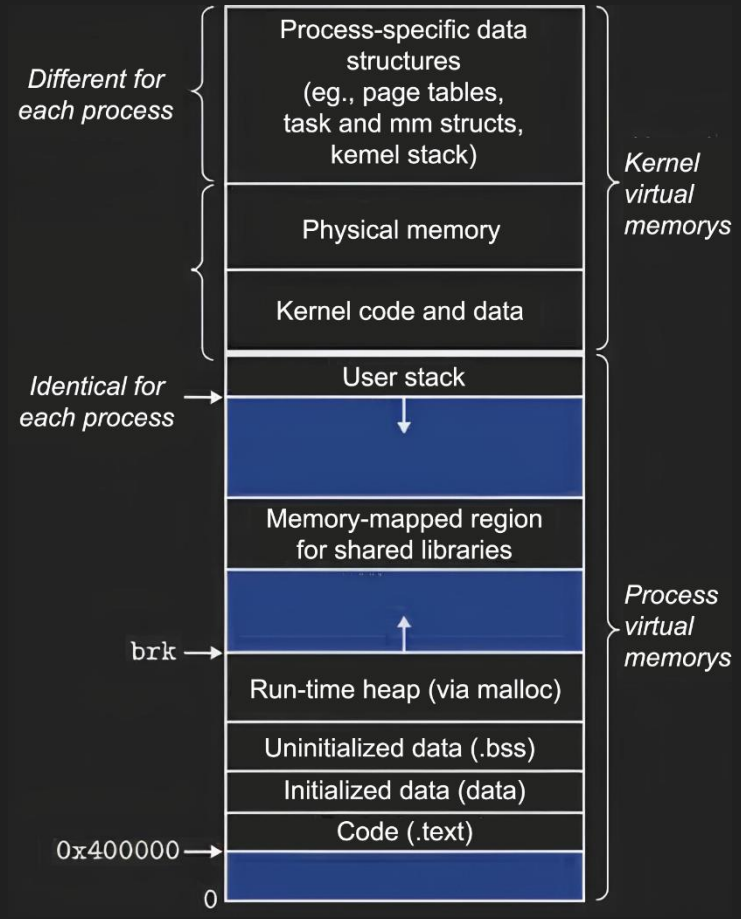
SPECTRAL:
TECHNOLOGIES

WEEK 1

Memory layout,
internal object representation

SPECTRAL:
TECHNOLOGIES

Virtual memory layout



Stack

- Memory allocation is "free" and "continuous"

```
int data[10]; -> subq    $48,  %rsp
```

- Usually limited in size, but can be increased
- Allocation (stack growth) can lead to a page fault, so there is a potential cost to it, but its 1-time only, since stack doesn't shrink

Heap

- An allocation here is time-consuming in general
- The performance can degrade as the application continues to run
- There are a lot of different allocators, each is designed around certain workload it needs to handle, so there is no silver bullet
- Allocations can easily take most of the running time if you don't pay attention
- Most allocators have really bad tail-latency values.

General performance tips

- time vs memory consumption tradeoff
- In general you want to preallocate as much memory as possible (in hft the entire memory pool is static or allocated on heap at the start)
- If you have to increase the default stack size, it usually means there is something wrong with your code.
- Consider what exactly happens when you use some std-container. Do not replace blindly `std::vector` with `std::array` just because its "faster" (its not).
- Don't mix memory management logic with the rest of your code. You can still have nice std-like interfaces and 0 allocations (home tasks).

SPECTRAL:
TECHNOLOGIES

Memory layout of simple types

Assume little-endian byte order when not specified in this course

```
char a = 1;
int b = 17239;
int c = -17239;
const char *d = "text";
char e[10];
std::cin >> e; // "hello"
double f = 2.0;
int64_t g = 3;
```

```
(gdb) x/60b &g
0x7fffffffe108: g<0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00>
0x7fffffffe110: f<0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x40>
0x7fffffffe118:   0x10    0x12    0x40    0x00    0x00    0x00  e<0x68    0x65
0x7fffffffe120:   0x6c    0x6c    0x6f    0x00    0x00    0x00    0x00    0x00>
0x7fffffffe128: d<0x10    0x20    0x40    0x00    0x00    0x00    0x00    0x00>
0x7fffffffe130:   0x30    0xe2    0xff    0xff  c<0xa9    0xbc    0xff    0xff>
0x7fffffffe138: b<0x57    0x43    0x00    0x00>   0x00    0x00    0x00  a<0x01>
0x7fffffffe140:   0x00    0x00    0x00    0x00
```

Memory layout and padding

```
struct Item {
    int64_t a;
    int b;
    char c;
};
...
std::vector<Item> data{Item{1, 2, 'a'}, Item{-1, 64, 'b'}};
std::cout << sizeof(Item) << " " << alignof(Item) << "\n"; // 16 8
```

```
(gdb) x/8wx &data[0]
0x416eb0:       0x00000001      0x00000000      0x00000002      0x00000061
0x416ec0:       0xffffffff      0xffffffff      0x00000040      0x00000062
```

Disable padding

```
#pragma pack(push, 1)
struct Item {
    int64_t a;
    int b;
    char c;
};
#pragma pack(pop) // size = 13, alignof = 1
```

Not recommended, since it slows down any memory access

SPECTRAL:
TECHNOLOGIES

Memory layout of complex classes

```
struct Item {
    std::vector<int> a;
    std::string b;
};

Item a{std::vector<int>{1, 2, 3}, "hello"};
std::cout << sizeof(a.a) << " " << sizeof(a.b) << "\n"; // 24 32
```

```
(gdb) x/60bx &a
0x7fffffffe108: 0xb0    0x6e    0x41    0x00    0x00    0x00    0x00    0x00
0x7fffffffe110: 0xbc    0x6e    0x41    0x00    0x00    0x00    0x00    0x00
0x7fffffffe118: 0xbc    0x6e    0x41    0x00    0x00    0x00    0x00    0x00
0x7fffffffe120: 0x30    0xe1    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe128: 0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe130: 0x68    0x65    0x6c    0x6c    0x6f    0x00    0x00    0x00
0x7fffffffe138: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe140: 0x00    0x00    0x00    0x00
(gdb) p &a.a[0]
$1 = (int *) 0x416eb0
```

Functions and classes (without inheritance)

- Functions (i.e the instructions) are located in text section of a file and so those are stored separately from the data.
- Method is basically a function with `this` as it's first argument.
- So the layout of any class is defined by the layout of it's members.
- `public` , `private` sections do not matter.
- templates is just an advance codegen mechanism, doesn't affect the layout

Inheritance

```
struct A {
    int a;
    explicit A(int _a): a(_a) {}
    virtual int f() { return 5; }
};
struct B : public A {
    int b;
    virtual int f() { return 7; }
    B(int a, int b): A(a), b(b) {}
};

A a{1};
B b{2, 3};
```

```
(gdb) x/40bx &b
0x7fffffffe120: 0x48    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe128: 0x02    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0x7fffffffe130: 0x18    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe138: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe140: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Inheritance

```
B b{2, 3};
A *a = &b;
int c = a->f();
```

```
(gdb) x/24bx &a
0x7fffffffe128: 0x30    0xe1    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe130: 0x18    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe138: 0x02    0x00    0x00    0x00    0x03    0x00    0x00    0x00
(gdb) x/24bx 0x402018
0x402018 <_ZTV1B+16>:   0x60    0x12    0x40    0x00    0x00    0x00    0x00    0x00
0x402020 <_ZTS1B>:      0x31    0x42    0x00    0x31    0x41    0x00    0x00    0x00
0x402028 <_ZTI1A>:      0x50    0x3d    0x40    0x00    0x00    0x00    0x00    0x00
(gdb) x/24bx 0x401260
0x401260 <B::f()>:      0x55    0x48    0x89    0xe5    0x48    0x89    0x7d    0xf8
0x401268 <B::f()+8>:    0xb8    0x07    0x00    0x00    0x00    0x5d    0xc3    0x90
0x401270 <A::f()>:      0x55    0x48    0x89    0xe5    0x48    0x89    0x7d    0xf8
```

Inheritance

```
(gdb) x/15i 0x401260
   0x401260 <B::f()>:    push   %rbp
   0x401261 <B::f()+1>:  mov    %rsp,%rbp
   0x401264 <B::f()+4>:  mov    %rdi,-0x8(%rbp)
   0x401268 <B::f()+8>:  mov    $0x7,%eax
   0x40126d <B::f()+13>:         pop    %rbp
   0x40126e <B::f()+14>:         retq
   0x40126f:     nop
   0x401270 <A::f()>:    push   %rbp
   0x401271 <A::f()+1>:  mov    %rsp,%rbp
   0x401274 <A::f()+4>:  mov    %rdi,-0x8(%rbp)
   0x401278 <A::f()+8>:  mov    $0x5,%eax
   0x40127d <A::f()+13>:         pop    %rbp
   0x40127e <A::f()+14>:         retq
   0x40127f:     nop
   0x401280 <__libc_csu_init>:  endbr64
```

Inheritance

- In this case vptr is located at the start of an object
- May differ between compilers
- Compiler can optimize out vptr-call
- The pointer cast between base and derived class is free
- Its not true in case of multiple and/or virtual inheritance

Multiple inheritance

```
struct A {
    int a;
    virtual int f() { return 5; }
};

struct B {
    int b;
    virtual int g() { return 7; }
};

struct C: public A, public B {
    int c;
    C(int a, int b, int c): A(a), B(b), c(c) {}
};

C c{1, 2, 3};
A *a = &c;
B *b = &c;
```

Multiple inheritance

```
C c{1, 2, 3};
A *a = &c;
B *b = &c;
```

```
(gdb) x/60bx &b
0x7fffffffe108: 0x28    0xe1    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe110: 0x18    0xe1    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe118: 0x18    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe120: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe128: 0x30    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe130: 0x02    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0x7fffffffe138: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe140: 0x00    0x00    0x00    0x00
```

Bytes

- Types exist during compilation only, and any type in c++ is just a way to interpret bytes in memory.

```
struct Item {
    int a;
    int b;
    int64_t c;
    char payload[16];
};

char *data = ...; // assume it points somewhere we expect the same protocol
                  // a binary file for a example
Item *item = reinterpret_cast<Item*>(data); // or (Item*)(data);
```

- This is **UB** (but nobody cares)

Bytes

- May lead to problems, but its very unlikely
- Don't forget about an alignment and little/big-endian though.
- The right way to do this (`bit_cast` since c++20)

```
Item item;
memcpy(&item, data, sizeof(Item));
```

- The high 16 bits of pointers are not used on x86_64, so you can store some data there.
- The low bits of pointers can also be used due to the alignment of the underlying data.

Conclusion

- What we've omitted
  - virtual inheritance
  - pointers to class members (`class A` , what exactly is `&A::a`)
  - pointers to methods (what is `&A::method`)

- Additional materials
  - Inside the C++ Object Model By Stanley B. Lippman
  - https://en.wikipedia.org/wiki/Two's_complement
  - https://graphics.stanford.edu/~seander/bithacks.html