Compilation

0368-3133

Lecture 13:

Course summary: Advanced Topics

Noam Rinetzky

Compiling OO Programs

Features of OO languages

- **Inheritance**
  - Subclass gets (inherits) properties of superclass
- **Method overriding**
  - Multiple methods with the same name with different signatures
- **Abstract** (aka **virtual**) **methods**
- **Polymorphism**
  - Multiple methods with the same name and different signatures but with different implementations
- **Dynamic dispatch**
  - Lookup methods by (runtime) type of target object

Compiling OO languages

- “Translation into C”
- Powerful runtime environment

- Adding “gluing” code

Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Memory management
- **Runtime type information**
  - **Method invocation**
  - **Type conversions**

Handling Single Inheritance

- Simple type extension

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m3() {…}
}
```

Adding fields

Fields aka Data members, instance variables

- Adds more information to the inherited class
  - "Prefixing" fields ensures consistency

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {…}
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

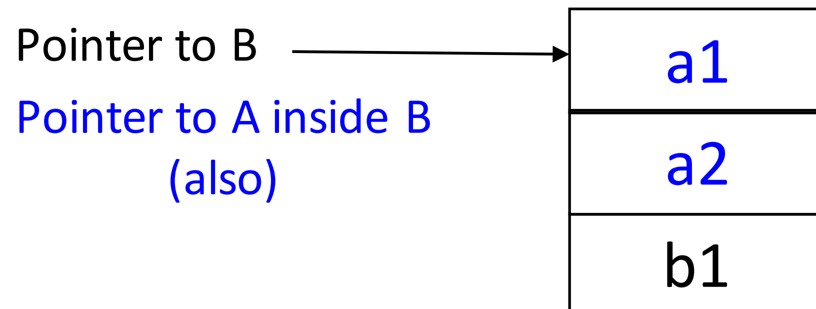
```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

Handling Polymorphism

- When a class B extends a class A
  - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Prefixing fields guarantees validity

```
class B *b = …;

class A *a = b ;
```

```
classA *a = convert_ptr_to_B_to_ptr_A(b) ;
```



Dynamic Binding

- An object ("pointer") o declared to be of class A can actually be ("refer") to a class B

- What does 'o.m()' mean?
  - Static binding
  - Dynamic binding
- Depends on the programming language rules

- How to implement dynamic binding?
  - The invoked function is not known at compile time
  - Need to operate on data of the B and A in consistent way

Virtual function table

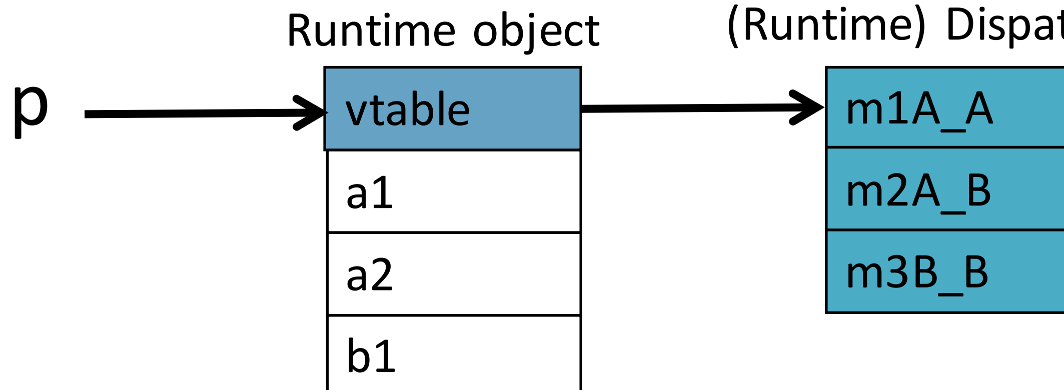
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

convert_ptr_to_B_to_ptr_to_A(p)

p.m2(3);

p→dispatch_table→m2A(, 3);



Multiple Inheritance

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Multiple Inheritance

- Allows unifying behaviors
- But raises semantic difficulties
  - Ambiguity of classes
  - Repeated inheritance
- Hard to implement
  - Semantic analysis
  - Code generation
    - Prefixing no longer work
    - Need to generate code for downcasts
- Hard to use

A simple implementation

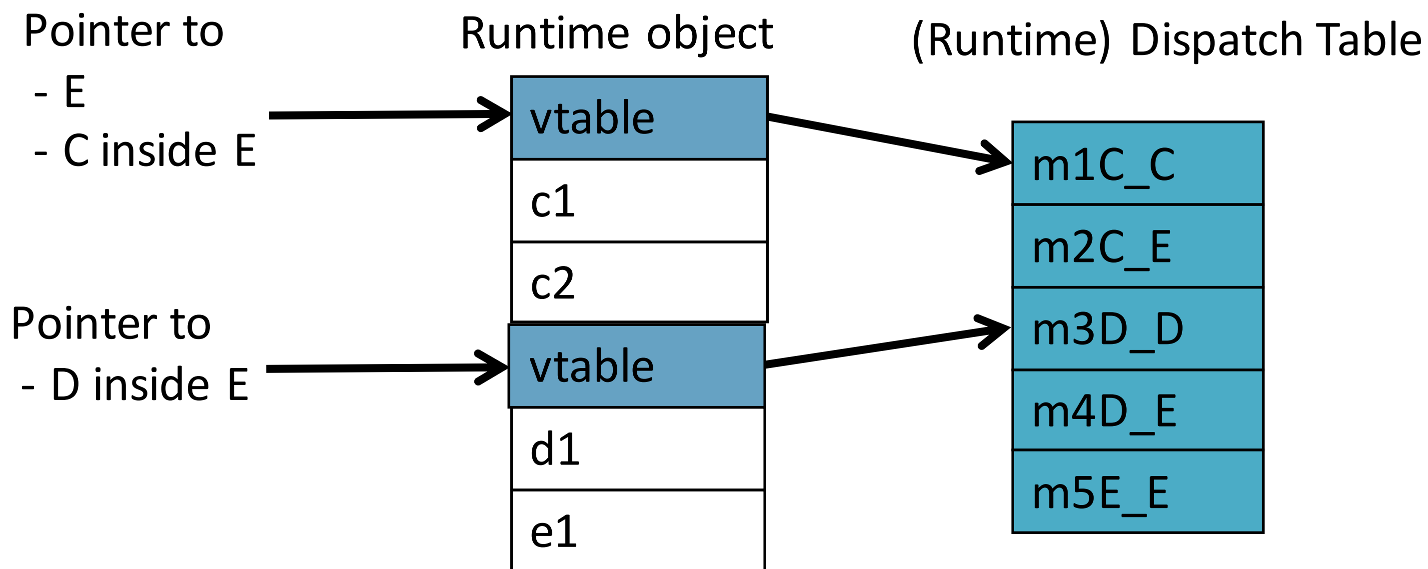
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

A simple implementation

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```



Dependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}

class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}

class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}

class E extends C, D {
    field e1;

    method m2()  {…}
    method m4()  {…}
    method m5(){…}
}
```

Interface Types

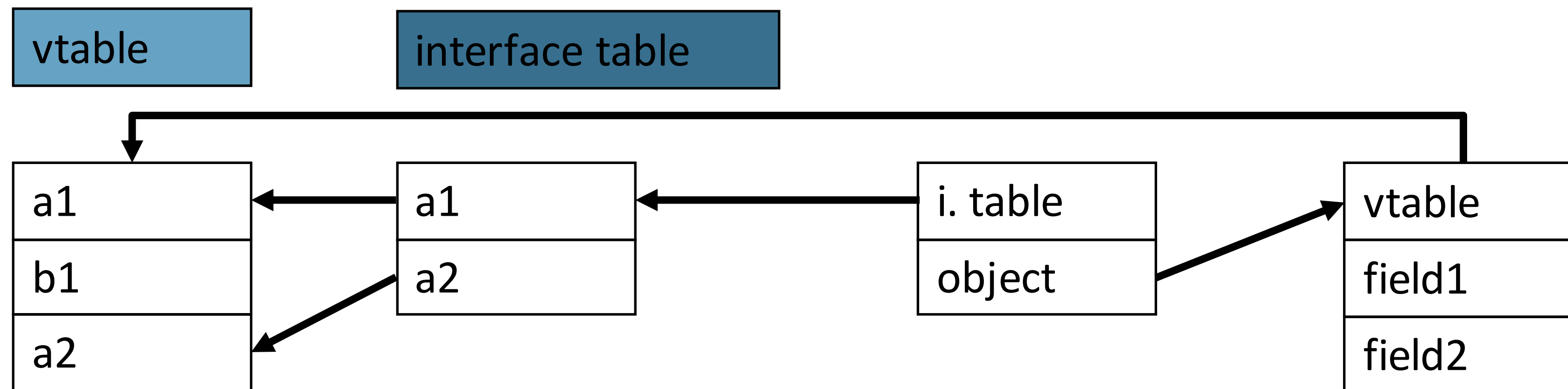
- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields

```
public interface Comparable {
    public int compare(Comparable o);
}
```

- A class can implement multiple interfaces
  Simpler to implement/understand/use

Interface Types

- Implementation: record with 2 pointers:
  - A separate dispatch table per interface
  - A pointer to the object



Assembly → Image

Source file (*e.g., utils*)	Source file (*e.g., main*)	library
Compiler	**Compiler**	**Compiler**
Assembly (.s)	Assembly (.s)	Assembly (.s)
Assembler	**Assembler**	**Assembler**
Object (.o)	Object (.o)	Object (.o)

Linker

Executable (“.elf”)

Loader

Image (in memory):

Assembler

- Converts (symbolic) assembler to binary (object) code
    - Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory
  - Yet another(simple) compiler
    - One-to one translation
- Converts constants to machine repr. (3➔0…011)
- Resolve internal references
- Records info for code & data relocation

Object File Format

Header	Text Segment	Data Segment	Relocation Information	Symbol Table	Debugging Information

- Header: Admin info + “file map”
- Text seg.: machine instruction
- Data seg.: (Initialized) data in machine format
- Relocation info: instructions and data that depend on absolute addresses
- Symbol table: “exported” references + unresolved references

Resolving Internal Addresses

- Two scans of the code
  - Construct a table label → address
  - Replace labels with values

- One scan of the code (Backpatching)
  - Simultaneously construct the table and resolve symbolic addresses
    - Maintains list of unresolved labels
  - Useful beyond assemblers

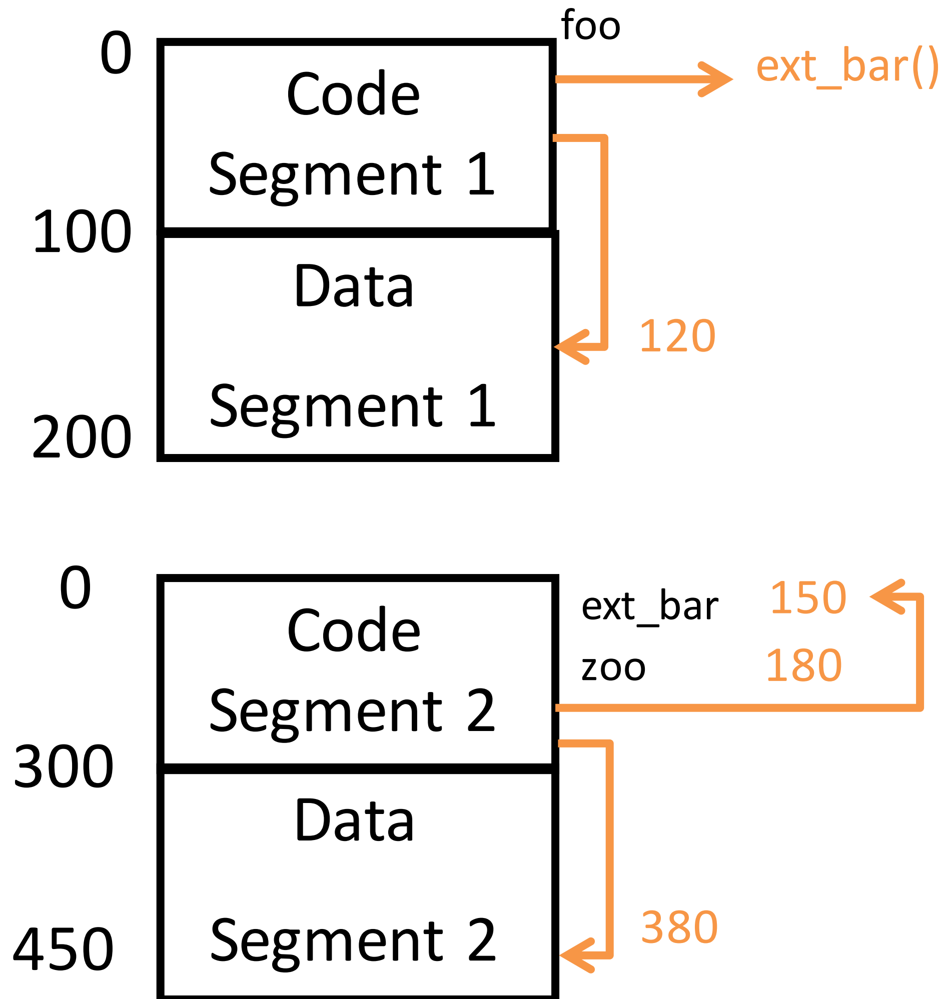
Handling External Addresses

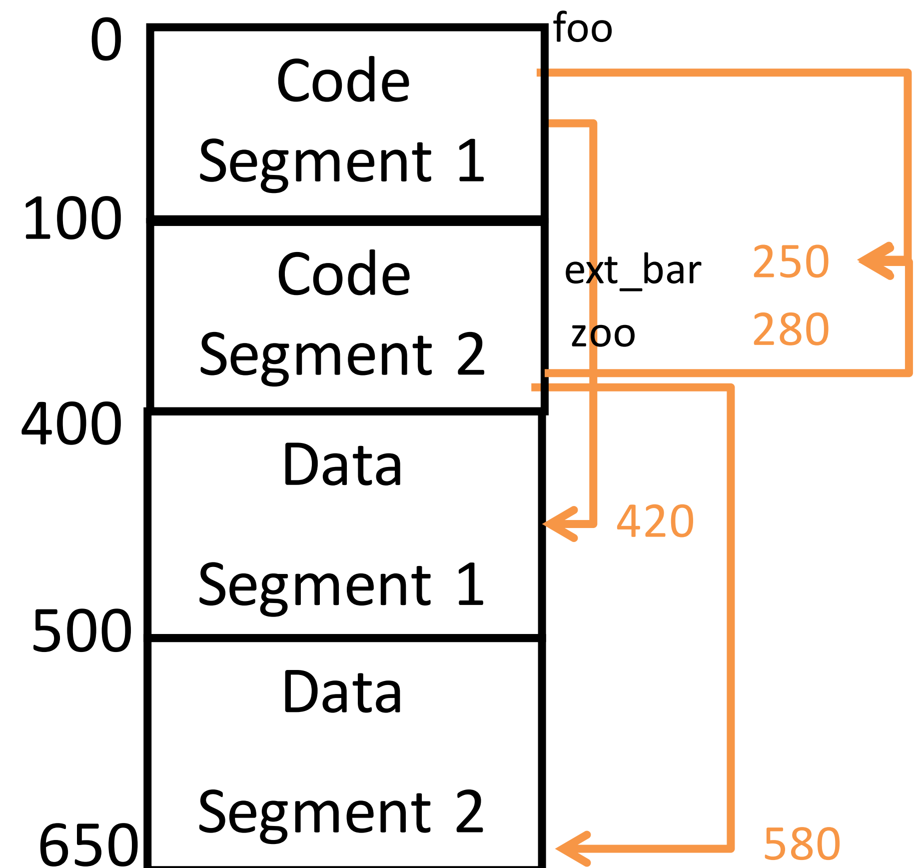
- Record symbol table in “external” table
  - Exported (defined) symbols
    - G, foo()
  - Imported (required) symbols
    - Extern_G, extern_bar(), printf()
- Relocation bits
  - Mark instructions that depend on absolute (fixed) addresses
    - Instructions using globals,

Linker

- Merges object files to an executable
  - Enables separate compilation
- Combine memory layouts of object modules
  - Links program calls to library routines
    - printf(), malloc()
  - Relocates instructions by adjusting absolute references
  - Resolves references among files

Linker





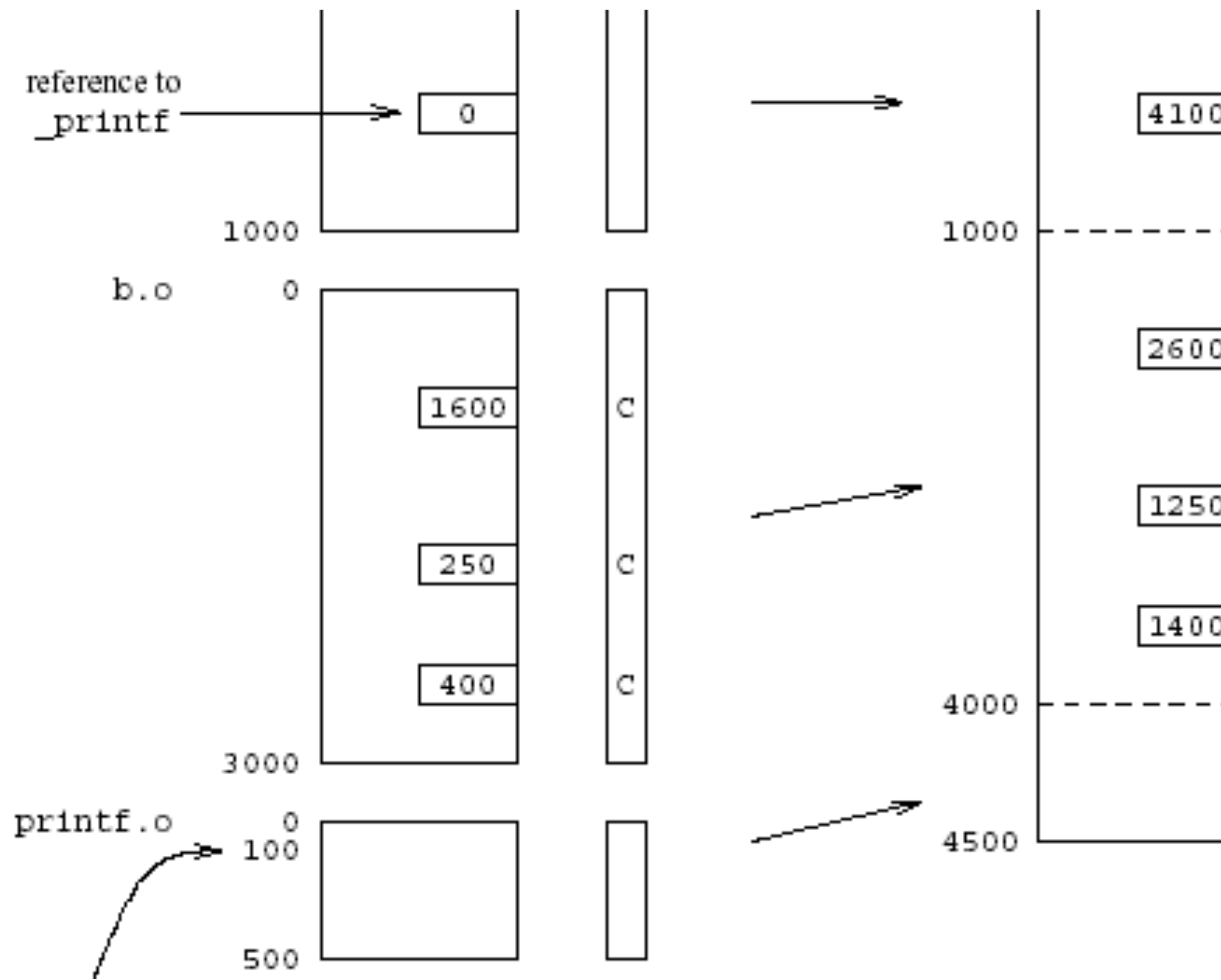
External References

- The code may include references to external names (identifiers)
  - Library calls
  - External data
- Stored in external symbol table

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Example



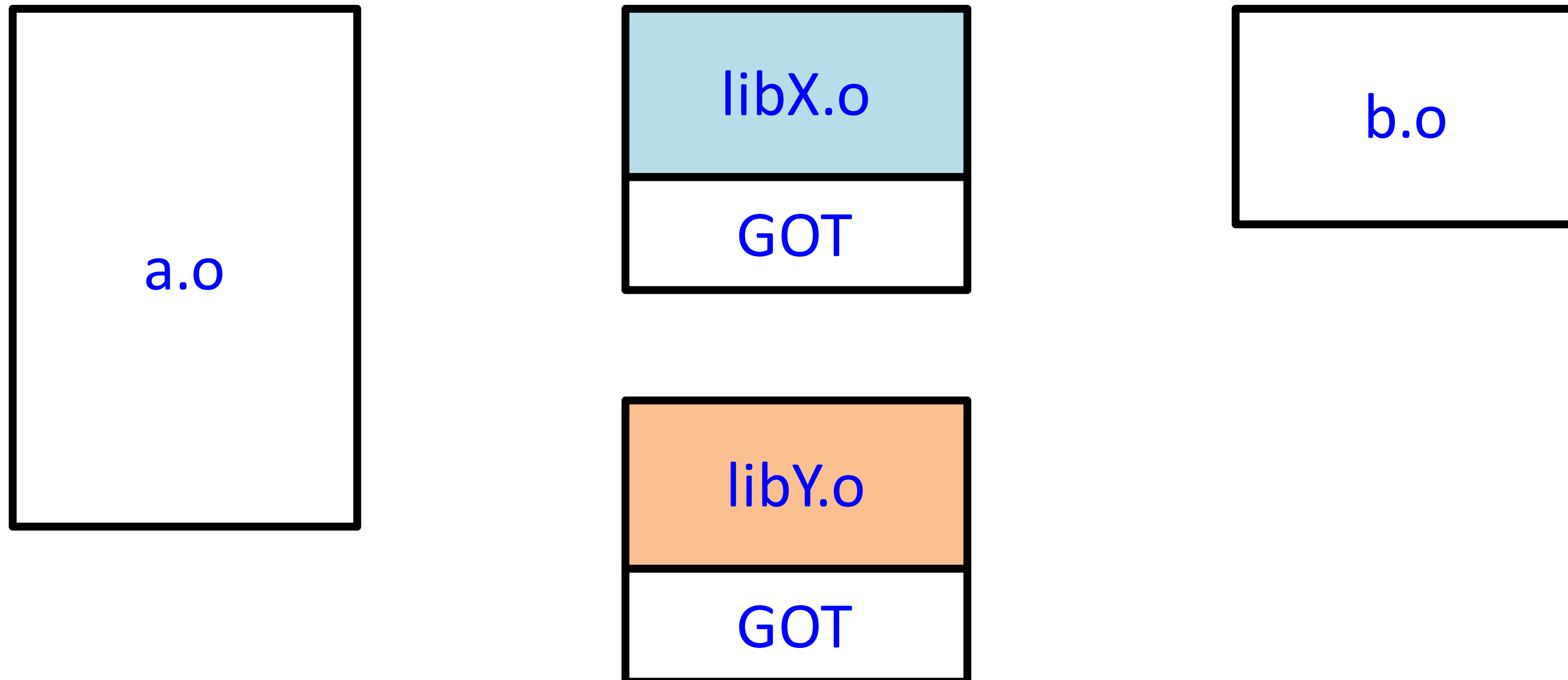
Loader (Summary)

- Initializes the runtime state
- Part of the operating system
  - Privileged mode
- Does not depend on the programming language
- “Invisible activation record”

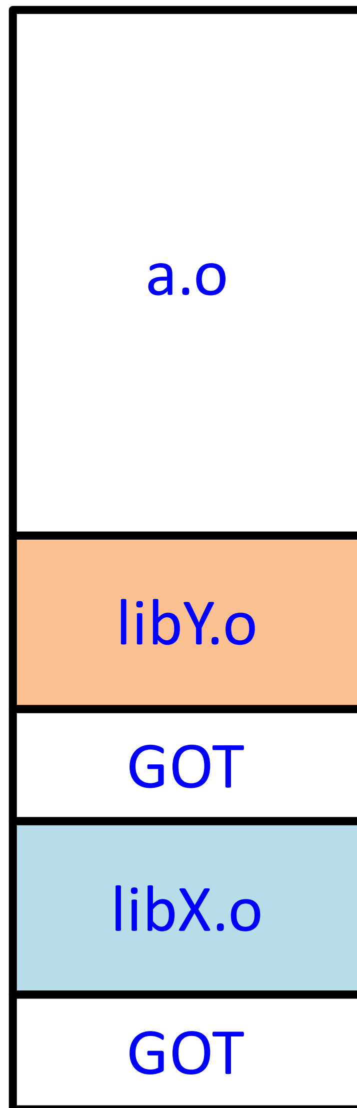
Dynamic Linking

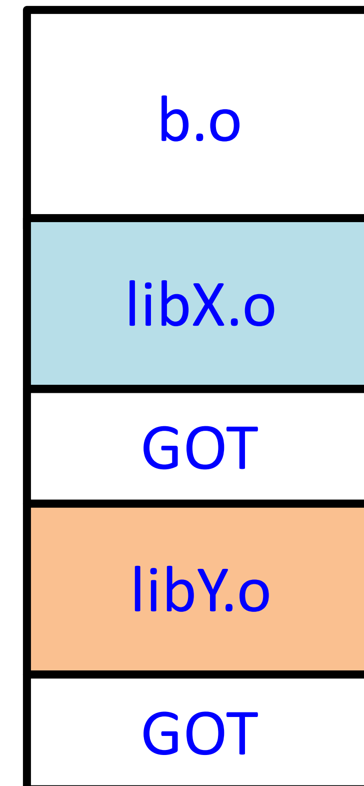
- Why dynamic linking?
  - Shared libraries
    - Save space
    - Consistency
  - Dynamic loading
    - Load on demand

Sharing code

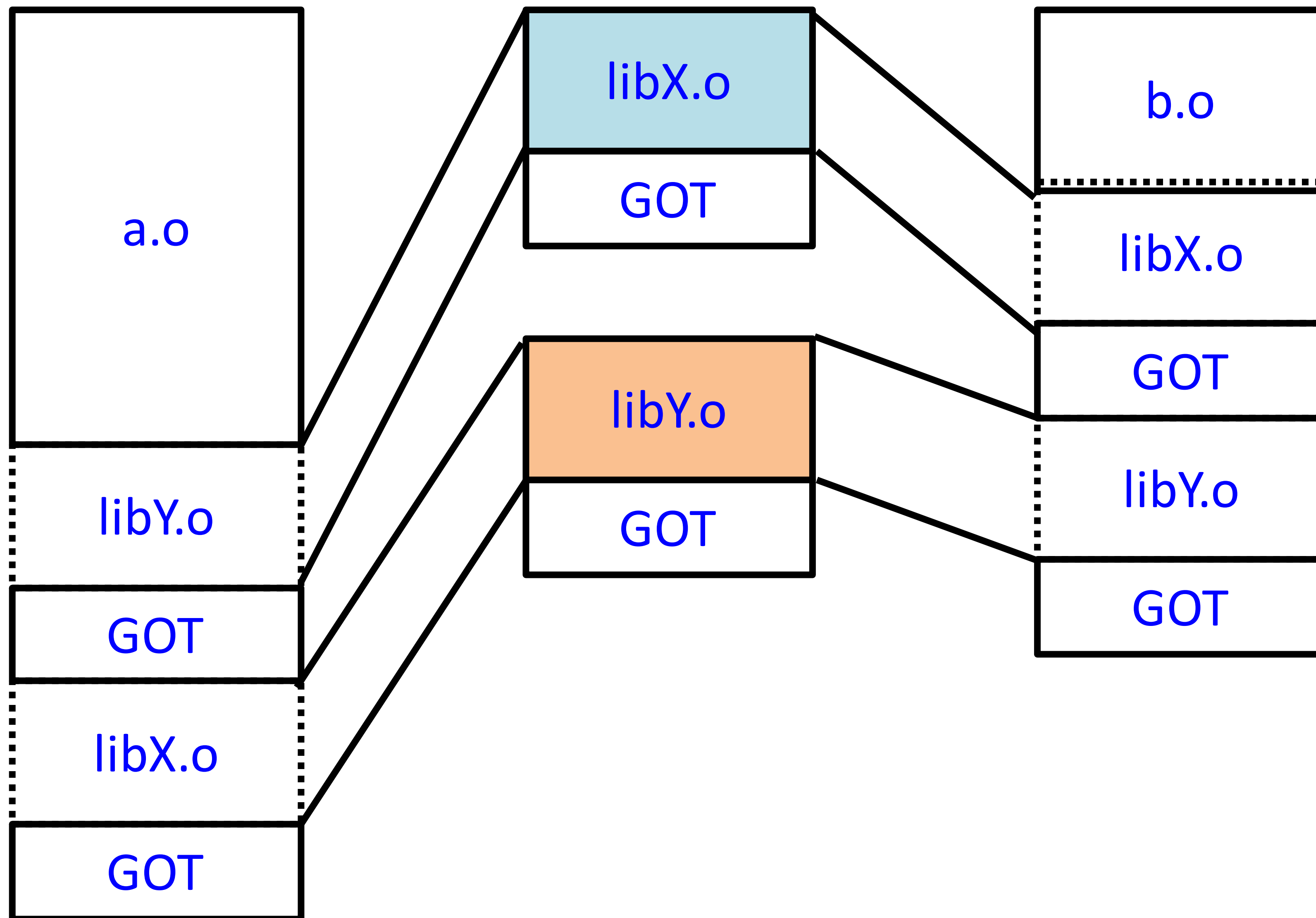


Sharing code





Sharing code



Position-Independent Code (PIC)

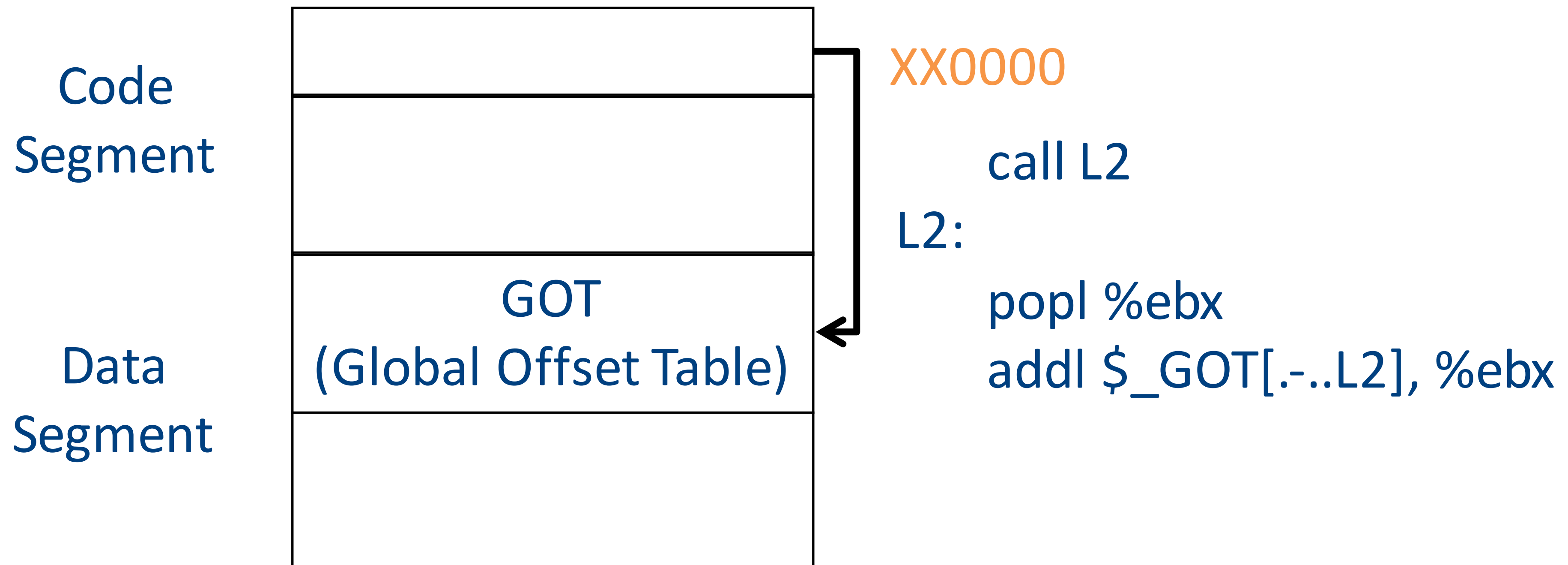
- Code which does not need to be changed regardless of the address in which it is loaded
  - Enable loading the same object file at different addresses
    - Thus, shared libraries and dynamic loading

- “Good” instructions for PIC: use relative addresses
  - relative jumps
  - reference to activation records

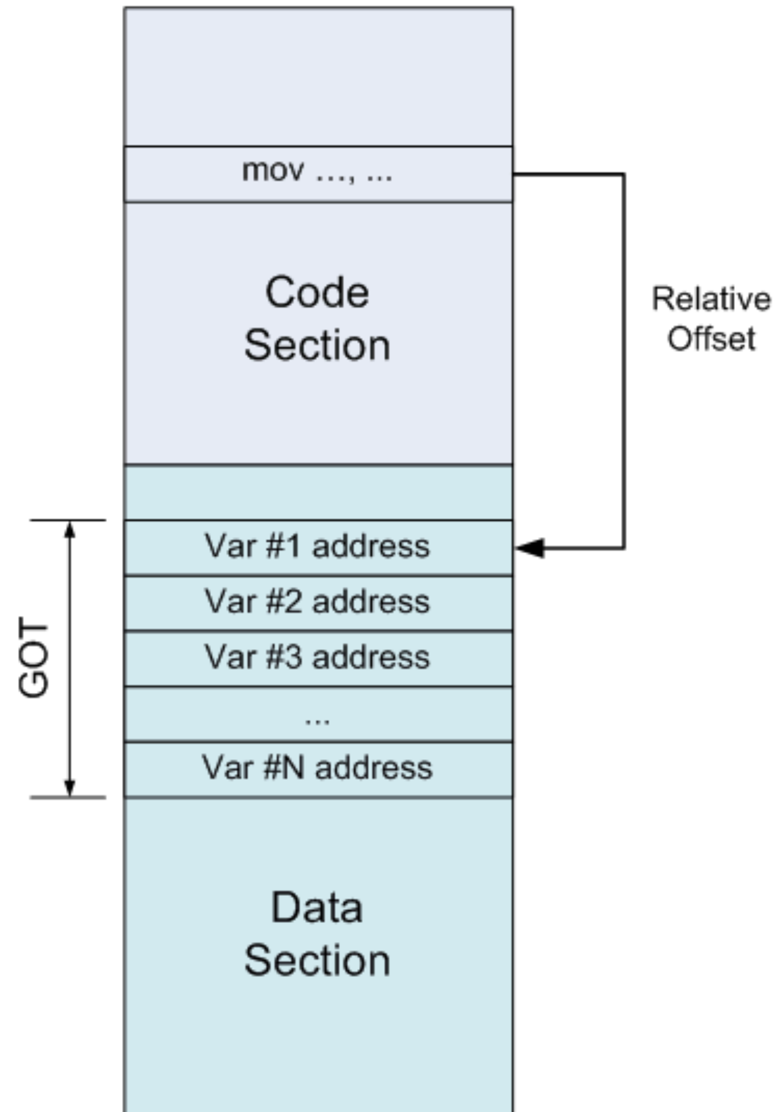
- “Bad” instructions for : use fixed addresses
  - Accessing global and static data
  - Procedure calls
    - Where are the library procedures located?

ELF-Position Independent Code

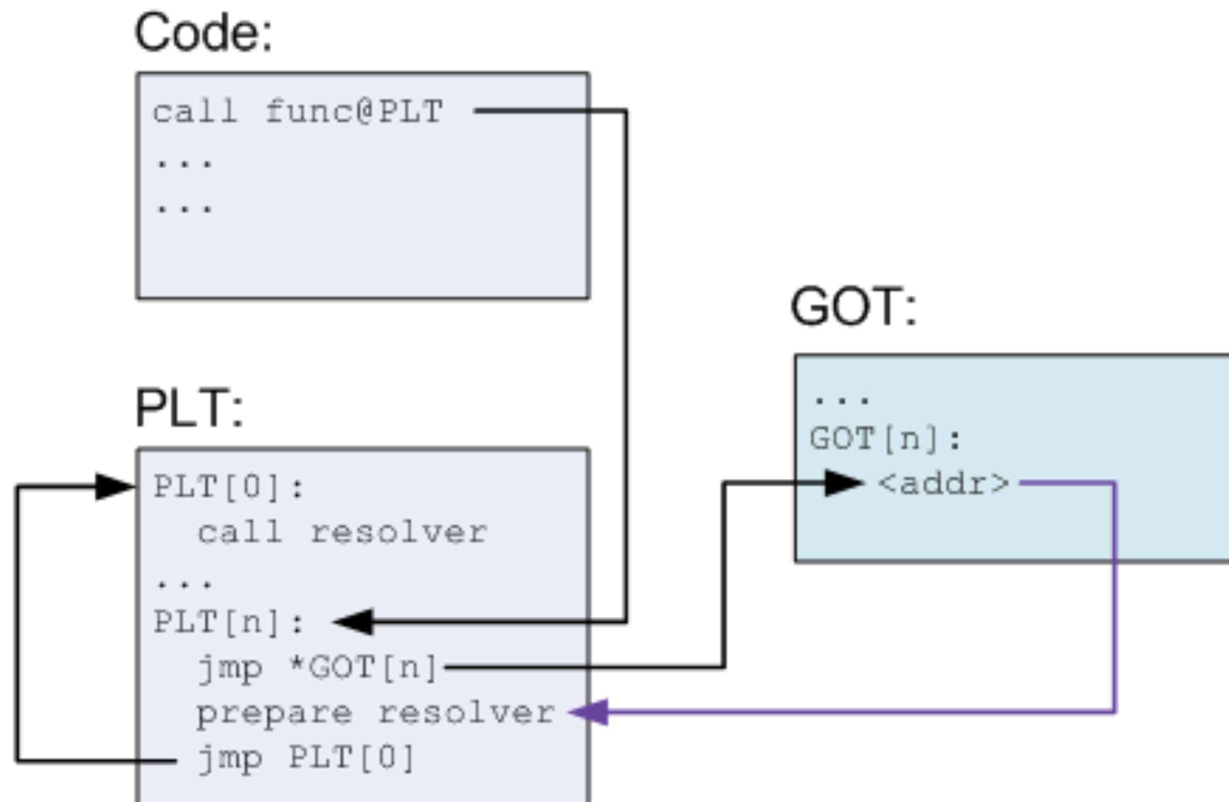
- Executable and Linkable code Format
  - Introduced in Unix System V
- Observation
  - Executable consists of code followed by data
  - The offset of the data from the beginning of the code is known at compile-time



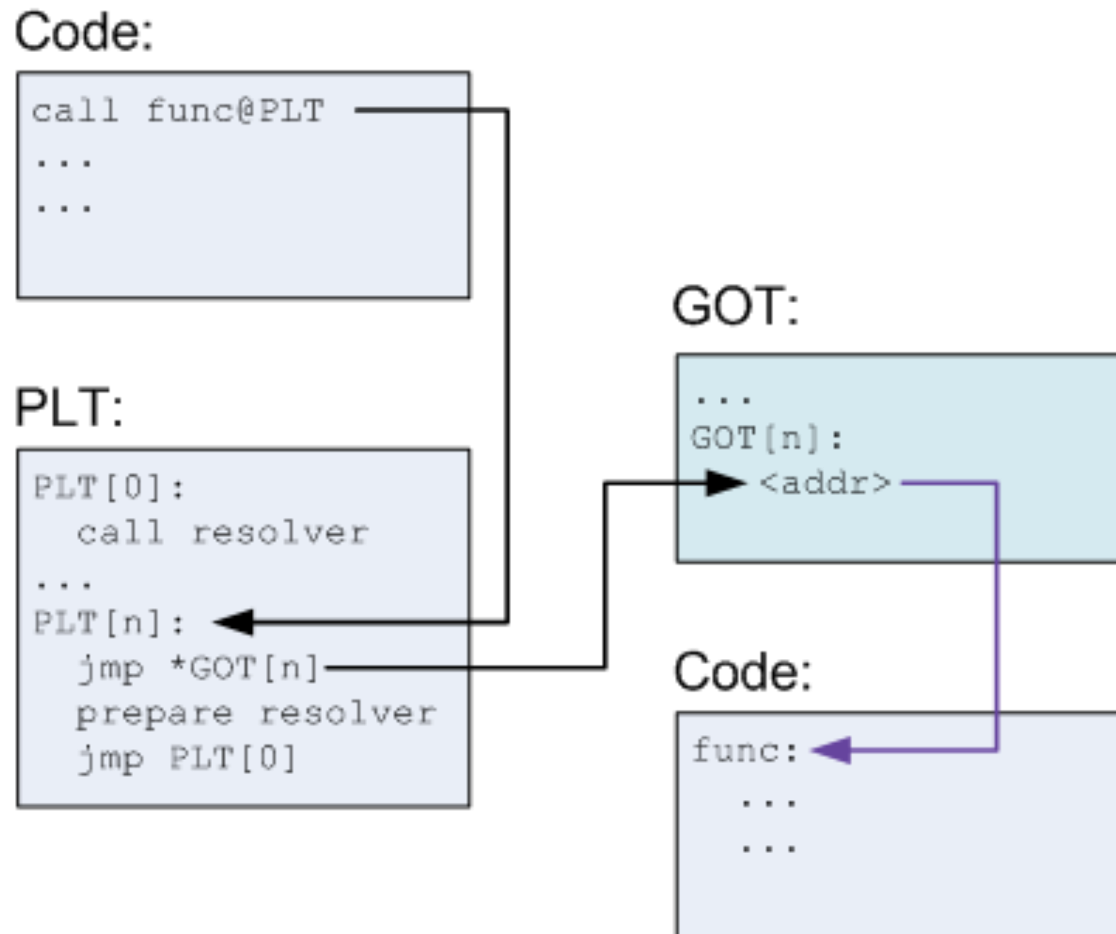
ELF: Accessing global data



ELF: Calling Procedures
(before 1st call)



ELF: Calling Procedures
(after 1st call)



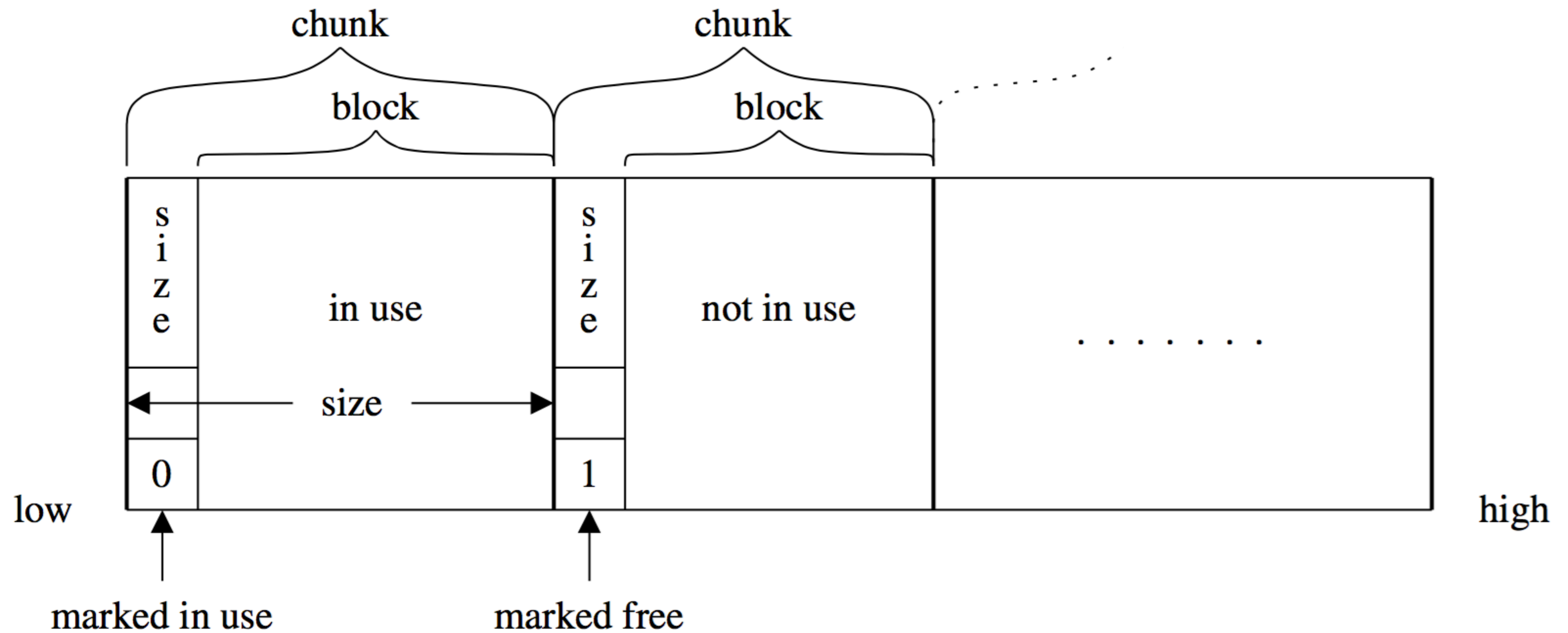
Memory Management

- Manual memory management
- Automatic memory management

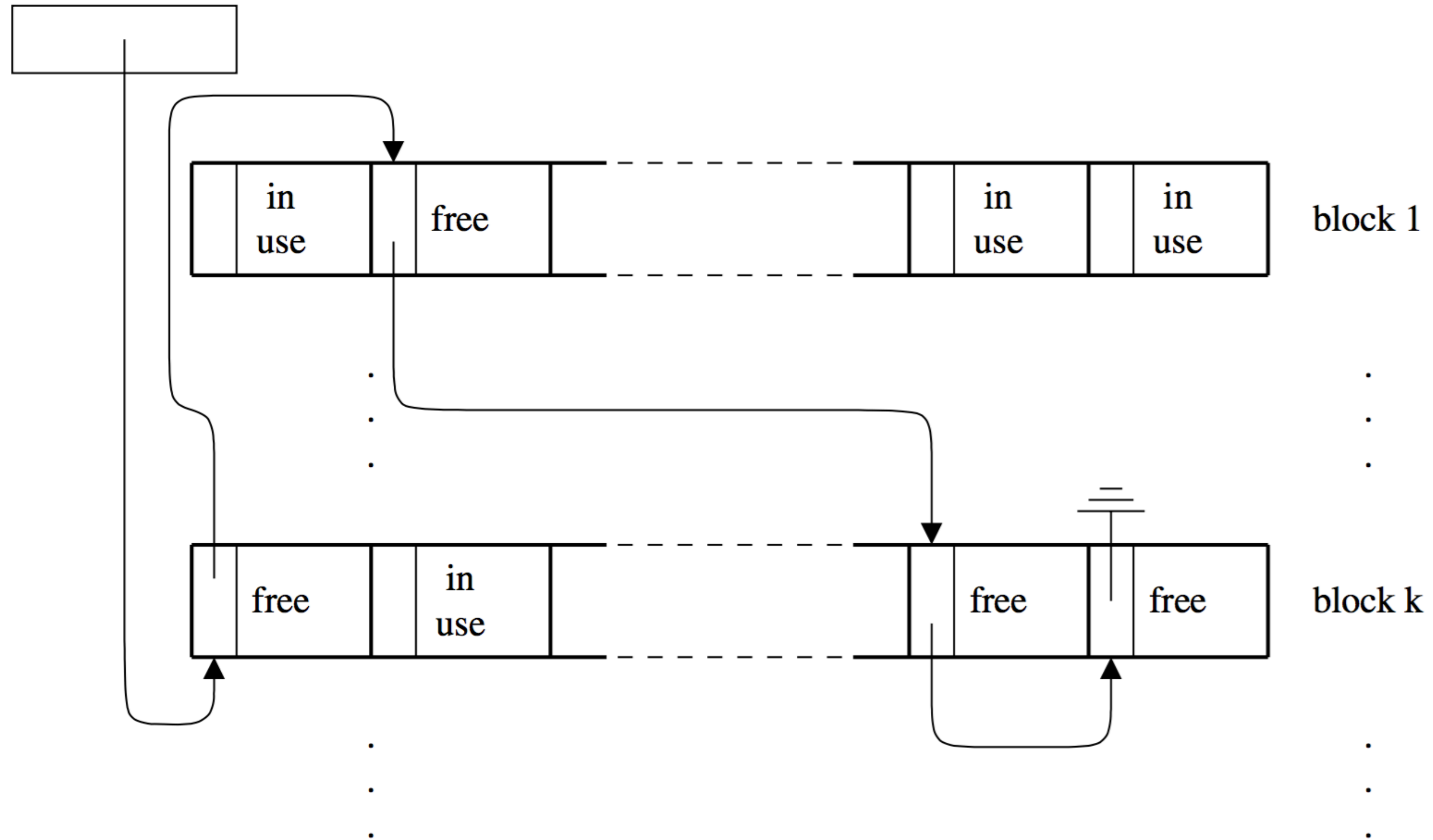
Free-list Allocation

- A data structure records the location and size of free cells of memory.
- The allocator considers each free cell in turn, and according to some policy, chooses one to allocate.
- Three basic types of free-list allocation:
  - First-fit
  - Next-fit
  - Best-fit

Memory chunks



Free list



free

- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

Garbage collection

- approximate reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead

Garbage Collection – Classical Techniques

- reference counting
- mark and sweep
- copying

GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when (rc==0) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)

The Mark-and-Sweep Algorithm [McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object
- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

Mark&Sweep in Depth

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

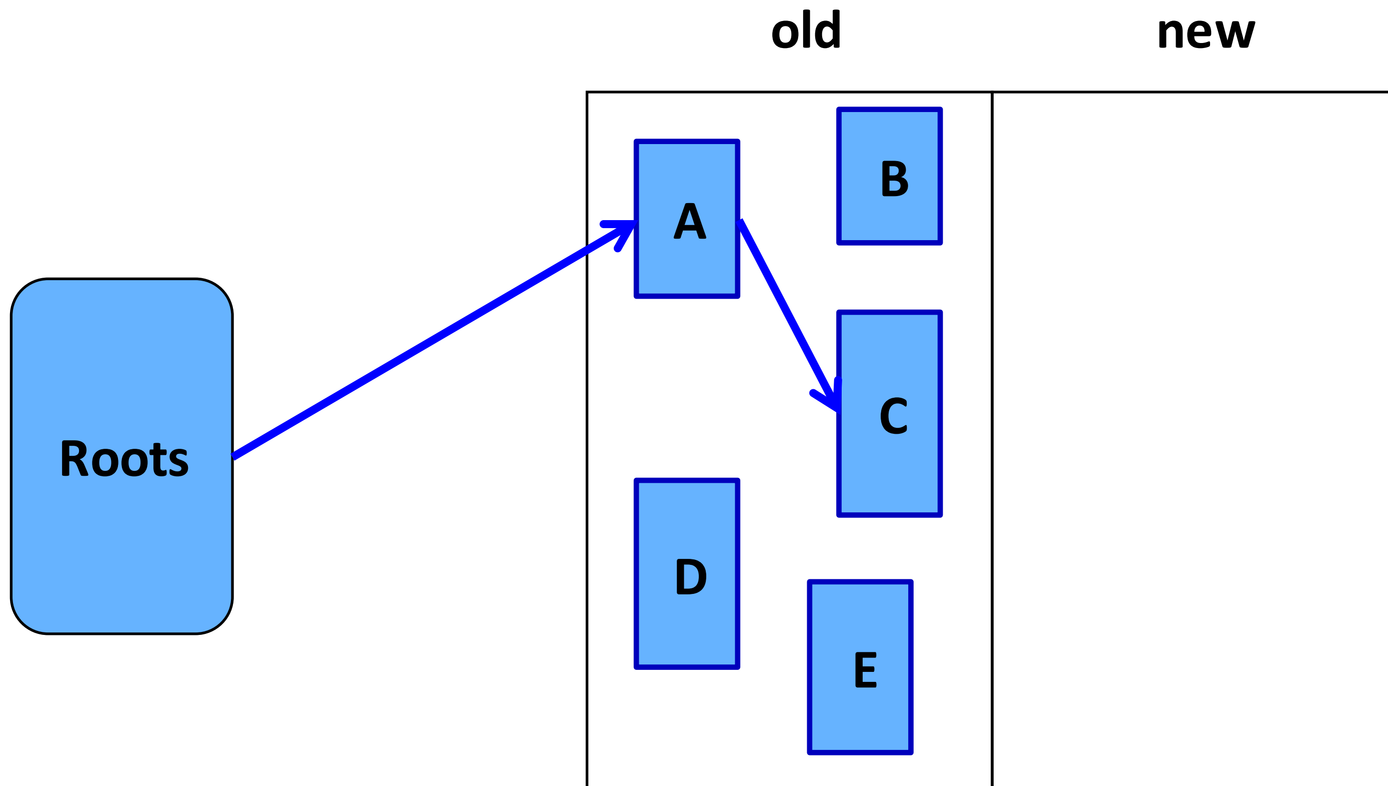
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Copying GC

- partition the heap into two parts
  - old space
  - new space

- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - swap roles of old/new space

Example



Example