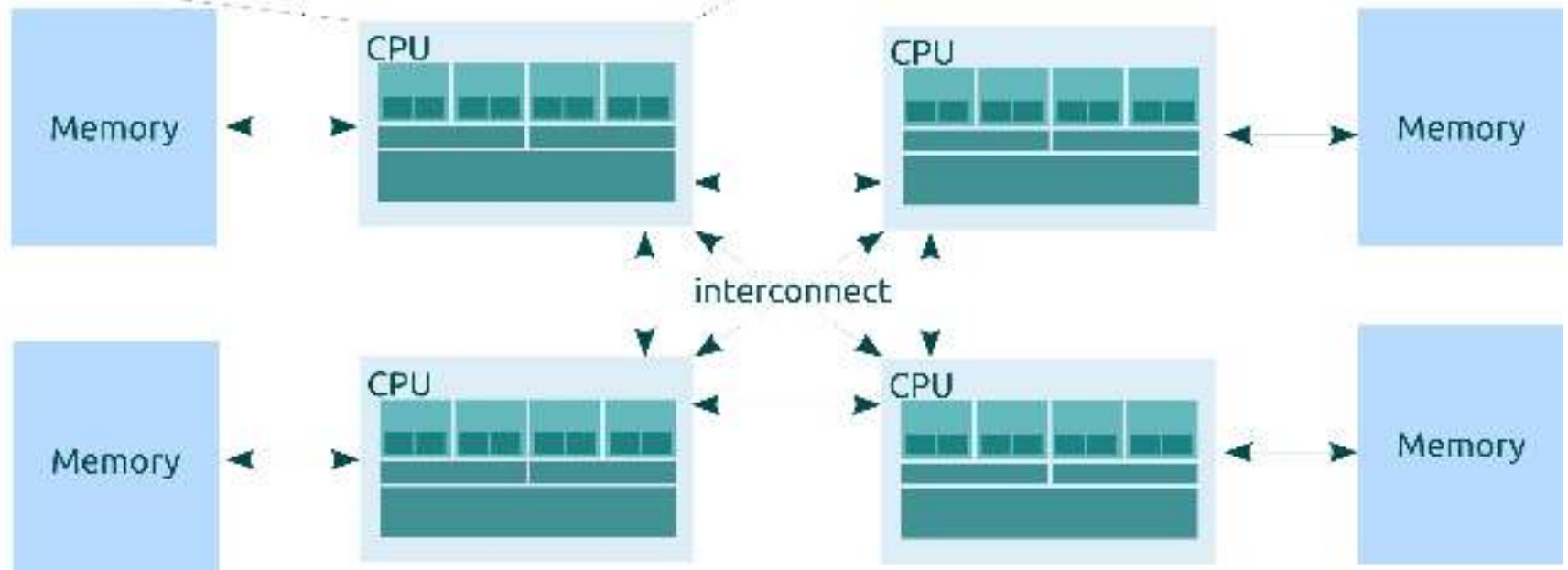

[Computer Architecture: A Quantitative Approach, 4th ed (John L. Hennessy, David A. Patterson)]

Introduction

- Typical, simplified, CPU and system layout
  - Non Uniform Memory Access

What's the ideal memory?

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

- Speed of SRAM, cost and capacity of disk

- The ideal situation can be approximated with a hierarchy of different memory types

Hierarchy levels

Processor

Data is transferred
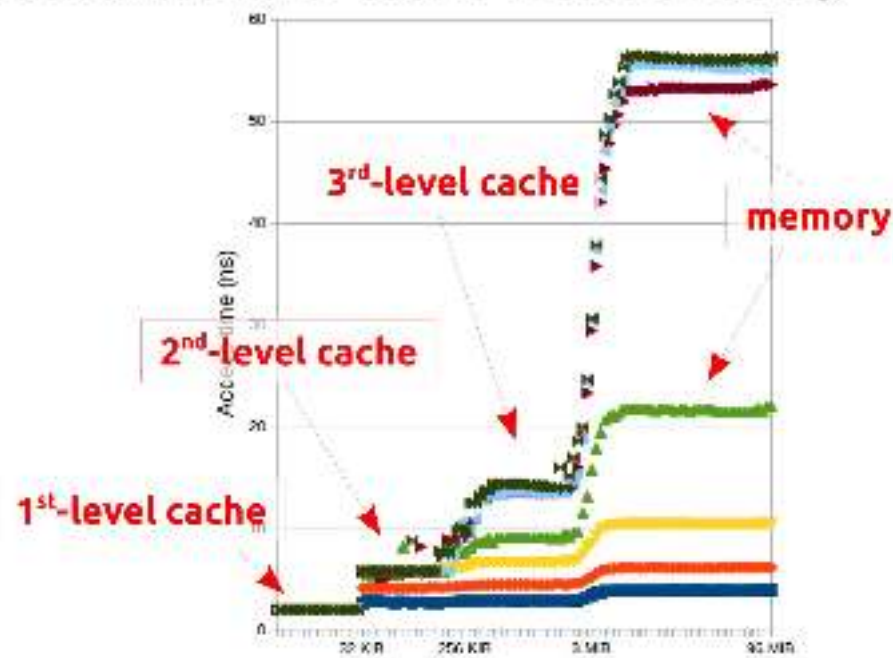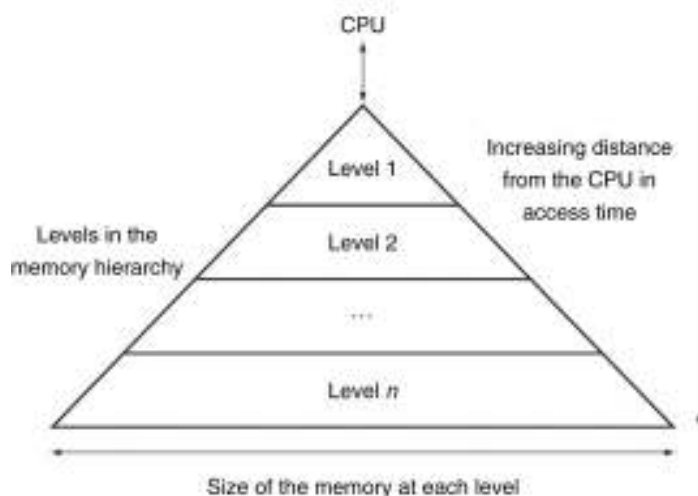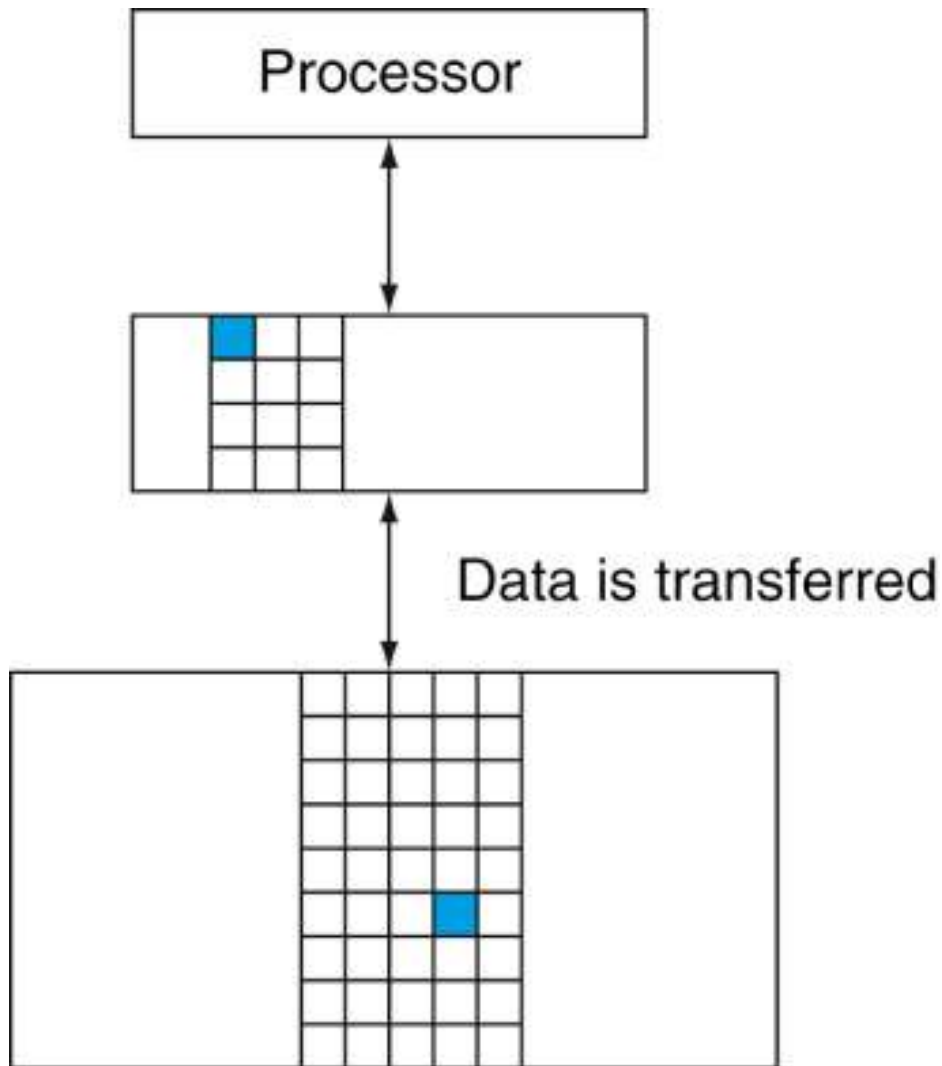
- The data is **present** in the highest level
  - *hit*
    hit rate = hits / accesses
- The data is **not present** in the highest level
  - *miss*: data is looked for in the lower level
  - miss penalty: the cost of getting the data
  - likely causes stalls in the execution
- Data is moved in blocks (cache lines)

Locality principle

```
int strlen(char const* str)
{
  int len = 0;
  while (*str++) ++len;
  return len;
}
```

- Data

  - Multiple accesses to variable `len`

  - Scanning of array `str`

- Instructions

  - Repetition of the instructions corresponding to the expressions *str++ e ++len

 - Execution of consecutive instructions

Locality principle

- In a limited time interval a program accesses only a small part of its whole address space

- Temporal locality

  - Memory locations recently accessed tend to be accessed again in the near future

    - e.g. instructions and counters in a loop

- Spatial locality

  - Memory locations near those recently accessed tend to be accessed in the near future

    - e.g. sequential access to instructions in a program or to data in an array

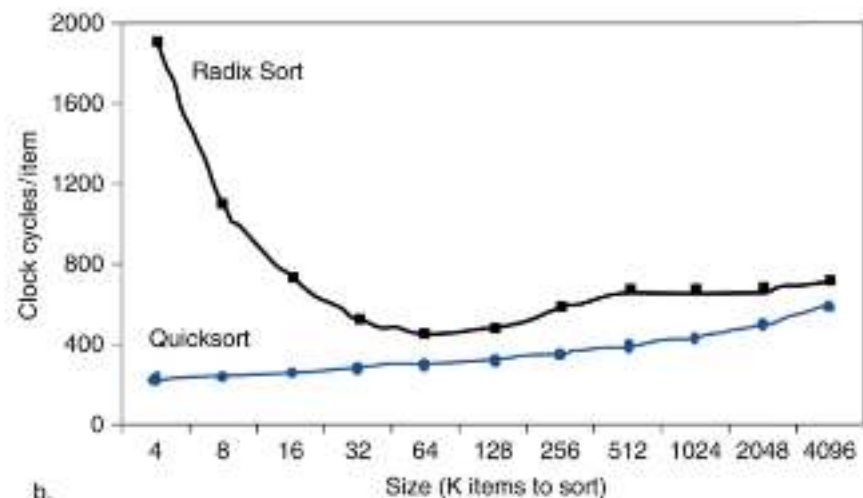- Hardware components like caches and pipelines are justified by the locality principle

Cache effect

- The efficiency of a program does not depend only on the computational complexity of an algorithm...
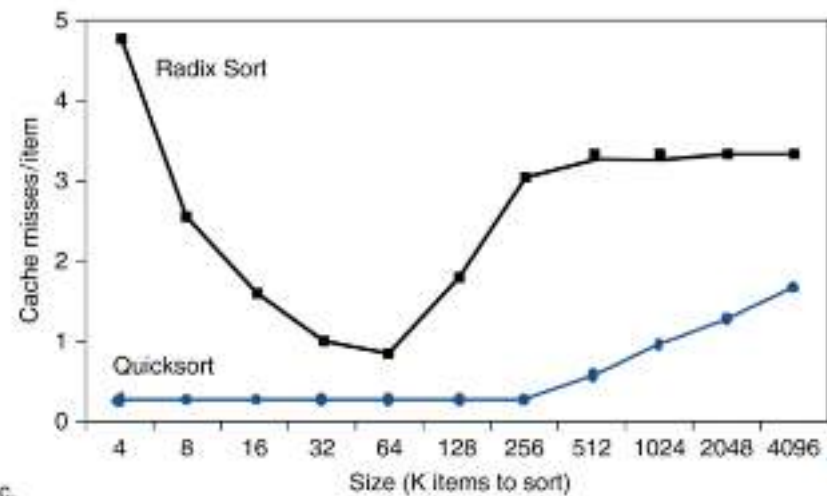
Be friendly to the cache

a.

b.

c.

7
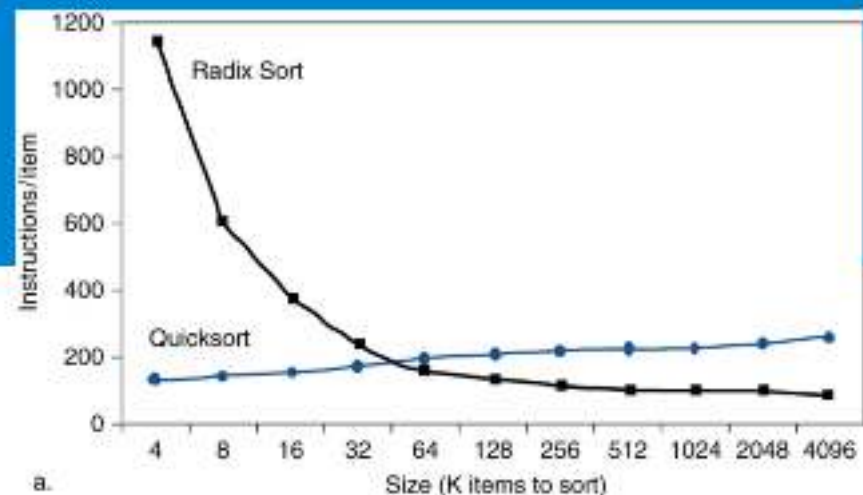
Size of a type

- Determined statically (i.e. at compile time)
- Queried with the `sizeof` operator
  - returns multiples of `sizeof(char)`, which by definition is 1
  - typically a `char` is 1 byte, 8 bits

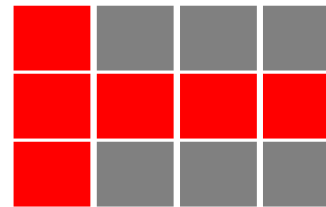- For primitive types
  - on my laptop

Type	sizeof
bool	1
char	1
short	2
int	4
long	8
long long	8
float	4
double	8
long double	16
void*	8

Layout

- Consider

```
struct S
{
    char c1;
    int  n;
    char c2;
};

static_assert(sizeof(S) == 12);
```
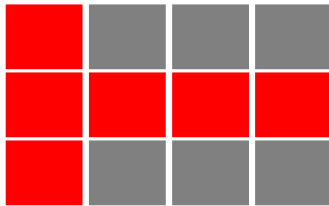

- The size is influenced by alignment constraints
    - the address of a variable of a certain type is typically a multiple of the size of that type
    - e.g. an `int` can reside only at an address multiple of 4

Does it matter?

- ## Yes, it does

```
struct P
{
  char c1;
  int  n;
  char c2;
};

static_assert(sizeof(P) == 12);
```

```
struct P
{
  int  n;
  char c1;
  char c2;
};

static_assert(sizeof(P) == 8);
```

```
vector<S> v = …;
sort(v.begin(), v.end(), [](S const&, S const&) {…});
```

(10'000'000 elements, on my laptop)

```
$ g++ -O3 sort_packed.cpp && ./a.out
1.83519 s
$ g++ -O3 sort_packed.cpp -DPACKED && ./a.out
1.40651 s
```

Cold data

- ## Consider

```
struct S
{
  int    n;
  float  f;
  double d;
};

static_assert(sizeof(S) == 16);
```

optimal layout

```
vector<S> v = …;
sort(v.begin(), v.end(), [](S const& l, S const& r) { return l.n < r.n; });
```

the order depends only on S::n

cache line (64 bytes)

- ## Data is brought into the cache, but it's not used
  - – NB the "usefulness" depends on the specific operation

Does it matter?

- Yes, it does

```
struct S
{
  int  n;
  char ext[EXTSIZE]
};

vector<S> v = …;
sort(v.begin(), v.end(), [](S const& l, S const& r) { return l.n < r.n; });
```

```
$ g++ -O3 sort_cold.cpp -DEXTSIZE=0 && ./a.out
1.29311 s
$ g++ -O3 sort_cold.cpp -DEXTSIZE=4 && ./a.out
1.29337 s
$ g++ -O3 sort_cold.cpp -DEXTSIZE=16 && ./a.out
1.50923 s
$ g++ -O3 sort_cold.cpp -DEXTSIZE=64 && ./a.out
2.40513 s
$ g++ -O3 sort_cold.cpp -DEXTSIZE=128 && ./a.out
5.52278 s
```

(10'000'000 elements, on my laptop)

Alternative design techniques

- Externalize cold data from the data structure

```
using Ext = char[EXTSIZE];
struct Particle {
  Vec position_;
  Ext ext_;
  void translate(Vec const& t) {
    position_ += t;
  }
};
```

```
using Ext = char[EXtSIZE];
struct ParticleExt { Ext ext; };
struct Particle {
  Vec position_;
  unique_ptr<ParticleExt> ext_;
  void translate(Vec const& t) {
    position_ += t;
  }
};
```

```
using Particles = vector<Particle>;
void translate(Particles& ps, Vec const& t) {
  for_each(ps.begin(), ps.end(),
    [=](Particle& p) { p.translate(t); }
  );
}
```

no impact on client code

```
$ for i in 0 8 16 64 128; do
> g++ -O3 aos.cpp -DEXTSIZE=$i && ./a.out
> done
2.23362 s
3.01734 s
3.86343 s
8.82871 s
11.1572 s
```

```
$ for i in 0 8 16 64 128; do
> g++ -O3 aos_impr.cpp -DEXTSIZE=$i && ./a.out
> done
3.03092 s
3.00671 s
3.02062 s
3.03086 s
3.03834 s
```
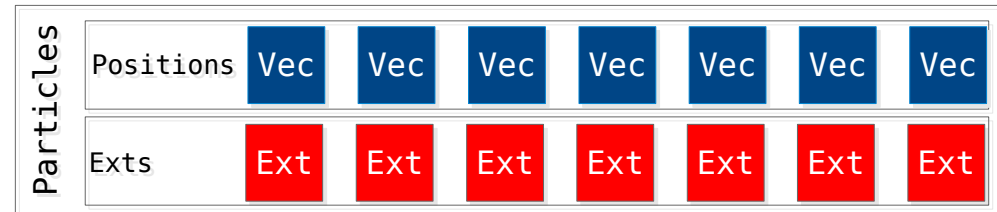
Alternative design techniques

- Structure of Arrays instead of Array of Structures

```
struct Particle {
  Vec position;
  Ext ext;
  void translate(Vec const& t) {
    position += t;
  }
};


using Particles = vector<Particle>;
```
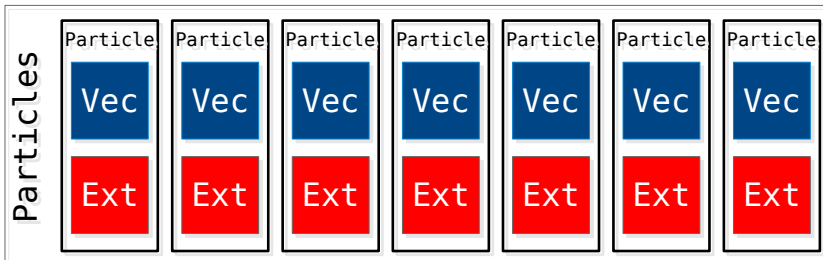
```
struct Particles {
  vector<Vec> positions;
  vector<Ext> exts;
};
void translate(Vec& position, Vec const& t) {
  position += t;
}
```


- The technique can be brought to the extreme, down to the primitive types

Alternative design techniques

- Structure of Arrays

```
struct Particle {
  Vec position;
  Ext ext;
  void translate(Vec const& t) {
    position += t;
  }
};

Particles v;
v[i].position;
```

```
struct Particles {
  vector<Vec> positions;
  vector<Ext> exts;
};
void translate(Vec& position, Vec const& t) {
  position += t;
}

Particles v;
v.positions[i];
```
some impact on client code

```
void translate(Particles& ps, Vec const& t) {
  for_each(ps.begin(), ps.end(),
    [=](Particle& part) { part.translate(t); }
  );
}
```

```
void translate(Particles& ps, Vec const& t) {
  auto& positions = ps.positions;
  for_each(positions.begin(), positions.end(),
    [=](Vec& pos) { translate(pos, t); }
  );
}
```

```
$ for i in 0 8 16 64 128; do
> g++ -O3 aos.cpp -DEXTSIZE=$i && ./a.out
> done
2.23362 s
3.01734 s
3.86343 s
8.82871 s
11.1572 s
```

```
$ for i in 0 8 16 64 128; do
> g++ -O3 soa.cpp -DEXTSIZE=$i && ./a.out
> done
2.25027 s
2.24427 s
2.23608 s
2.24376 s
2.24427 s
```

Hands-on

- Inspect, build and run
  - `sort_packed.cpp`
  - `sort_cold.cpp`
  - `aos.cpp`
  - `aos_impr.cpp`
  - `soa.cpp`

Dynamic memory allocation

- It's not always possible to know at compile time which type of objects are needed or how many of them

  - run-time polymorphism

```
struct Shape { … };
struct Rectangle : Shape { … };
struct Circle : Shape { … };

Shape* s = nullptr;
char c;
cin >> c;
switch (c) {
 case 'r': s = new Rectangle; break;
 case 'c': s = new Circle; break;
}
```
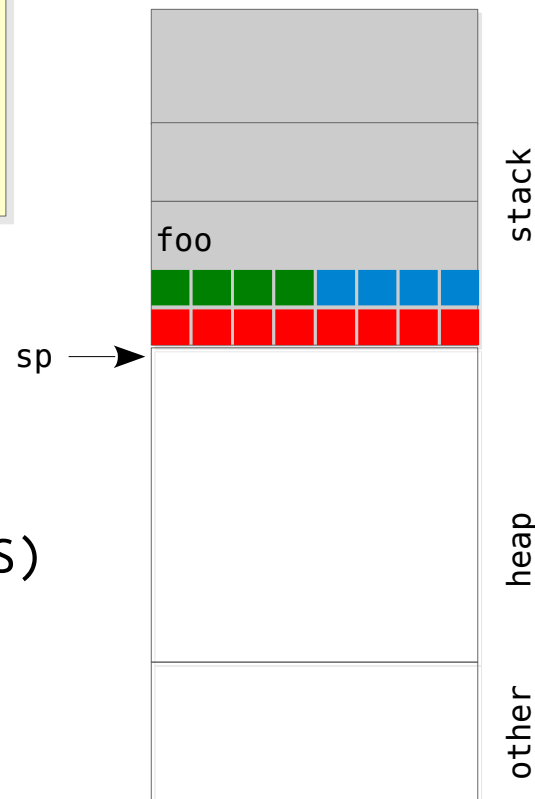
  - dynamic collections of objects

```
int n;
cin << n;
vector<Particle> v;
for ( ; n; --n) {
  v.push_back(Particle{ … });
}
```
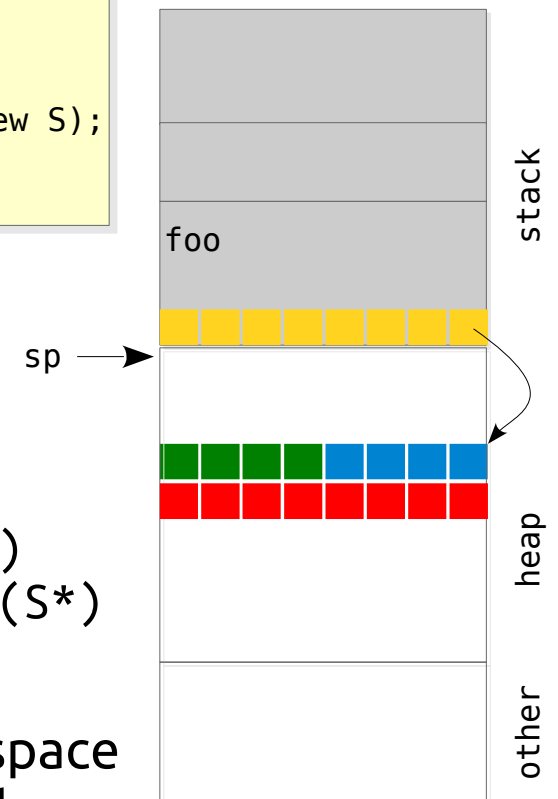
Stack vs heap: space

- ## Stack allocation

```
struct S {
  int    n;
  float  f;
  double d;
};

auto foo() {
  S s;
  …
}
```

- ## Occupancy

  – `sizeof(S)`

- ## Heap allocation

```
struct S {
  int    n;
  float  f;
  double d;
};

auto foo() {
  unique_ptr<S> u(new S);
  …
}
```

- ## Occupancy

  – `sizeof(S)`
    `+ sizeof(S*)`

  – plus new internal space overhead

Stack vs heap: time

- ## Stack

```
void stack()
{
  int m{123};
}
```

```
stack():
    subq %4, %rsp
    movl $123, (%rsp)
    addq $4, %rsp
    ret
```
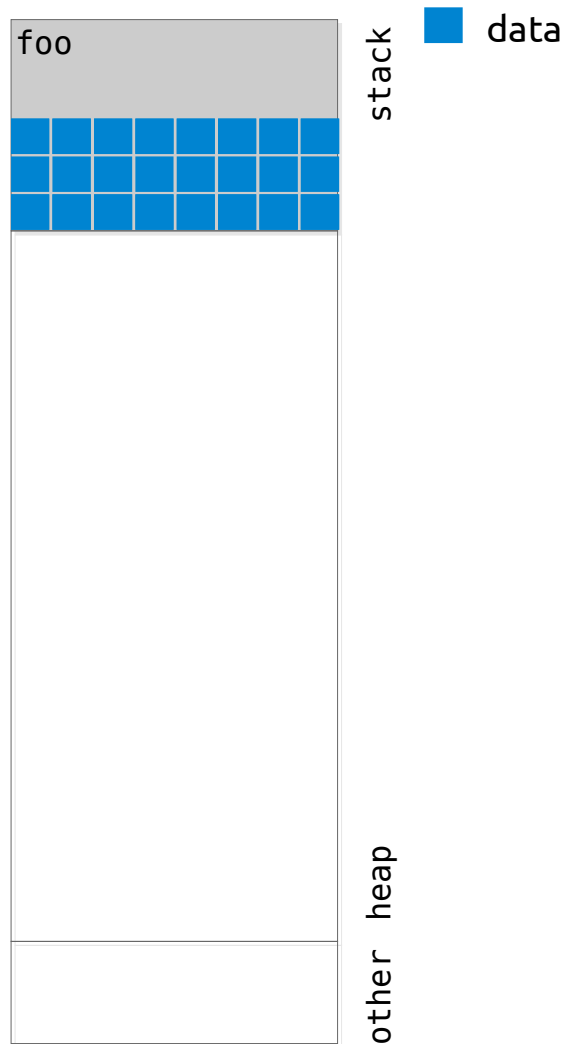
- ## Heap

```
void heap()
{
  int* m = new int{123};
  delete m;
}
```

```
heap():
    subq $8, %rsp
    movl $4, %edi
    call operator new(unsigned long)
    movl $123, (%rax)
    movl $4, %esi
    movq %rax, %rdi
    call operator delete(void*, unsigned long)
    addq $8, %rsp
    ret
```
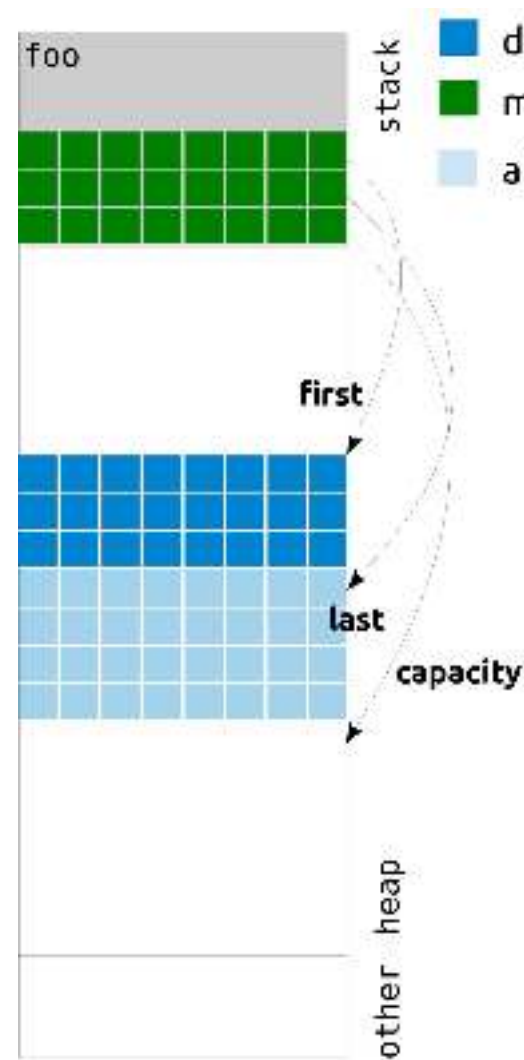
```
$ g++ -O3 heap.cpp && ./a.out
1000000 iterations: 0.0494411 s
```

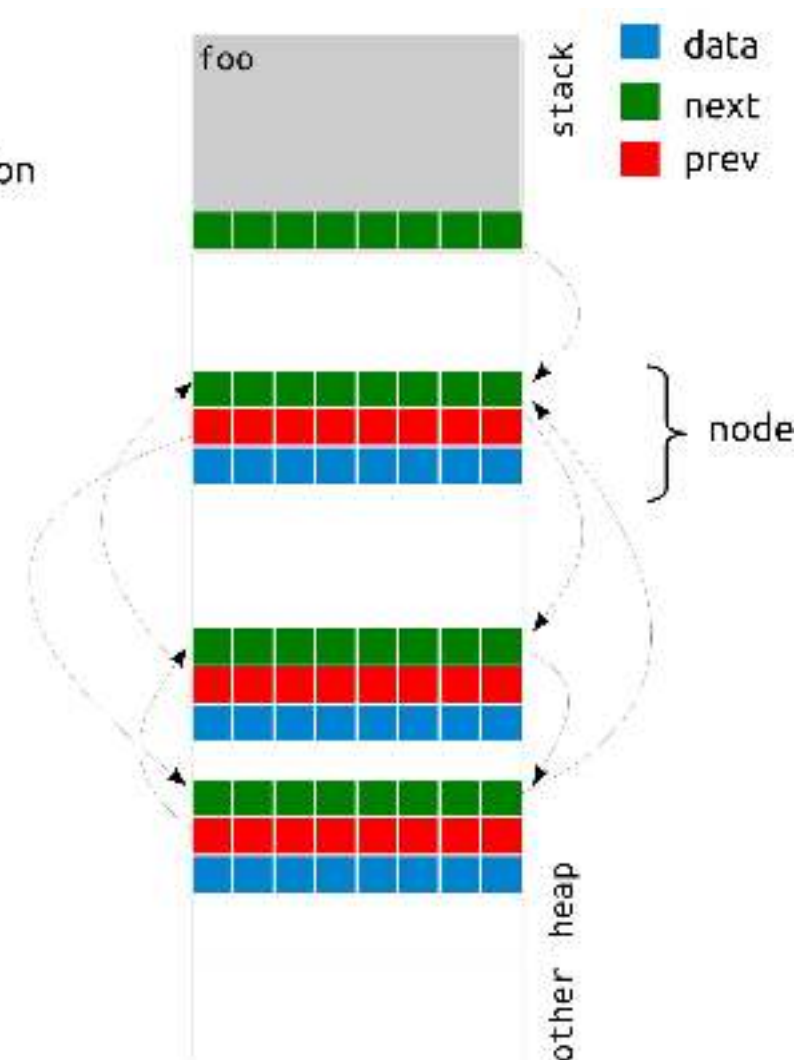i.e. 50 ns just to allocate/deallocate an `int`

# Memory overhead of std containers
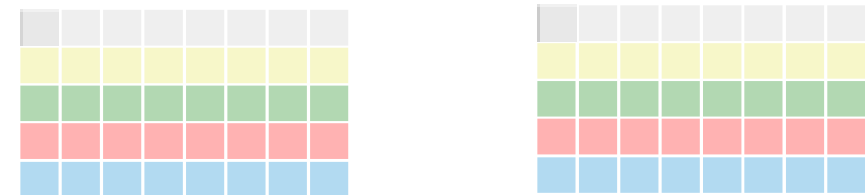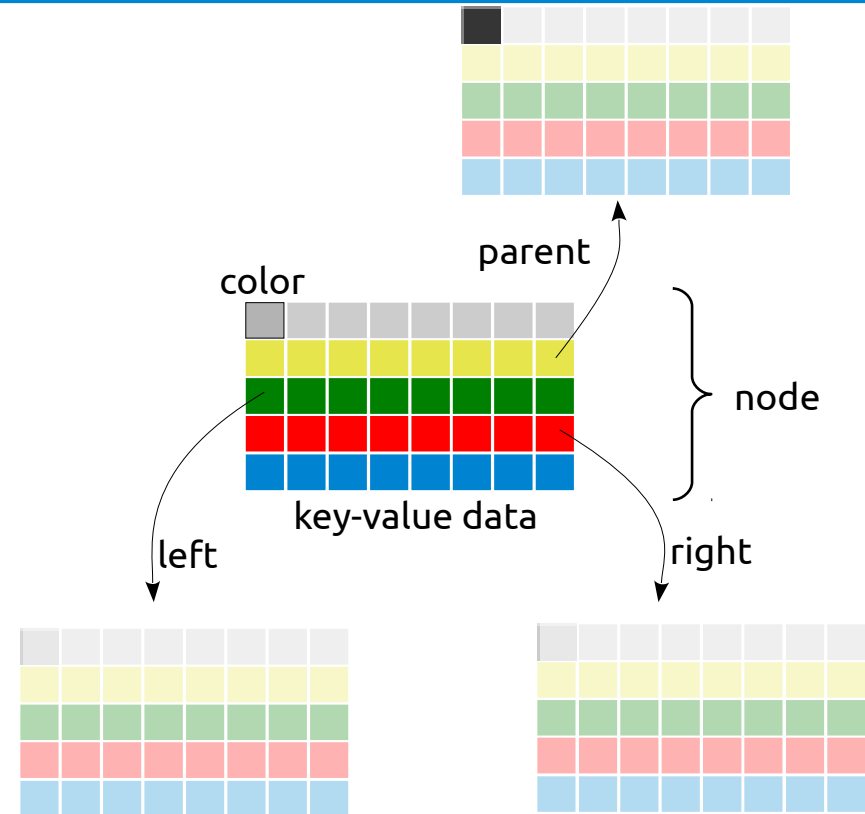

std::array

std::vector

std::list

std::map

- Sorted associative container that contains key-value pairs with unique keys

- Search, removal, and insertion have logarithmic complexity

- Implemented as a red-black tree

```
std::map<int, double> m = {
  {123, 2.}, {456, 1.}, {789, 3.}
};

m[123];
auto n = m;              // makes a full copy
auto o = std::move(m); // efficient move
m.find(123);            // as member
```


color   parent   node

key-value data

left   right
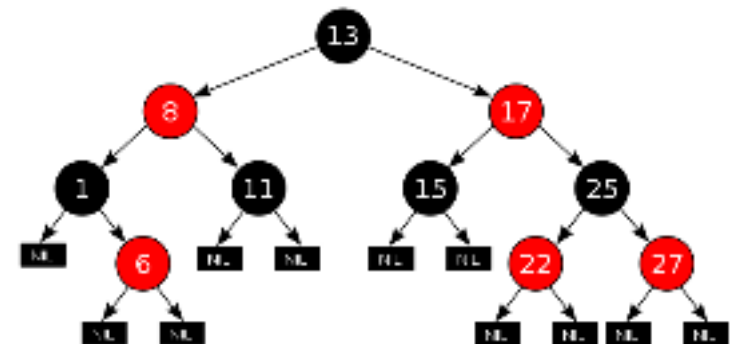
Hands-on

- Inspect, build and run `containers.cpp`, also through `perf`
- Extend it to manage an `std::list`