NTNU – Trondheim
Norwegian University of Science and Technology

(Simple) Objects

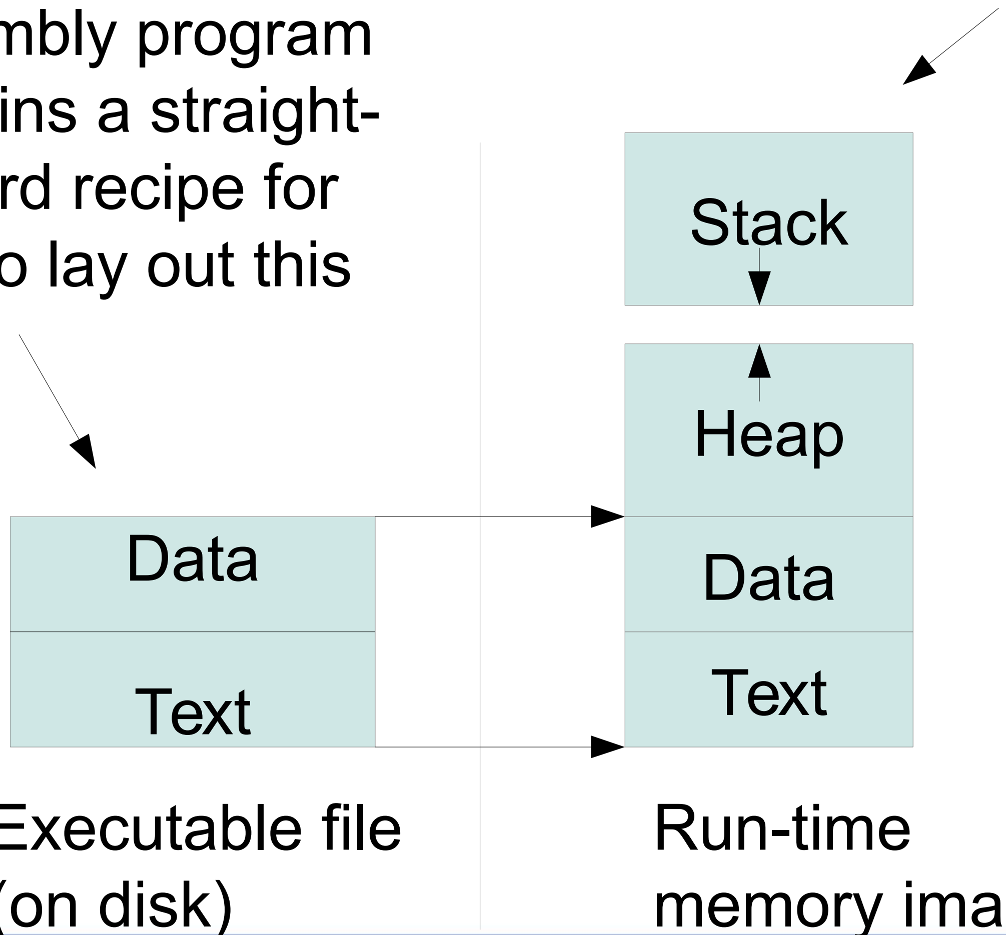
Where we were

- Last time, we looked at the details of function call mechanisms
- Object types require some extension to this, but we can cover the basics by taking a quick look at it
- That is today's topic

Process address space (again...)

OS loader expands file to image every time program is run

Assembly program contains a straight-forward recipe for how to lay out this file



Executable file (on disk)

Run-time memory image

Code generation for functions

- Functions become labels for addresses where the subsequent instructions accept the arguments

  (laid out as a stack frame matching the function's activation record)

Text segment
_factorial:
  (setup stackframe)
  (copy arg. 1)
  (compute)
  (remove stackframe)
  (return result)
_other_function:
  (all that stuff)
  (return)
_ one_more_function:
  (same story)
  (return)

We looked at the operations that go into these steps last time

Code generation for function calls

- Static function calls have unique names and type signatures, compiler can just push arguments in turn and insert call operation

<u>Text segment</u>
_factorial:
(setup stackframe)
(copy arg. 1)
(compute)
(remove stackframe)
(return result)

_main:
push 3
call _factorial

This location is mapped from a symbolic name into a target for the program counter:

1) Assembler substitutes name with symbolic adr.
2) Linker resolves adr. relative to text segment start
3) Loader maps it to actual address, visible to OS

The need for run-time dispatch

```
interface Point { int getx(); int gety(); float norm(); }
class ColoredPoint implements Point {           ...
  float norm() { return sqrt(x*x+y*y); }  ...
}
class 3DPoint implements Point {       ...
  float norm() { return sqrt(x*x+y*y+z*z); }       ...
}

Point p;
if ( cond ) p = new ColoredPoint();
else p = new 3DPoint();
float n = p.norm();
```

Which of these to call...

...is only known at run time

NTNU – Trondheim
Norwegian University of
Science and Technology

Method calls need indirection

- Even if we generate methods for each variant, the destination of a call can't be resolved once and for all...

```
Text segment
_cpoint_norm:
  (setup stackframe)
  (compute)
  (return result)

_3dpoint_norm:
  (setup stackframe)
  (compute)
  (return result)

_main:
  this = point
  push this
  call (something)
```

Which adr. to put here?

NTNU – Trondheim
Norwegian University of
Science and Technology

Number the methods

- Inherited/overridden methods can share the same index

```
Class A {
  void f();          0
}
Class B extends A {
   void f();         0
   void g();         1
   void h();         2
}
Class C extends B {
   void e();         3
}
```

Each class gets a table

- Keeping the indices consistent per method,

A	
f	&a_f

B	
f	&b_f
g	&b_g
h	&b_h

C	
f	&b_f
g	&b_g
h	&b_h
e	&c_e

a call to “f” for either of these classes is a call to “method #0”

NTNU – Trondheim
Norwegian University of Science and Technology

Static lookup by cast

- With an explicit cast, the table to use can be determined statically

A	
f	&a_f

B	
f	&b_f
g	&b_g
h	&b_h

C	
f	&b_f
g	&b_g
h	&b_h
e	&c_e

B my_b = new B();

((A) my_b).f() ← resolves to "call method 0 in table A", where we find ptr. to A-s implementation of f()

Dynamic lookup by instance

- With an explicit cast, the table to use can be determined statically

A	
f	&a_f

B	
f	&b_f
g	&b_g
h	&b_h

C	
f	&b_f
g	&b_g
h	&b_h
e	&c_e

B my_b = new B();

my_b.f() ← Resolves to "call method 0 in table B", where we find ptr. to B-s overridden implementation of f()

NTNU – Trondheim
Norwegian University of Science and Technology

Dynamic table identification

- In order to resolve which table to use based on an object instance, the instance must be constructed with a pointer to the right table

A_DV	
f	&a_f

B_DV		
f	&b_f	(0)
g	&b_g	
h	&b_h	

C_DV	
f	&b_f
g	&b_g
h	&b_h
e	&c_e

B my_b = new B(0);

creates an instance

my_b	
dv	&B_DV

my_b.f()

creates an indir. call

my_b	
dv	&B_DV

(data segment)

B_DV		
f	&b_f	(0)
g	&b_g	
h	&b_h	

(text segment)

b_f:
(do_stuff)

This (mildly) complicates the call mechanism

- Generated function calls go

  push param1

  push param2..

  call function

- Generated method calls go

  dv = dv_offset(this) ← 'this' is an object instance, dv is table's offset

  adr = n(dv) ← where 'n' is the method index, dv the table

  push param1

  push param2…

  push this ← implicit argument, as we discussed before

  call adr

Via this indirection, the function called will be found via the dv table an instance is constructed with

Why ‘dv’?

- This mechanism is called a *Dispatch Vector*

    ...or a *dispatch table*…

    ...or a *selector table*….

    ...but *vector* is as good a name as any.

- All DV-s can be statically generated at compile time

Data segment
```
A_DV:
 &a_f    /* 0 */
B_DV:
 &b_f    /* 0 */
 &b_g    /* 1 */
 &b_h    /* 2 */
C_DV:
 &b_f    /* 0 */
 &b_g    /* 1 */
 &b_h    /* 2 */
 &c_e    /* 3 */
```

Text segment
```
a_f:
  /* code for A.f */

b_f:
 /* code for B.f */
b_g:
 /* code for B.g */
b_h:
 /* code for B.h */

c_e:
 /*code for C.e */
```

(offset in table is a constant multiple of method index: all pointers have the same size...)

It allows inheritance

- C can get most of its methods from B
  - Syntax says it's a subclass
  - Compiler embeds that when generating the dispatch vector

<u>Data segment</u>

```
A_DV:
 &a_f     /* 0 */
B_DV:
 &b_f     /* 0 */
 &b_g     /* 1 */
 &b_h     /* 2 */
C_DV:
 &b_f     /* 0 */
 &b_g     /* 1 */
 &b_h     /* 2 */
 &c_e     /* 3 */
```

<u>Text segment</u>

```
a_f:
  /* code for A.f */

b_f:
 /* code for B.f */
b_g:
 /* code for B.g */
b_h:
 /* code for B.h */

c_e:
 /*code for C.e */
```

It allows overriding

- B provides a different implementation of f() than A does

```
Data segment
A_DV:
  &a_f     /* 0 */
B_DV:
  &b_f     /* 0 */
  &b_g     /* 1 */
  &b_h     /* 2 */
C_DV:
  &b_f     /* 0 */
  &b_g     /* 1 */
  &b_h     /* 2 */
  &c_e     /* 3 */
```

```
Text segment
a_f:
  /* code for A.f */

b_f:
 /* code for B.f */
b_g:
 /* code for B.g */
b_h:
 /* code for B.h */

c_e:
 /*code for C.e */
```

NTNU – Trondheim
Norwegian University of Science and Technology

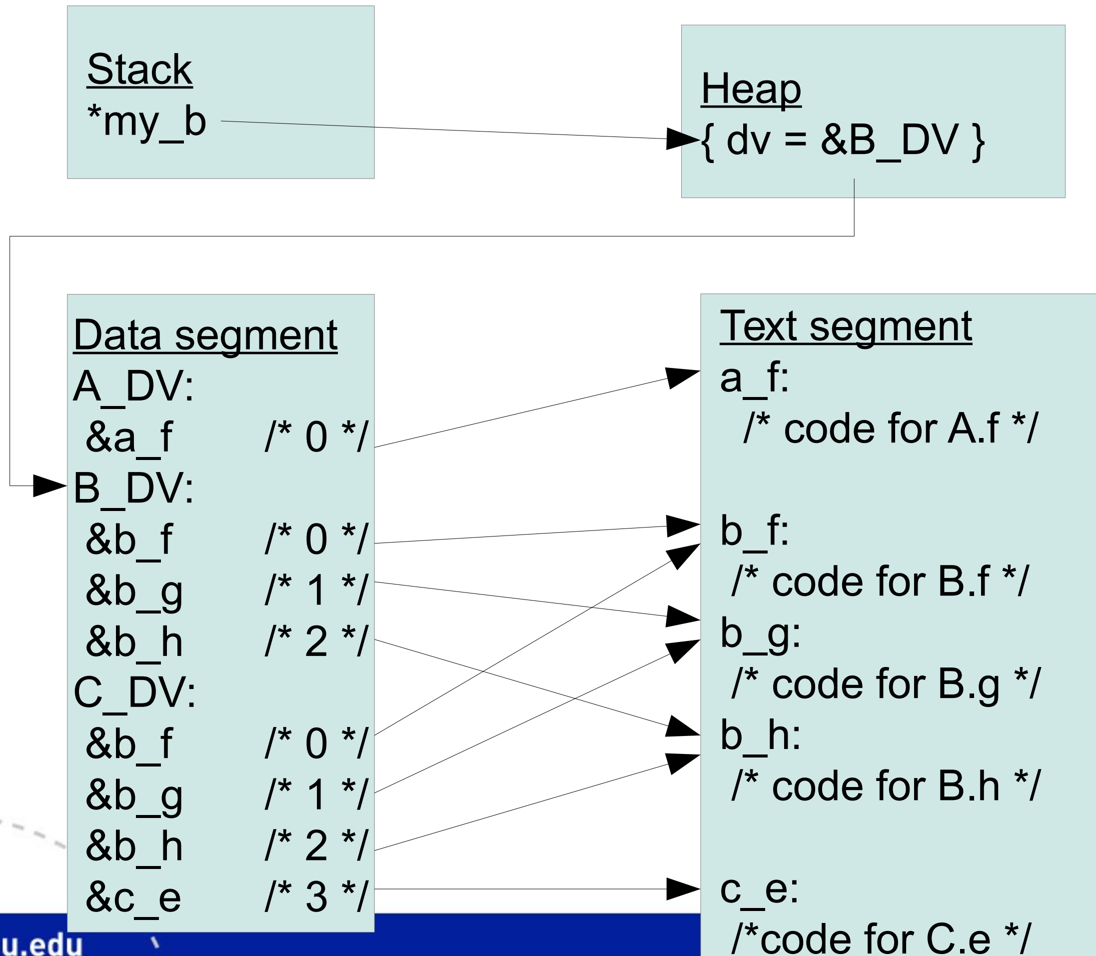
Interfaces

- This creates a natural interpretation of *interfaces* (which are classes without an implementation)
- They amount to constraints on the dispatch vector layout for classes that implement them
- They can be disposed of after compilation

- *Abstract classes* contain a dispatch vector layout and *some* specific implementations to point it at

Objects can be put on heap

- B my_b = new B();



Objects can be put on stack

- B my_b = B();

(The dv pointer is a field of constant size in either case)

```
Stack
my_b = {
 dv = &B_DV
}
```

```
Data segment
A_DV:
 &a_f      /* 0 */
B_DV:
 &b_f      /* 0 */
 &b_g      /* 1 */
 &b_h      /* 2 */
C_DV:
 &b_f      /* 0 */
 &b_g      /* 1 */
 &b_h      /* 2 */
 &c_e      /* 3 */
```

```
Text segment
a_f:
  /* code for A.f */

b_f:
 /* code for B.f */
b_g:
 /* code for B.g */
b_h:
 /* code for B.h */

c_e:
 /*code for C.e */
```

Footnote on memory access

- Fields that are not multiples of register size can be laid out densely, or with padding
  - For e.g. a CPU with 4-byte words,
    struct { char a; int16_t b; char c }
    can be laid out as

a	b_1	b_2	c

or alternatively,

a	0	0	0
b_1	b_2	0	0
c	0	0	0

NTNU – Trondheim
Norwegian University of Science and Technology

Byte-aligned access is not always supported

- Some processors demand register-aligned adresses, so

a	b_1	b_2	c

will force the compiler to generate a fetch of the whole thing, and code to mask out and shift the elements you want

i.e. for access to b

a	b_1	b_2	c	Load
0	b_1	b_2	0	Delete
b_1	b_2	0	0	Shift

(The code to do this can easily take more space than you save by packing data)

Byte-aligned access is slow

- Hardware-support for unaligned access typically does the load-mask-delete thing anyway
- You don't have to write it, but it takes time (~10x)

- I'm just mentioning this because the memory-indirection scheme might indicate that dynamic dispatch adds great run-time overhead
- Memory access is expensive, but not always in a way that's easy to expect...

Next up

- An introduction to 64-bit x86 assembly programming