Compilation

0368-3133 2016/17a

Lecture 11

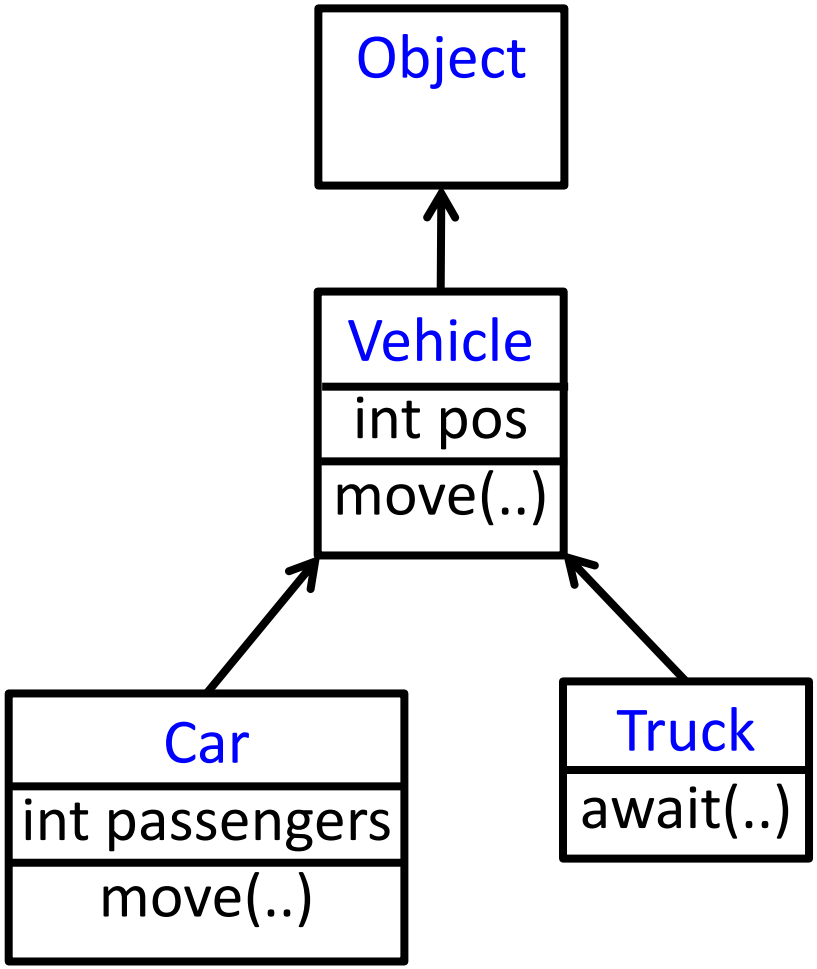
Compiling Object-Oriented Programs

Noam Rinetzky

Object Oriented Programs

- C++, Java, C#, Python, …
- Main abstraction: **Objects** (usually of type called class)
  - Code
  - Data
- Naturally supports **A**bstract **D**ata **T**ype implementations
- Information hiding
- Evolution & reusability
- Important characteristic: Extension/Inheritance

A Simple Example



A Simple Example

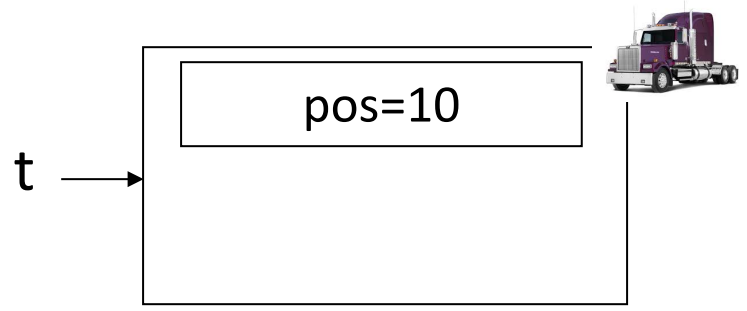
```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

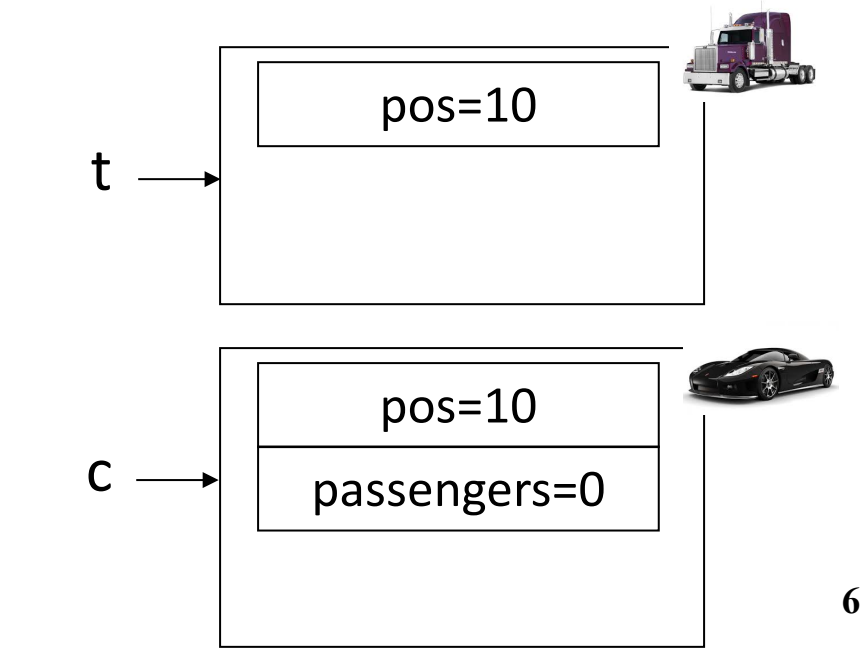
```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

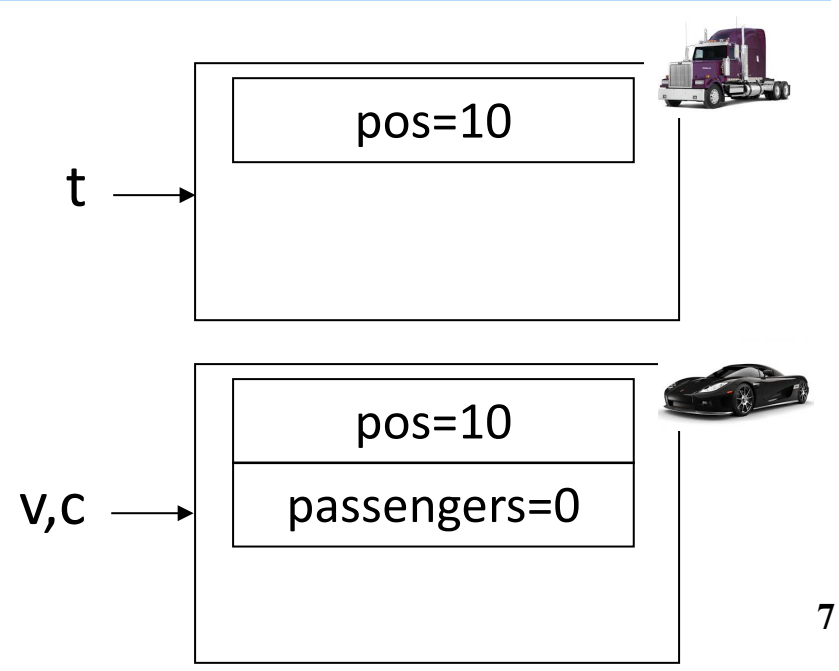
```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

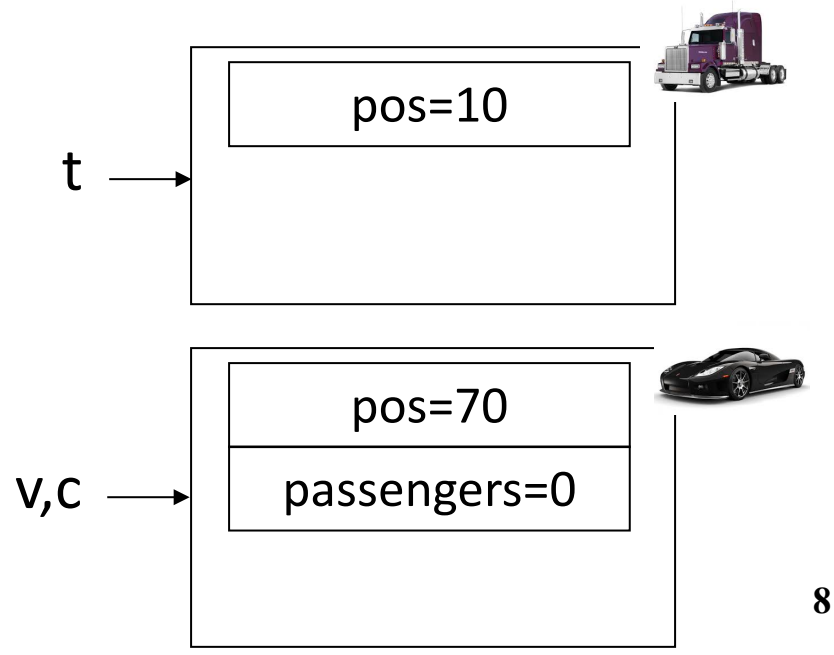
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

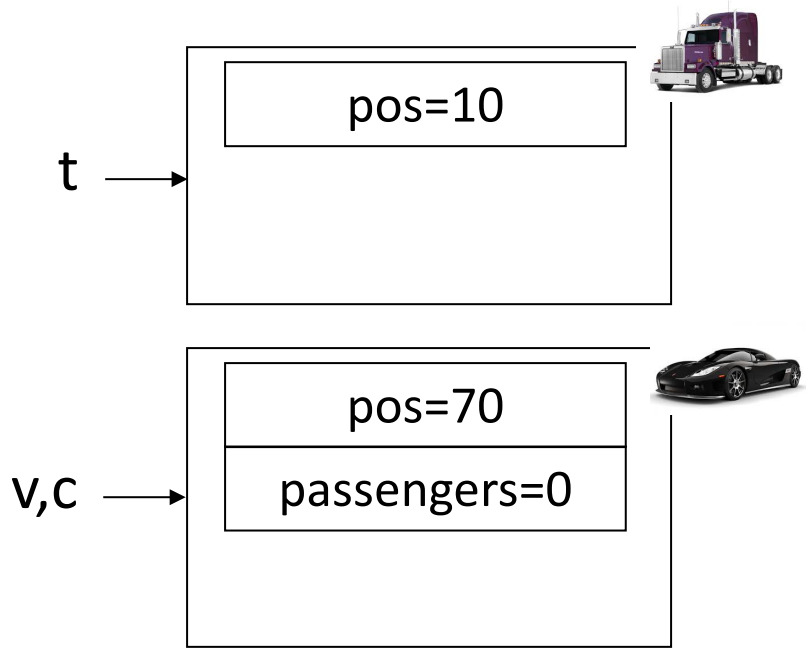
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

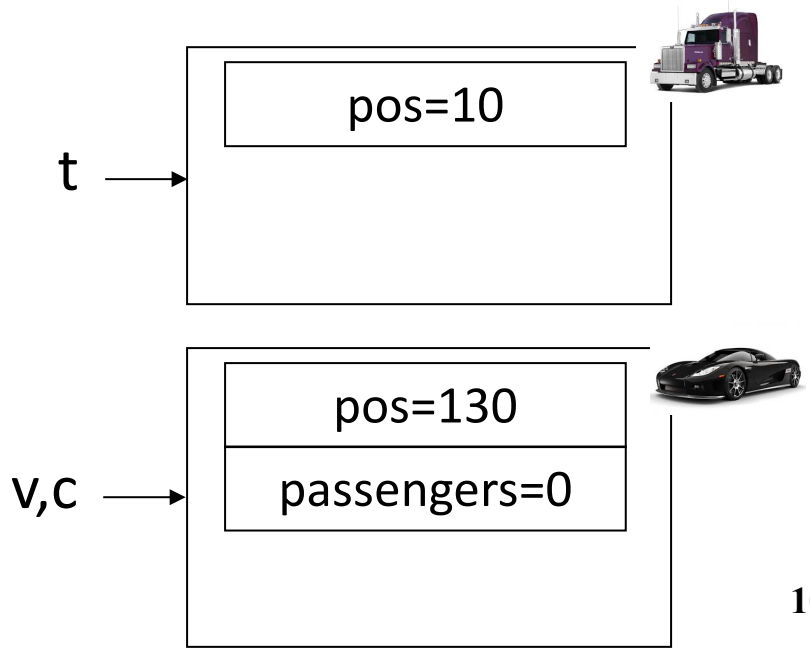
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



Translation into C

Translation into C (Vehicle)

```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;
```

Translation into C (Vehicle)

```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;

void NewVe(Ve *this){
  this→pos = 10;
}

void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Translation into C (Truck)

```
class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}
```

```
typedef struct Truck {
  int pos;
} Tr;

void NewTr(Tr *this){
  this→pos = 10;
}

void moveTr(Ve *this, int x){
  if (x<55)
    this→pos = this→pos + x;
}
```

Naïve Translation into C (Car)

```
class Car extends Vehicle {
 int passengers = 0;
 void await(vehicle v){
  if (v.pos < pos)
    v.move(pos – v.pos);
  else
    this.move(10);
 }
}
```

```
typedef struct Car{
  int pos;
  int passengers;
} Ca;

void NewCa (Ca *this){
  this→pos = 10;
  this→passengers = 0;
}

void awaitCa(Ca *this, Ve *v){
  if (v→pos < this→pos)
    moveVe(this→pos – v→pos)
  else
    MoveCa(this, 10)
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;
```

```
typedef struct Car{
  int pos;
  int passengers;
} Ca;
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

Vehicle x = t;
x.move(20);

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
Ve *x = t;
moveTr((Tr*)x, 20);
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

Vehicle x = t;
x.move(20);

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
Ve *x = t;
moveTr((Tr*)x, 20);
```

```
void moveVe(Ve *this, int x){…}
```

```
void moveTr(Ve *this, int x){…}
```

Translation into C

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

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

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA  *this, int i) {
   // Body of m2 with any object
   // field f as this->f
   …
}
```

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

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

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
a.m2(5)
```

```
m2A(a,5)
```

```
void m2A(classA  *this, int i) {
   // Body of m2 with any
   // object-field f as this->f
   …
}
```

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

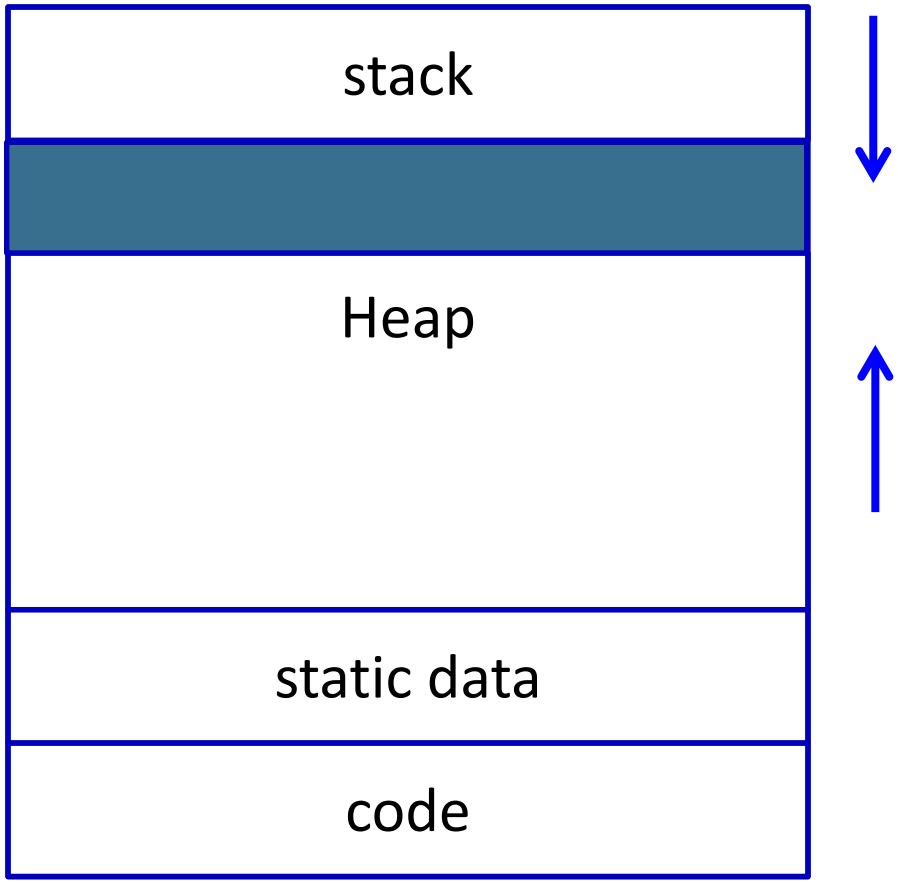
- “Translation into C”
- Powerful runtime environment

- Adding “gluing” code

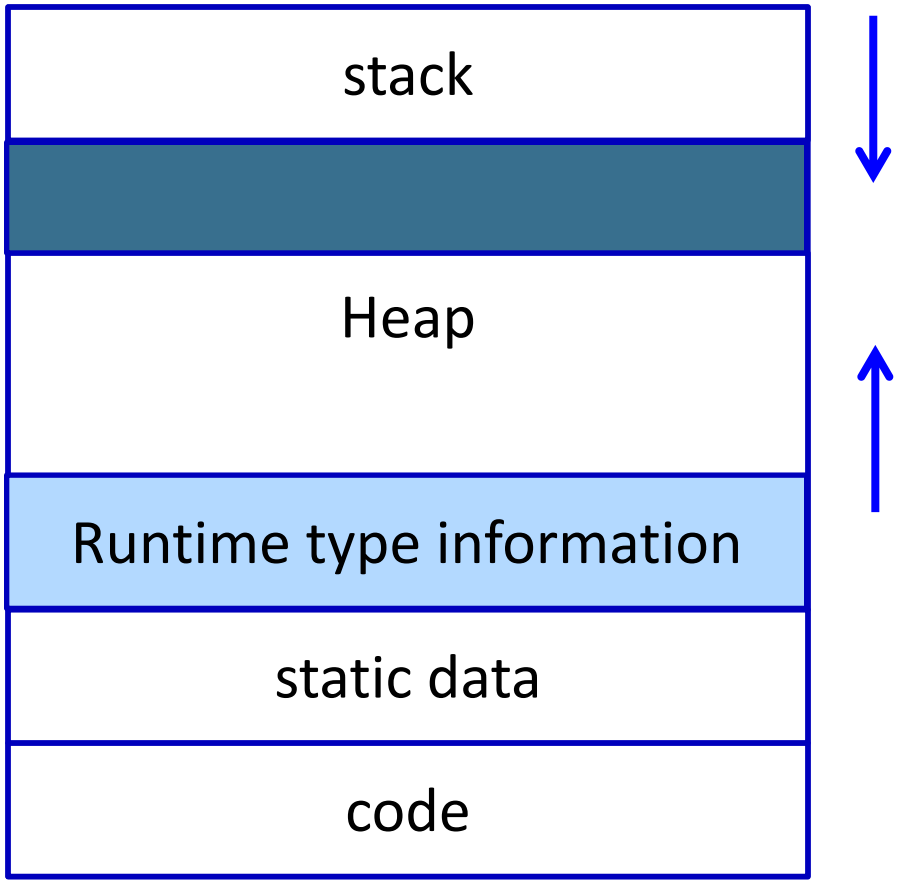
Runtime Environment

- 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**

Memory Layout



Memory Layout



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() {…}
}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

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

m2 is declared and defined

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

m2 is redefined

Method Overriding

- Redefines functionality
- Affects semantic analysis

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

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

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

- Redefines functionality
- Affects semantic analysis

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

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

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

declared

defined

Method Overriding

```
a.m2(5)  // class(a) = A
```

```
m2A_A(a, 5)
```

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

```
b.m2(5)  // class(b) = B
```

```
m2A_B(b, 5)
```

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

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

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

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

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

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

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

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

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

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

a.m2(5) // class(a) = A

m2A_A(a, 5)

b.m2(5) // class(b) = B

m2A_B(b, 5)

```
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) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Abstract methods & classes

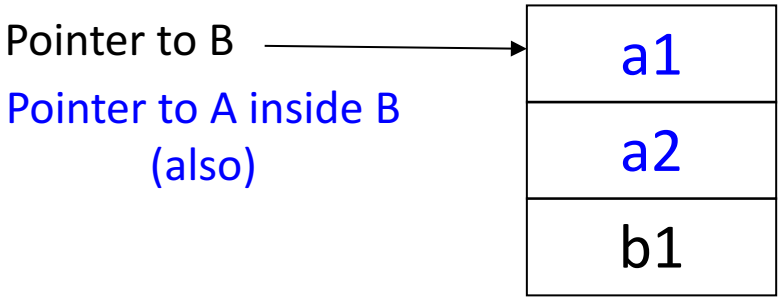
- Abstract methods
  - Declared separately, defined in child classes
  - E.g., C++ pure virtual methods, abstract methods in Java
- Abstract classes = class may have abstract methods
  - E.G., Java/C++ abstract classes
  - Abstract classes cannot be instantiated
- Abstract aka "virtual"
- Inheriting abstract class handled like regular inheritance
  - Compiler checks abstract classes are not allocated

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

Conceptual Impl. of Dynamic Binding

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

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

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

```
class B extends A {
    field b1;
    method m2() {
     … a3 …
    }
    method m3() {…}
}
```

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

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

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

”p.m2(3)”

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B: m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
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) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

”p.m2(3)”

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B: m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
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) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

More efficient implementation

- Apply pointer conversion in sublasses
  - Use dispatch table to invoke functions
  - Similar to table implementation of case

```
void m2A_B(classA *this_A) {
    Class_B *this = convert_ptr_to_A_ptr_to_A_B(this_A);
   …
}
```

More efficient implementation

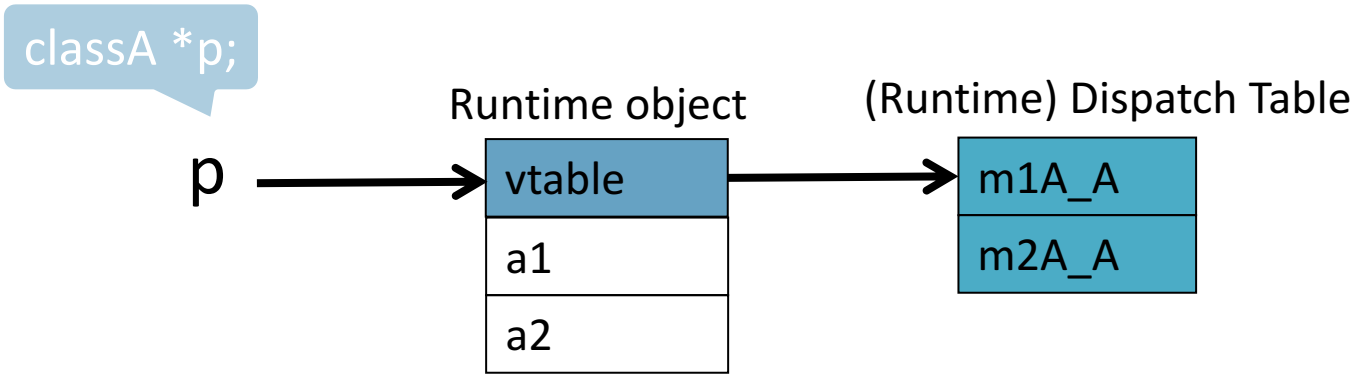
```
typedef struct  {
    field a1;
    field a2;
} A;

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){…}
```

More efficient implementation

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

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

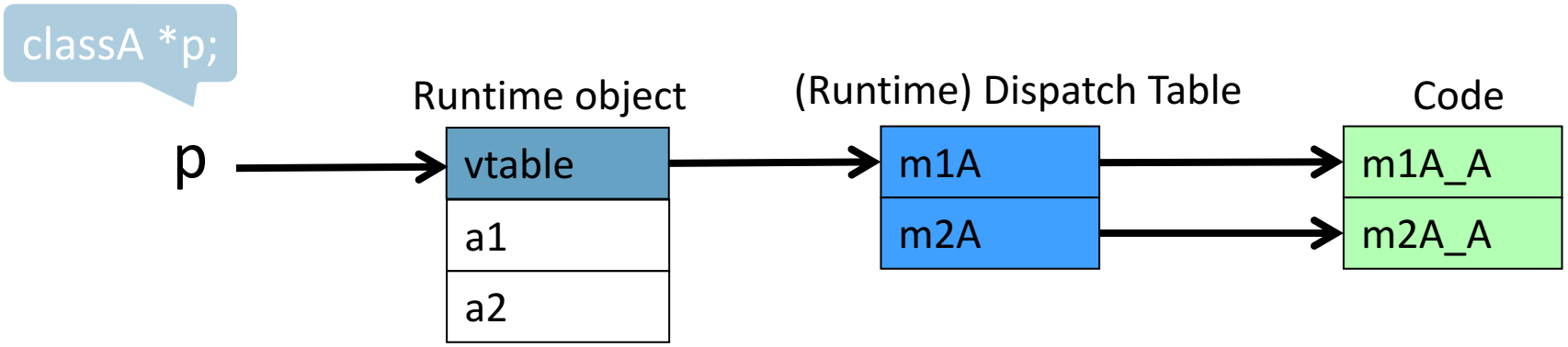
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

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

void m3B_B(B* this){…}
```



More efficient implementation

```
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){…}
```



More efficient implementation

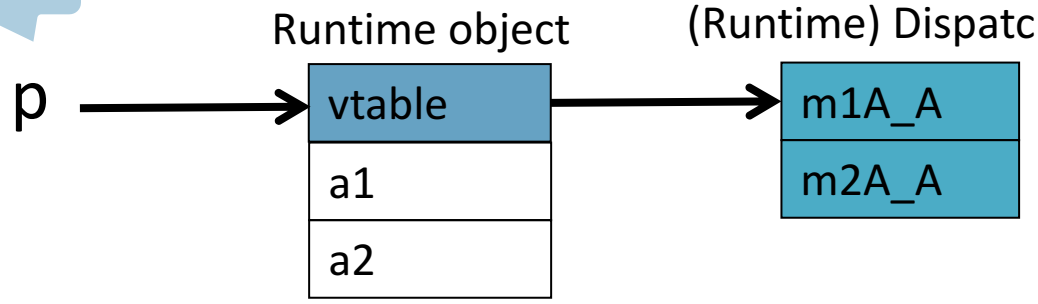
```
typedef struct  {
    field a1;
    field a2;
} A;

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){…}
```

classA *p;

p.m2(3);

p→dispatch_table→m2A(p, 3);



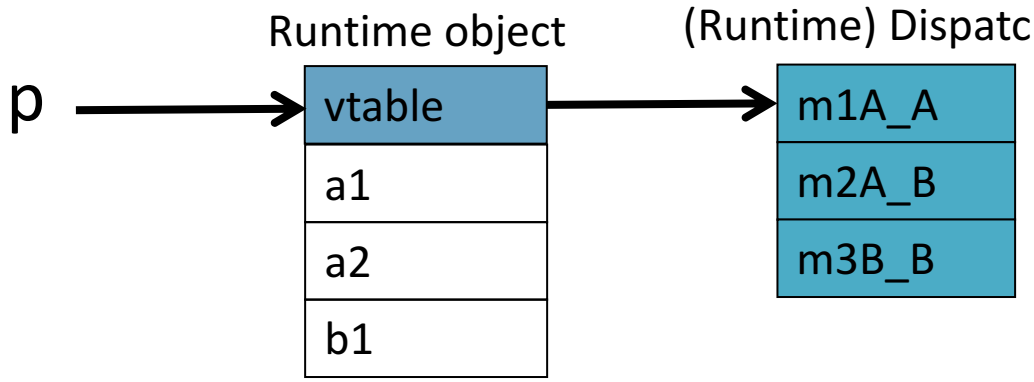
More efficient implementation

```
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){…}
```

p.m2(3);

p→dispatch_table→m2A(p, 3);



More efficient implementation

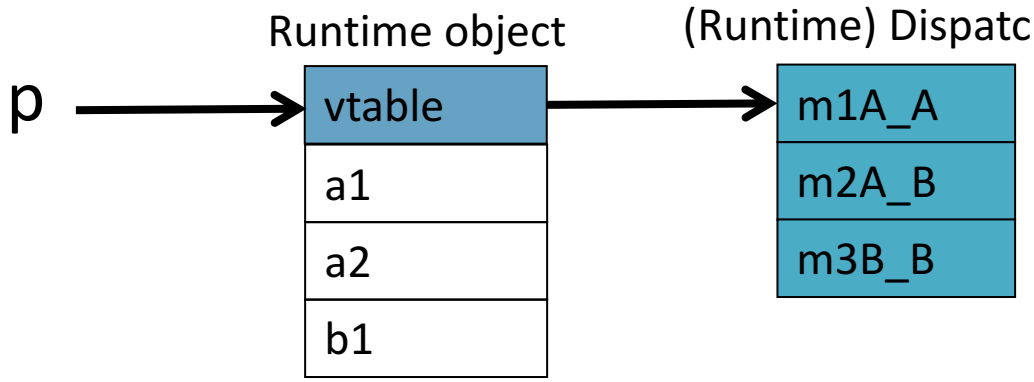
```
typedef struct  {
    field a1;
    field a2;
} A;

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);



Allocating objects (runtime)

- x = new A()

- Allocate memory for A's fields + pointer to vtable
- Initialize vtable to point to A's vtable
- initialize A's fields (call constructor)
- return object

Executing a.m2(5)

- Fetch the class descriptor at offset 0 from a
- Fetch the method-instance pointer p from the constant m2 offset of a
- call m2 (jump to address p)

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

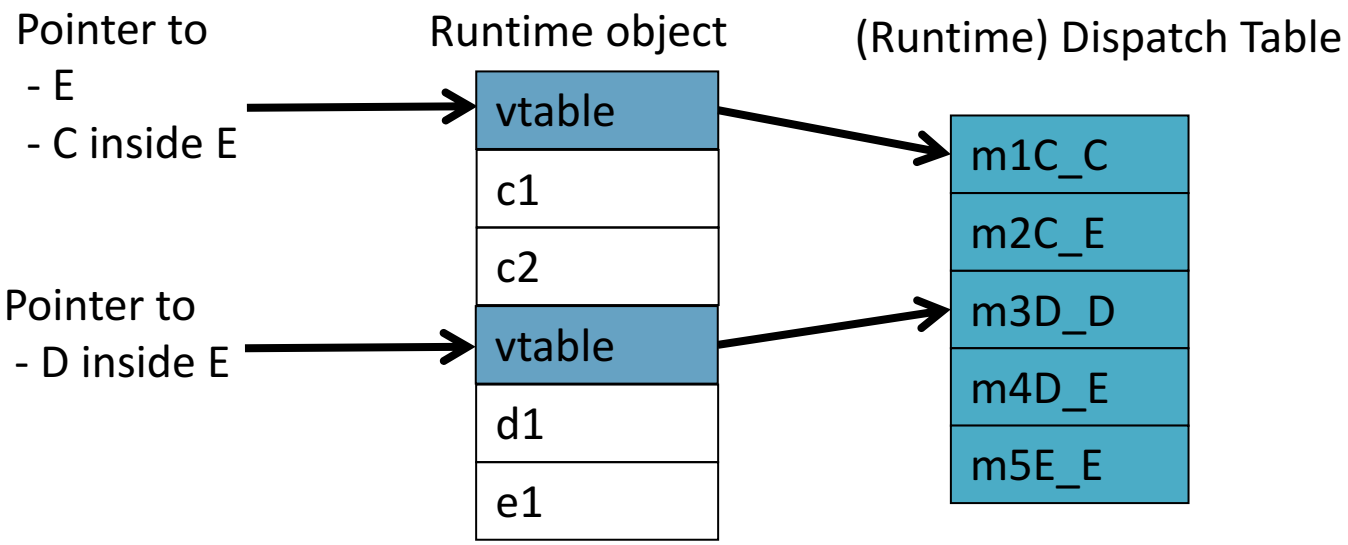
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

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(){…}
}
```



Downcasting (E→C,D)

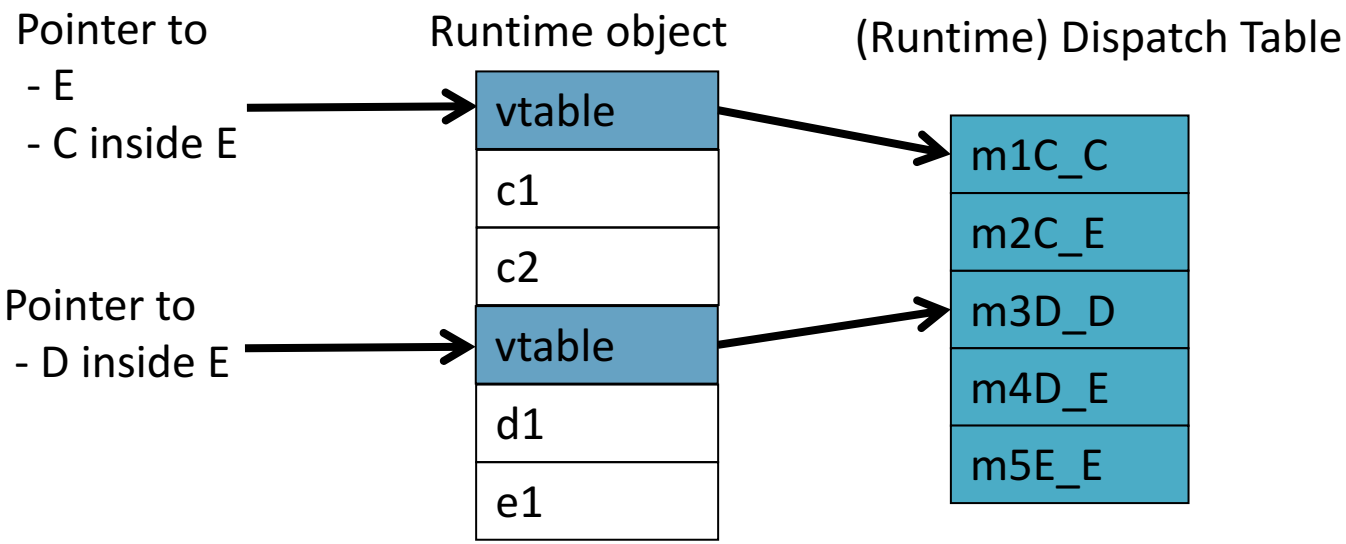
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

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

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

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

convert_ptr_to_E_to_ptr_to_C(e) = e;

convert_ptr_to_E_to_ptr_to_D(e) = e + sizeof(C);



Upcasting (C,D→E)

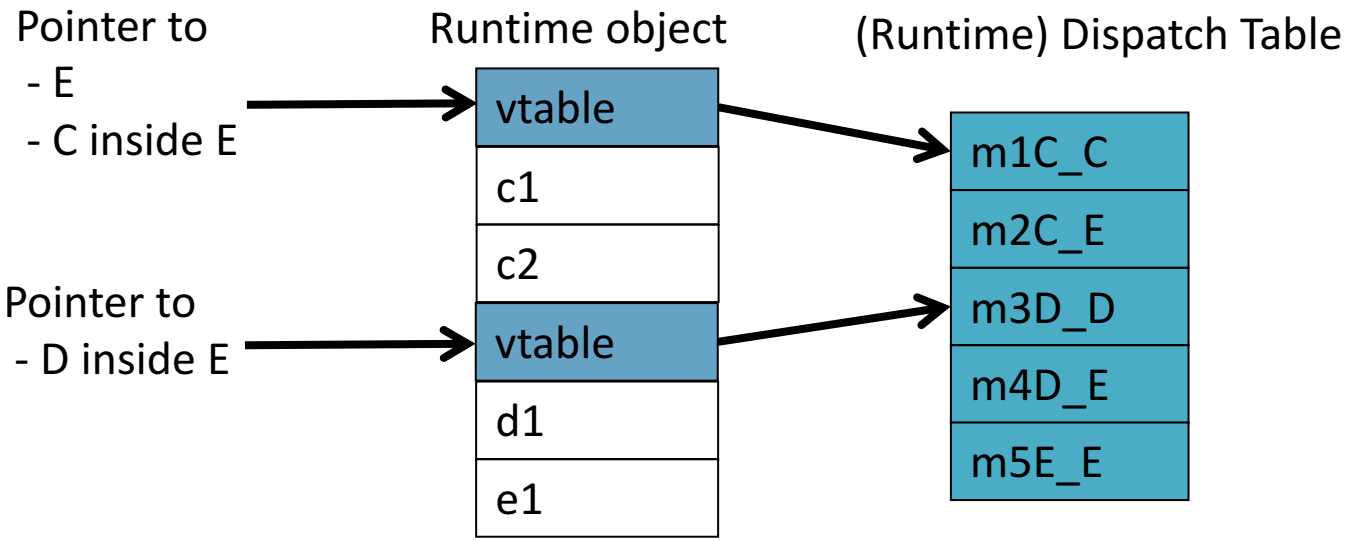
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

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

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

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

convert_ptr_to_C_to_ptr_to_E(c) = c;

convert_ptr_to_D_to_ptr_to_E(d) = d - sizeof(C);



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

```
class D extends A {
    field d1;

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

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

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

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

```
class D extends A {
    field d1;

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

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

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

Independent multiple Inheritance

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

```
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(){…}
}
```

Independent multiple Inheritance

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

```
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 m3(){…}  //alt explicit qualification
    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

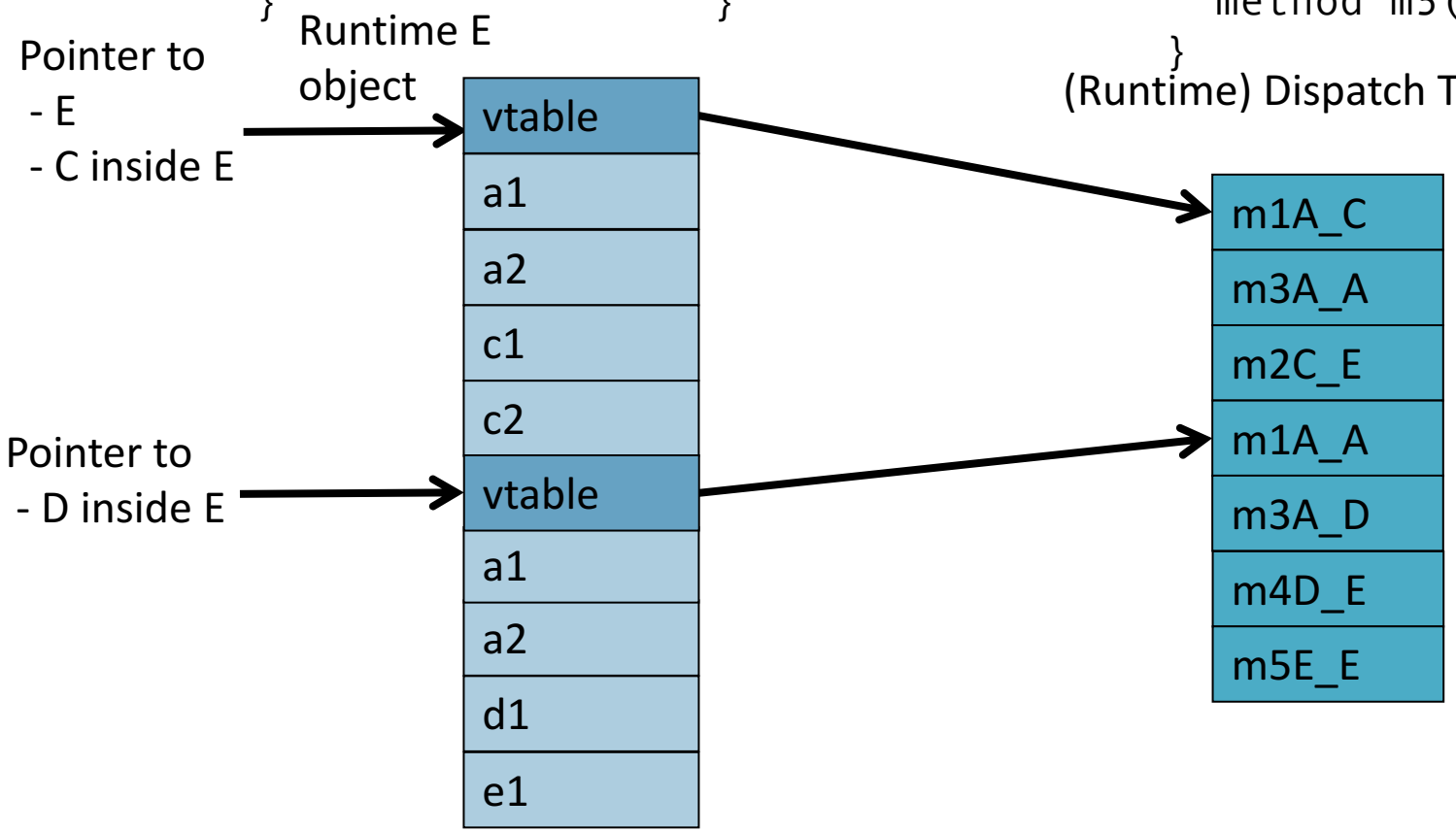
Independent Inheritance

```
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(){…}
}
```



Dependent multiple inheritance

- Superclasses share their own superclass

- The simple solution does not work
- The positions of nested fields do not agree

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(){…}
}
```

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(){…}
}
```

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(){…}
}
```

Dependent Inheritance

- Superclasses share their own superclass

- The simple solution does not work
- The positions of nested fields do not agree

Implementation

- Use an index table to access fields
- Access offsets indirectly

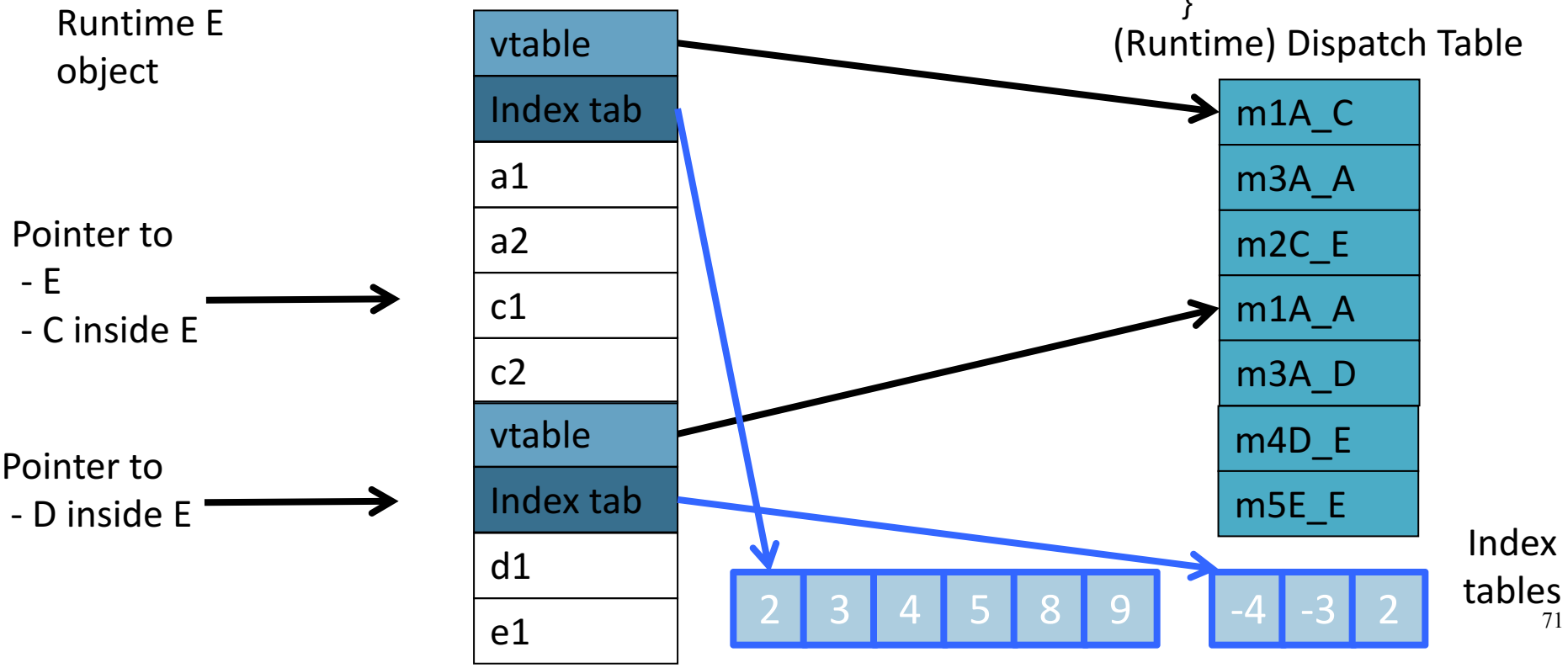
Implementation

```
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(){…}
}
```



Class Descriptors

- Runtime information associated with instances
- Dispatch tables
  - Invoked methods
- Index tables
- Shared between instances of the same class

- Can have more (reflection)

Interface Types

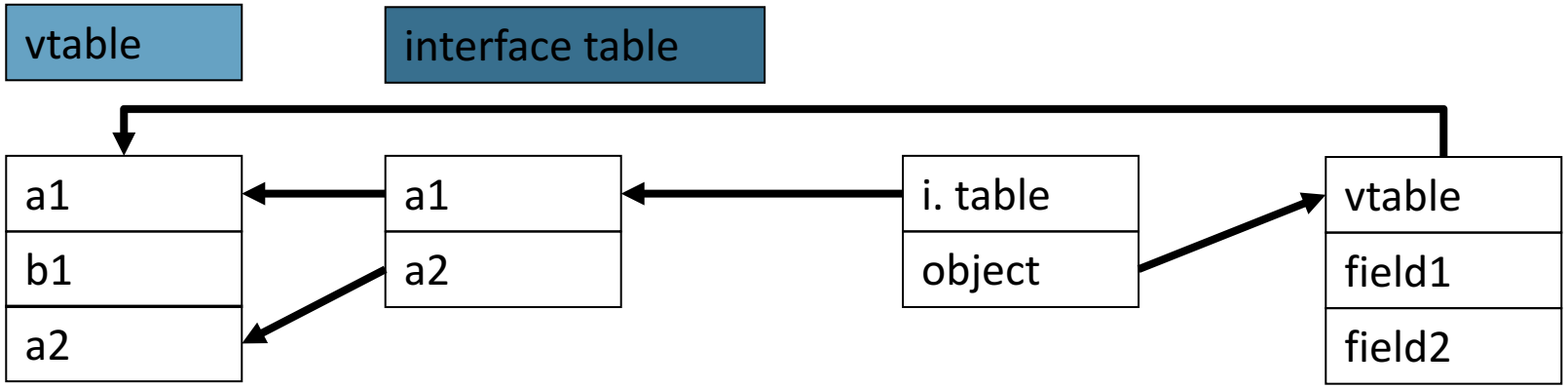
- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields

```
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



Dynamic Class Loading

- Supported by some OO languages (Java)
- At compile time
  - the actual class of a given object at a given program point may not be known
- Some addresses have to be resolved at runtime
  - problem when multiple inheritance is allowed
    - Can use hash table to map fields name to pointers to functions

Other OO Features

- Information hiding
  - private/public/protected fields
  - Semantic analysis (context handling)

- Testing class membership
  - Single inheritance: look up the chain of “supers”
  - If inheritance is bounded
    - Allocate parent array at every class descriptors
    - look up at “depth-of-inheritance” index

- Up casting: (Possibly) runtime checks
  - B x = (B)a // B extends A

Optimizing OO languages

- Hide additional costs
  - Eliminate runtime checks

    A x = new B() // B extends A

    B y = (B) x
  - Eliminate dead fields

  - Replace dynamic by static binding when possible

    x = new B() // B extends A and overrides f

    x.f() // invoke B_f(x)
    - Type propagation analysis ~ reaching definitions

Optimizing OO languages

- Simultaneously generate code for multiple classes

```
class A {
  g() { }
  f() { g() }
}
```

```
class B extends A {
  g() {}
}
```

```
class B extends A {
  g()
}
```

- Generate code B_f() and C_f()
- Code space is an issue

Summary

- OO is a programming/design paradigm
- OO features complicates compilation
  - Semantic analysis
  - Code generation
  - Runtime
  - Memory management
- Understanding compilation of OO can be useful for programmers

Compilation

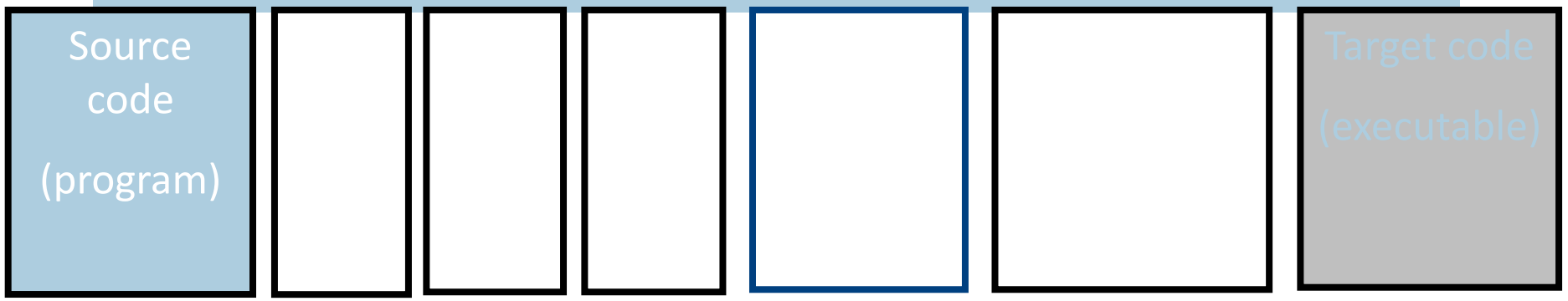
0368-3133 2016/17a

Lecture 11b

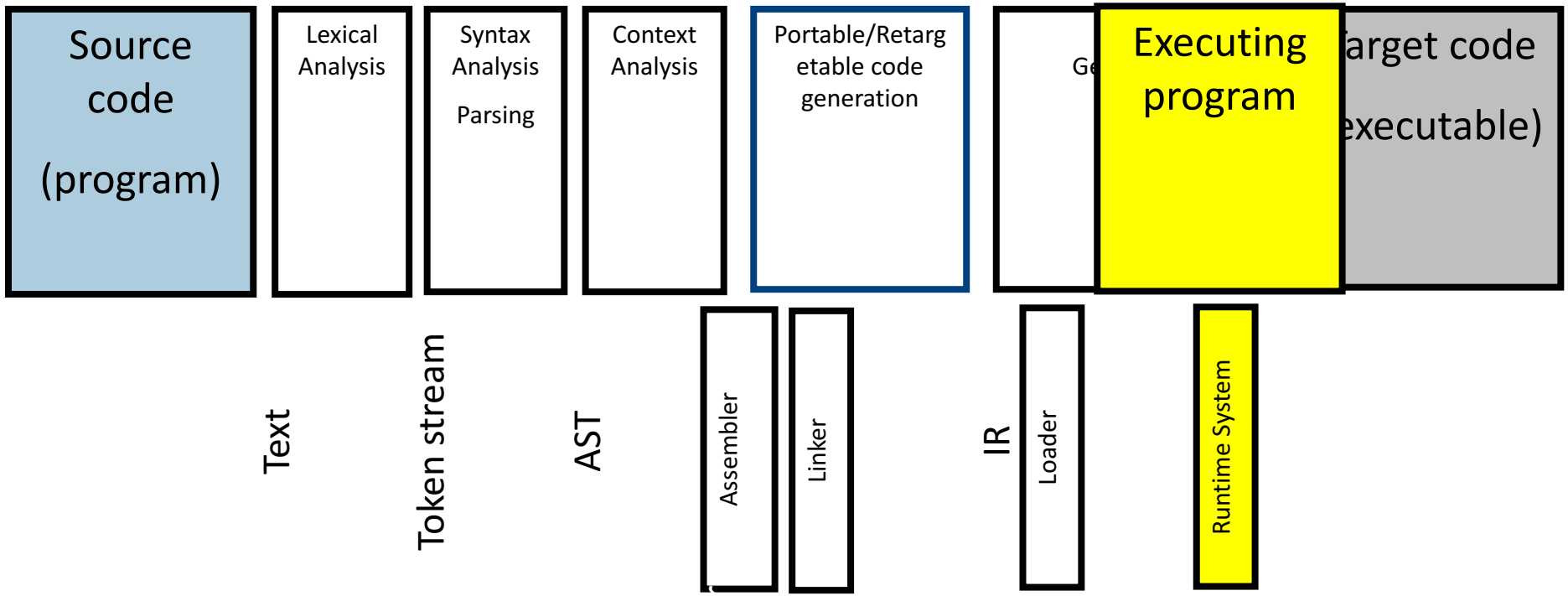
Memory Management

Noam Rinetzky

Stages of compilation



Compilation ➔ Execution



Runtime Environment

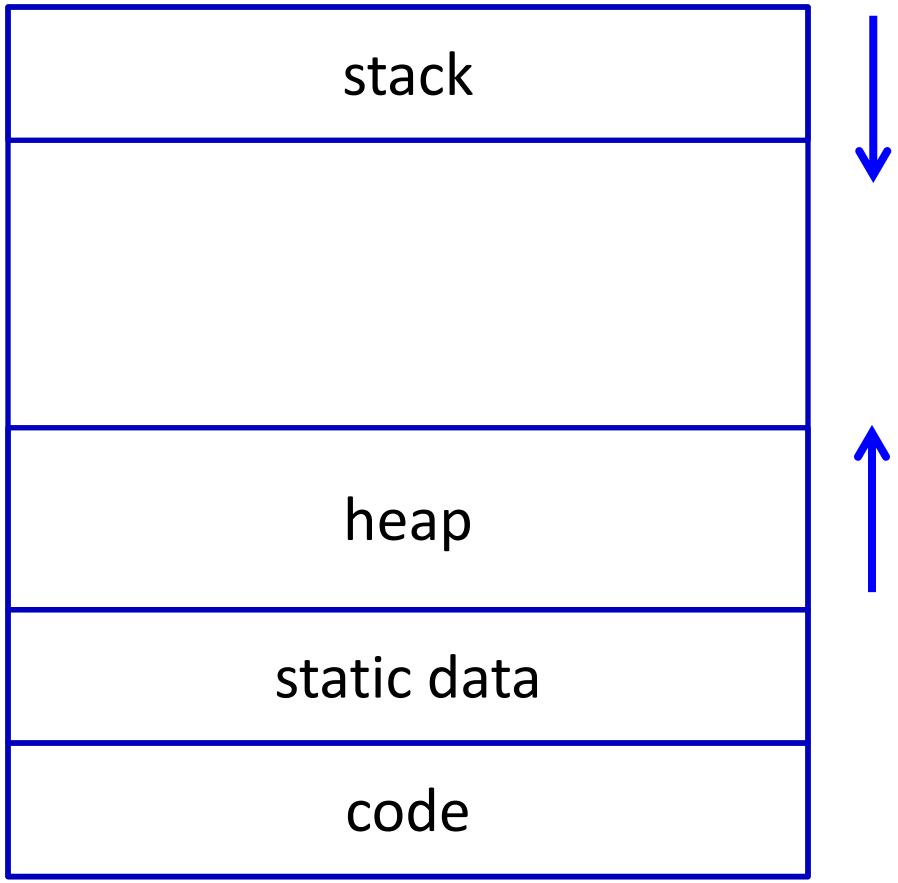
- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Dynamic optimization
  - Debugging
  - …

Where do we allocate data?

- Activation records
  - Lifetime of allocated data limited by procedure lifetime
  - Stack frame deallocated (popped) when procedure return

- **Dynamic memory allocation on the heap**

Memory Layout



Alignment

- Typically, can only access memory at aligned addresses
  - Either 4-bytes or 8-bytes
- What happens if you allocate data of size 5 bytes?
  - **Padding** – the space until the next aligned addresses is kept empty
- (side note: x86, is more complicated, as usual, and also allows unaligned accesses, but not recommended)

Allocating memory

- In C - malloc
- `void *malloc(size_t size)`

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its type
- How does malloc guarantee alignment?
  - After all, you don't know what type it is allocating for
  - It has to align for the largest primitive type
  - In practice optimized for 8 byte alignment (glibc-2.17)

Memory Management

- Manual memory management
- Automatic memory management

Manual memory management

- malloc
- free

```
a =  malloc(…) ;
// do something with a
free(a);
```

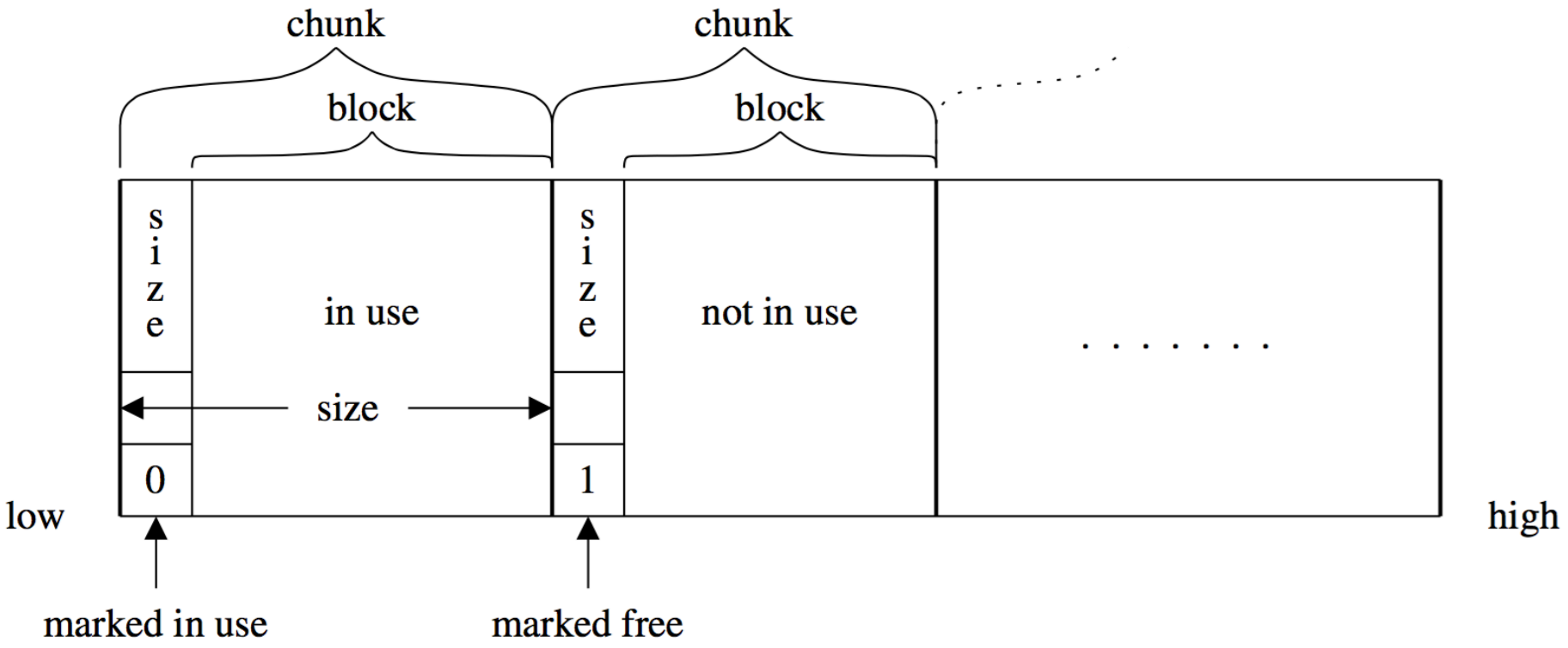
malloc

- where is malloc implemented?
- how does it work?

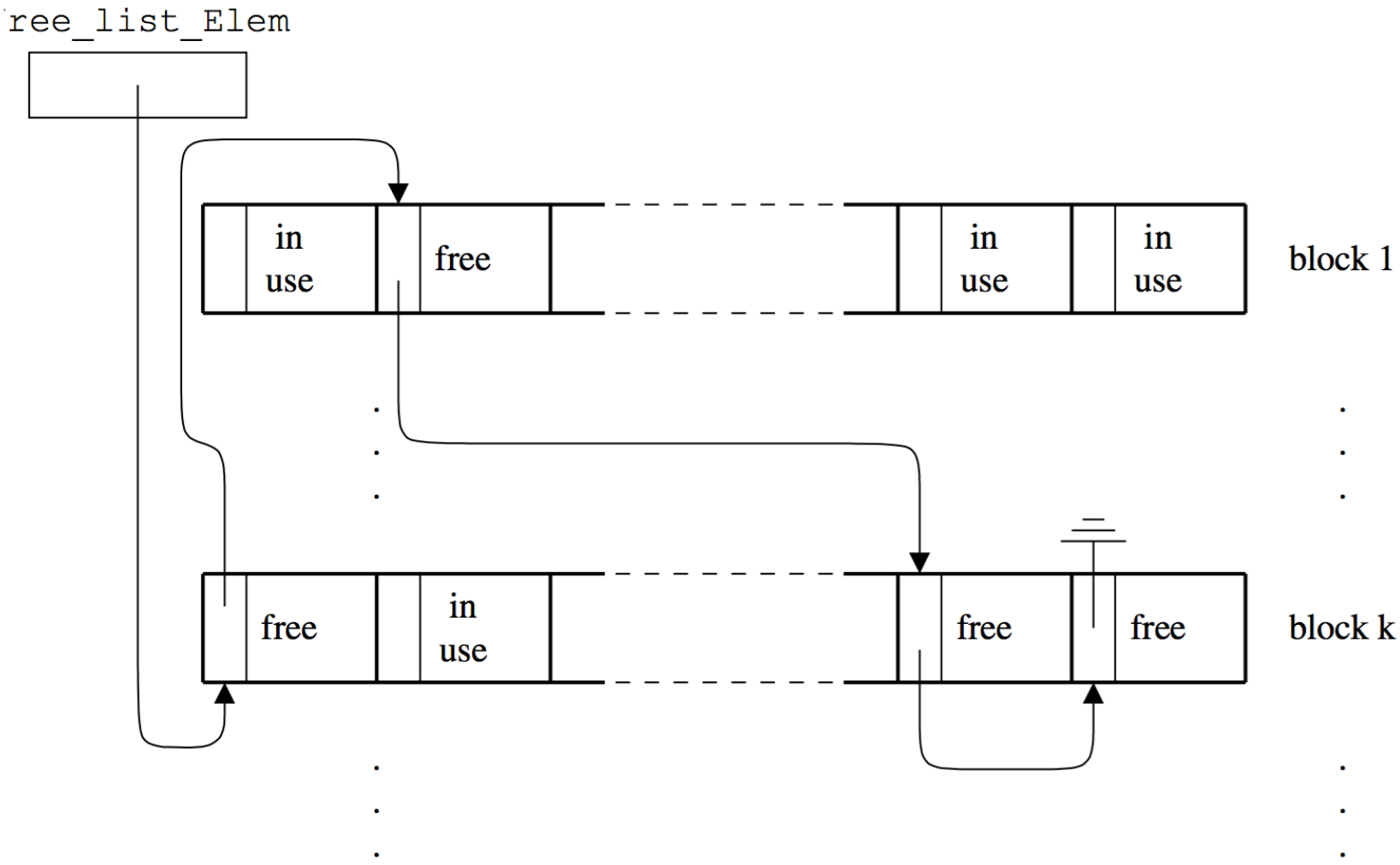
Free-list Allocation

- 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

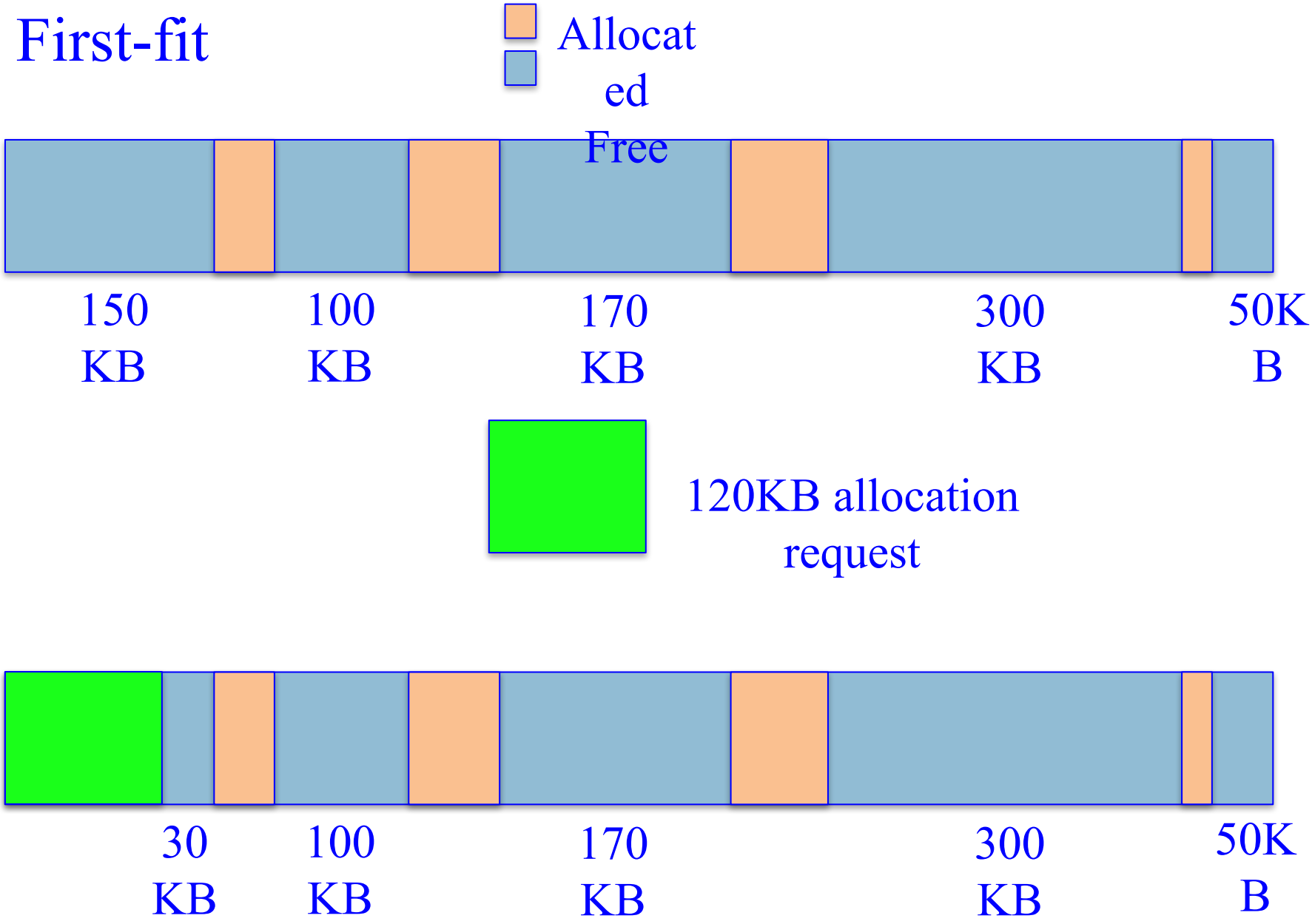
Memory chunks

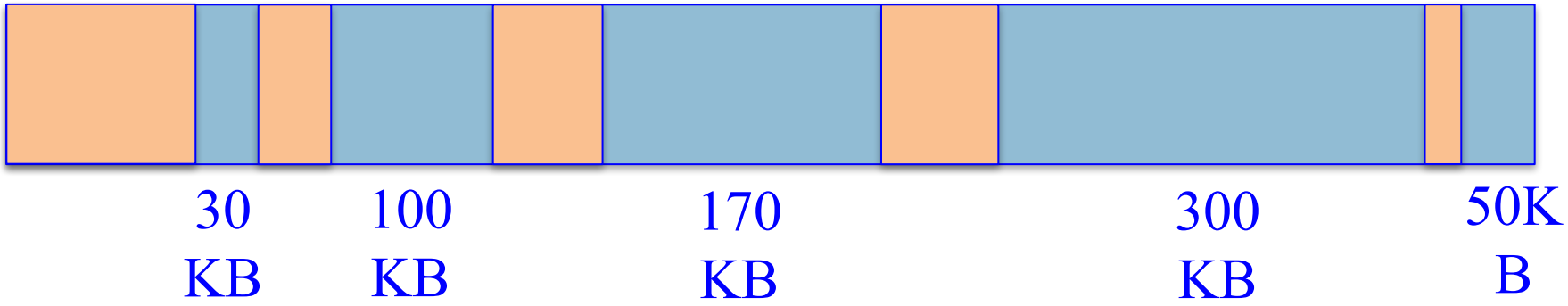


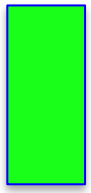
Free list


ree_list_Elem
in use
free
in use
in use
block 1
free
in use
free
free
block k

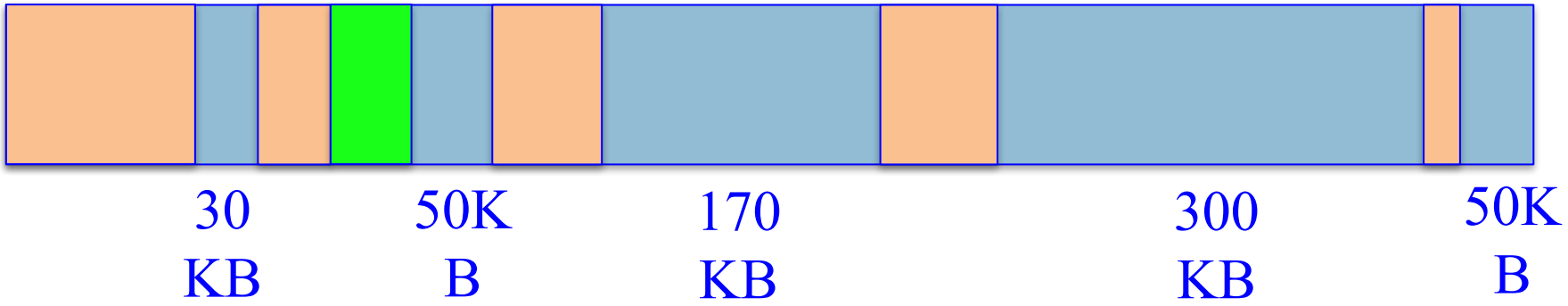
First-fit

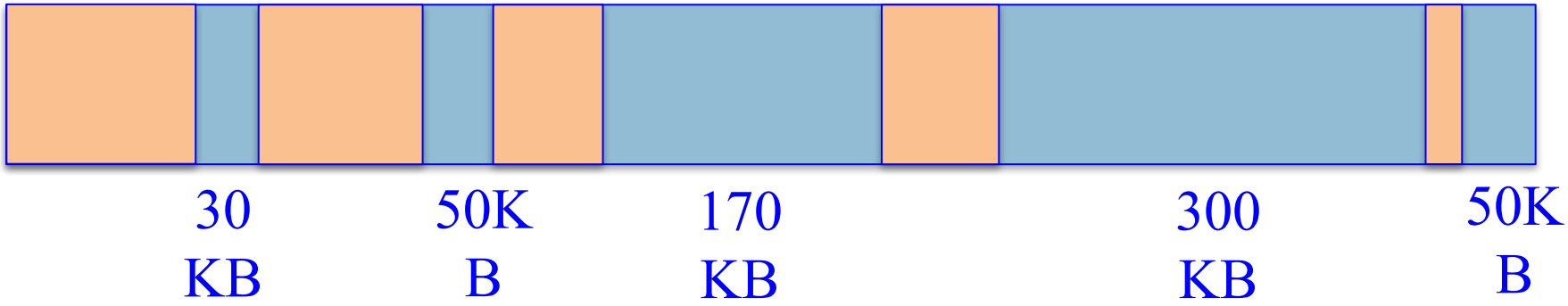

Allocated
Free
150 KB
100 KB
170 KB
300 KB
50KB
120KB allocation request
30 KB
100 KB
170 KB
300 KB
50KB

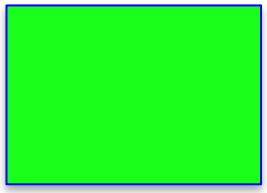




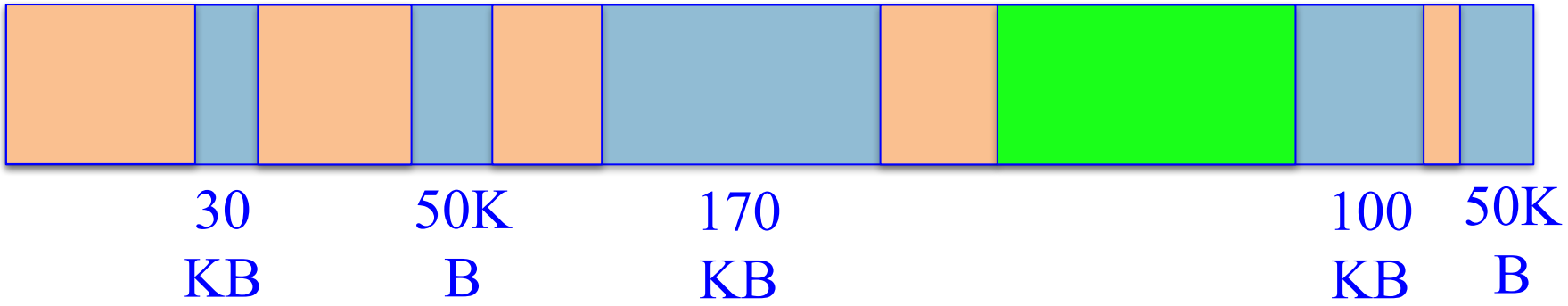
50KB allocation
request







200KB allocation
request



Fragmentation

- Dispersal of free memory across a possibly large number of small free cells.
- Negative effects:
  - Can prevent allocation from succeeding
  - May cause a program to use more address space, more resident pages and more cache lines.
- Fragmentation is impractical to avoid:
  - Usually the allocator cannot know what the future request sequence will be.
  - Even given a known request sequence, doing an optimal allocation is NP-hard.
- Usually There is a trade-off between allocation speed and fragmentation.

Segregated-fits Allocation

- Idea – use multiple free-list whose members are segregated by size in order to speed allocation.
- Usually a fixed number k of size values $s_0 < s_1 < \ldots < s_{k-1}$
- $k+1$ free lists $f_0,\ldots,f_k$
- For a free cell, b, on list f_i,

$$\text{size}(b) = s_i \;\; \forall \; 1 \leq i \; \leq k-1$$

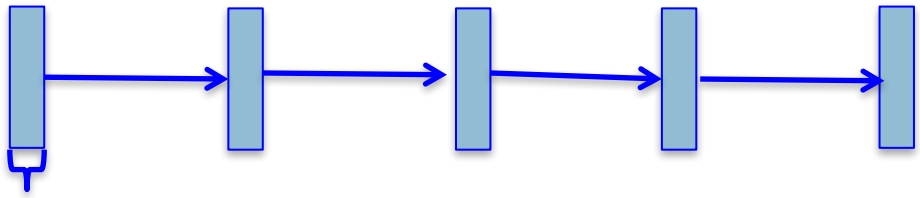
$$\text{size}(b) > s_{k-1} \text{ if } i=k$$

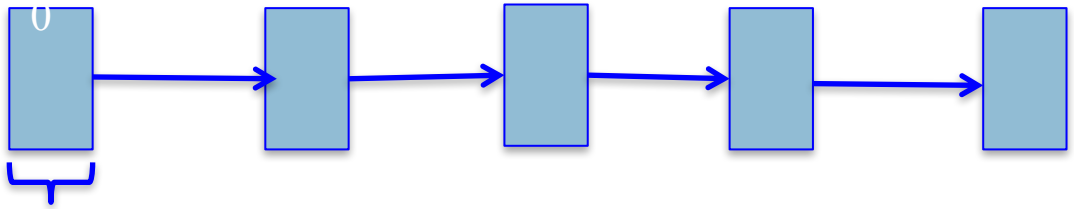
- When requesting a cell of size $b \leq s_{k-1}$, the allocator rounds the request size up to the smallest s_i such that $b \leq s_i$.
- S_i is called a **size class**

Segregated-fits Allocation

```
SegregatedFitAllocate(j):
        result ← remove(freeLists[j])
        if result = null
                large ← allocateBlock()
                if large = null
                        return null
                initialize(large, sizes[j])
                result ← remove(freeList[j])
        return result
```

- List f_k, for cells larger than s_k, is organized to use one of the basic single-list algorithm.
- Per-cell overheads for large cell are a bit higher but in total it is negligible.
- The main advantage: for size classes other than s_k, allocation typically requires constant time.




0


1

Runtime support for MM

- C's standard library provides basic memory management
  - Gets memory pages from the OS
  - Maintains inventory of free memory cells
    - mmap(), brk()

free

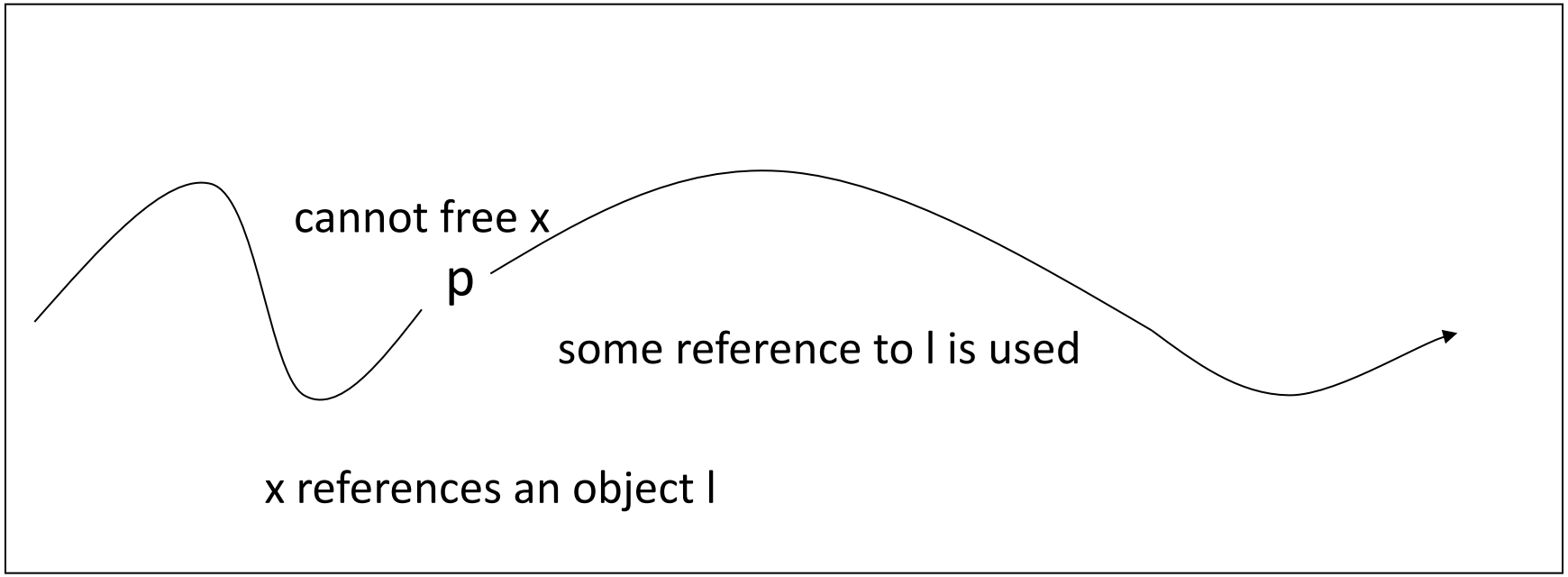
- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

When can we free an object?

```
a =  malloc(…) ;
b = a;
// free (a); ?
c = malloc (…);
if  (b == c)
     printf("unexpected equality");
```

Cannot free an object if it has a reference with a future use!

When can free x be inserted after p?



On all execution paths after p there are no uses of references to the object referenced by x ➔ inserting free x after p is valid

Automatic Memory Management

- automatically free memory when it is no longer needed
- not limited to OO languages
- prevalent in OO languages such as Java
  - also in functional languages

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)

Managing Reference Counts

- Each object has a reference count o.RC
- A newly allocated object o gets o.RC = 1
  - why?

- write-barrier for reference updates

```
update(x,old,new) {
   old.RC--;
   new.RC++;
   if (old.RC == 0) collect(old);
}
```

- collect(old) will decrement RC for all children and recursively collect objects whose RC reached 0.

Cycles!

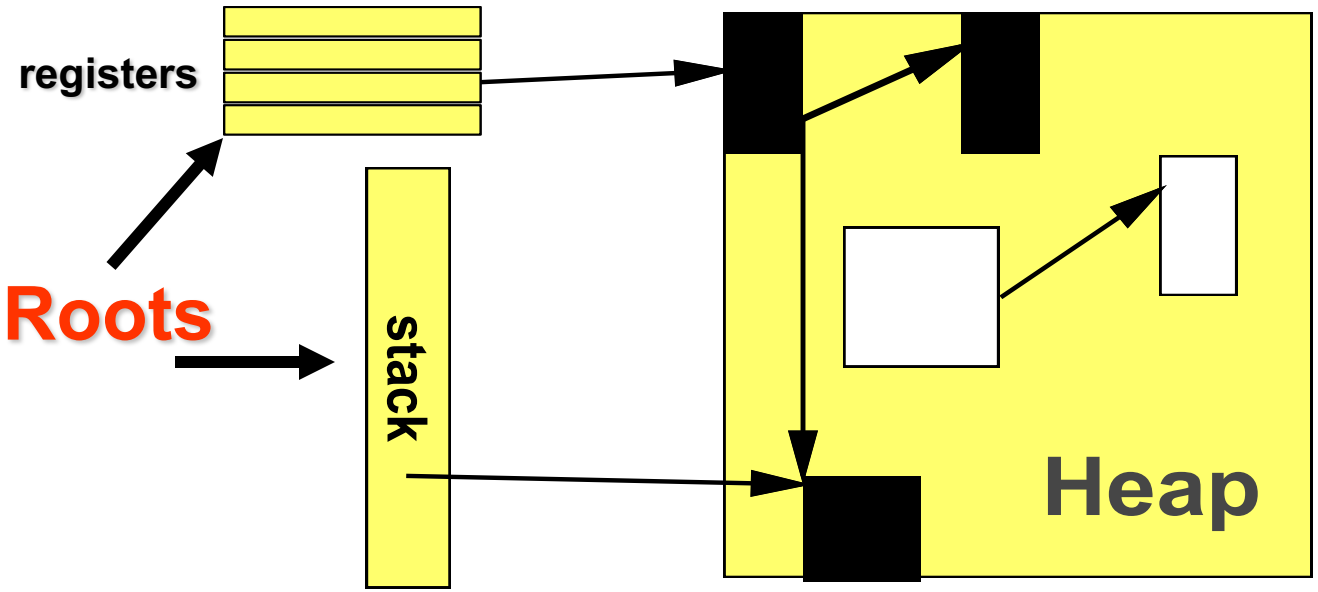
- cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - periodically invoke a tracing algorithm to collect cycles
  - specialized algorithms for collecting cycles

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

The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.



heap

Triggering

```
New(A)=
   if free_list is empty
            mark_sweep()
             if free_list is empty
                        return (“out-of-memory”)
   pointer = allocate(A)
   return (pointer)
```

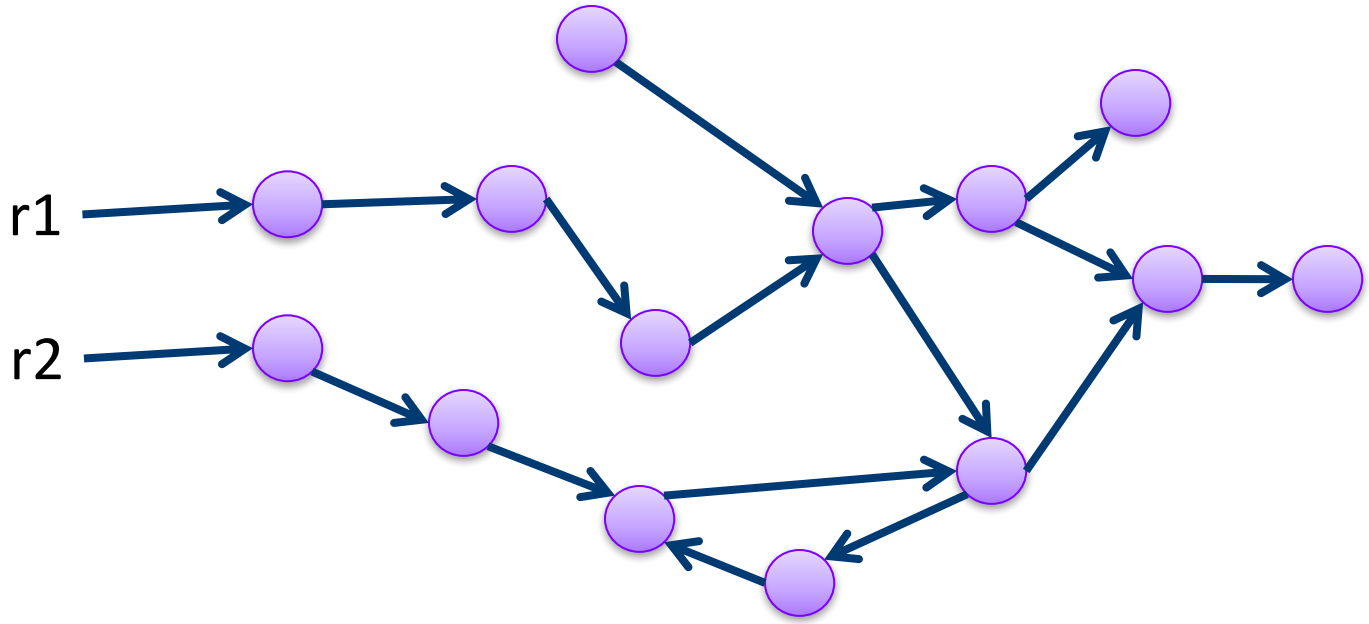
Basic Algorithm

```
mark_sweep()=
for Ptr in Roots
        mark(Ptr)
sweep()
```

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

```
Sweep()=
p = Heap_bottom
while (p < Heap_top)
        if (mark_bit(p) == unmarked) then free(p)
        else mark_bit(p) = unmarked;
        p=p+size(p)
```

Mark&Sweep Example



Mark&Sweep in Depth

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

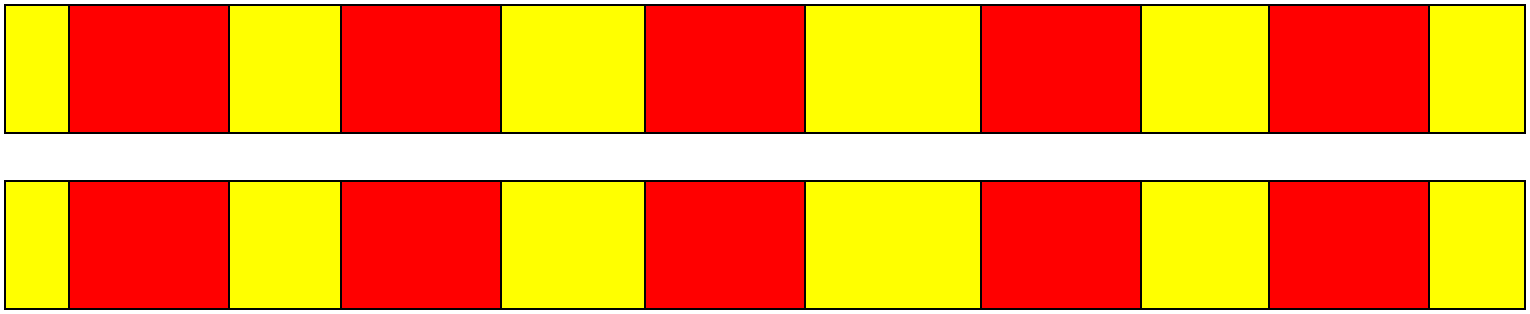
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Properties of Mark & Sweep

- Most popular method today
- Simple
- Does not move objects, and so **heap may fragment**
- Complexity
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size
- Termination: each pointer traversed once
- Engineering tricks used to improve performance

Mark-Compact

- During the run objects are allocated and reclaimed
- Gradually, the heap gets fragmented
- When space is too fragmented to allocate, a compaction algorithm is used
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations
- Compaction is very costly and we attempt to run it infrequently, or only partially



Mark Compact

- Important parameters of a compaction algorithm
  - Keep order of objects?
  - Use extra space for compactor data structures?
  - How many heap passes?
  - Can it run in parallel on a multi-processor?

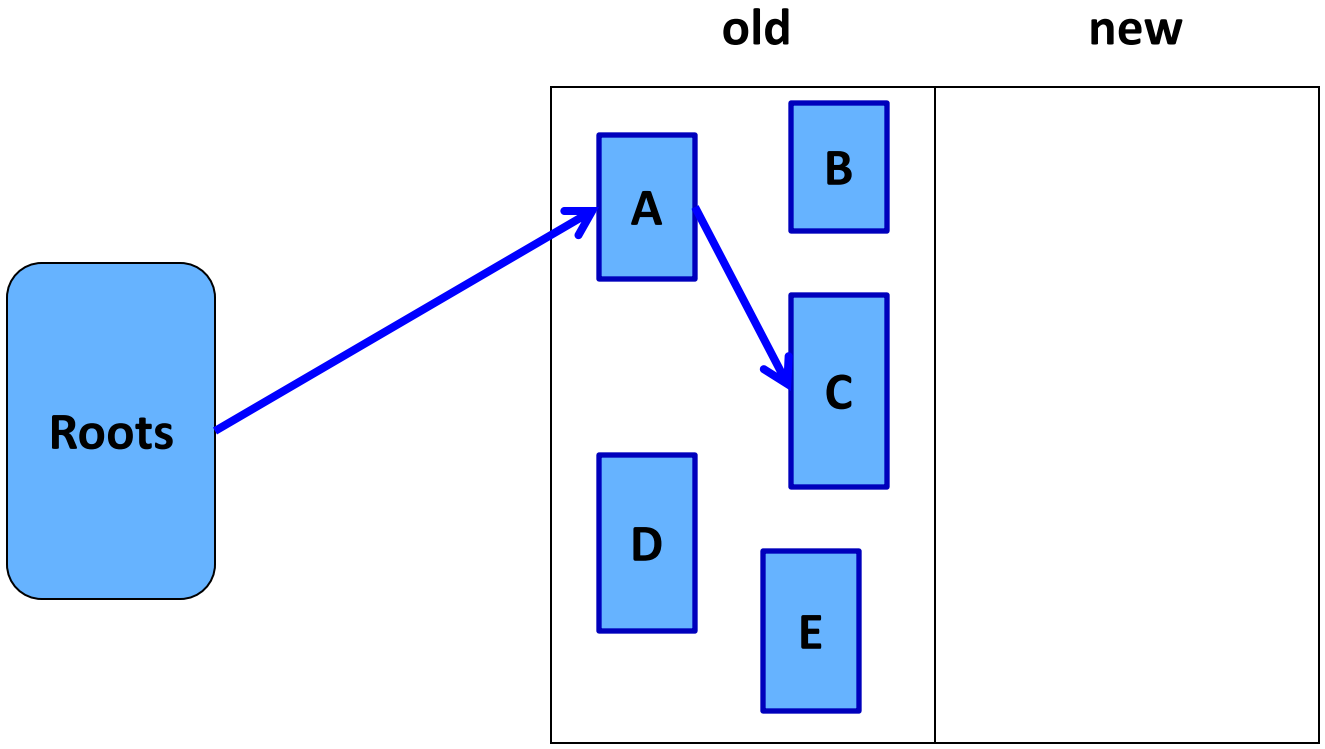
- We do not elaborate in this intro

Copying GC

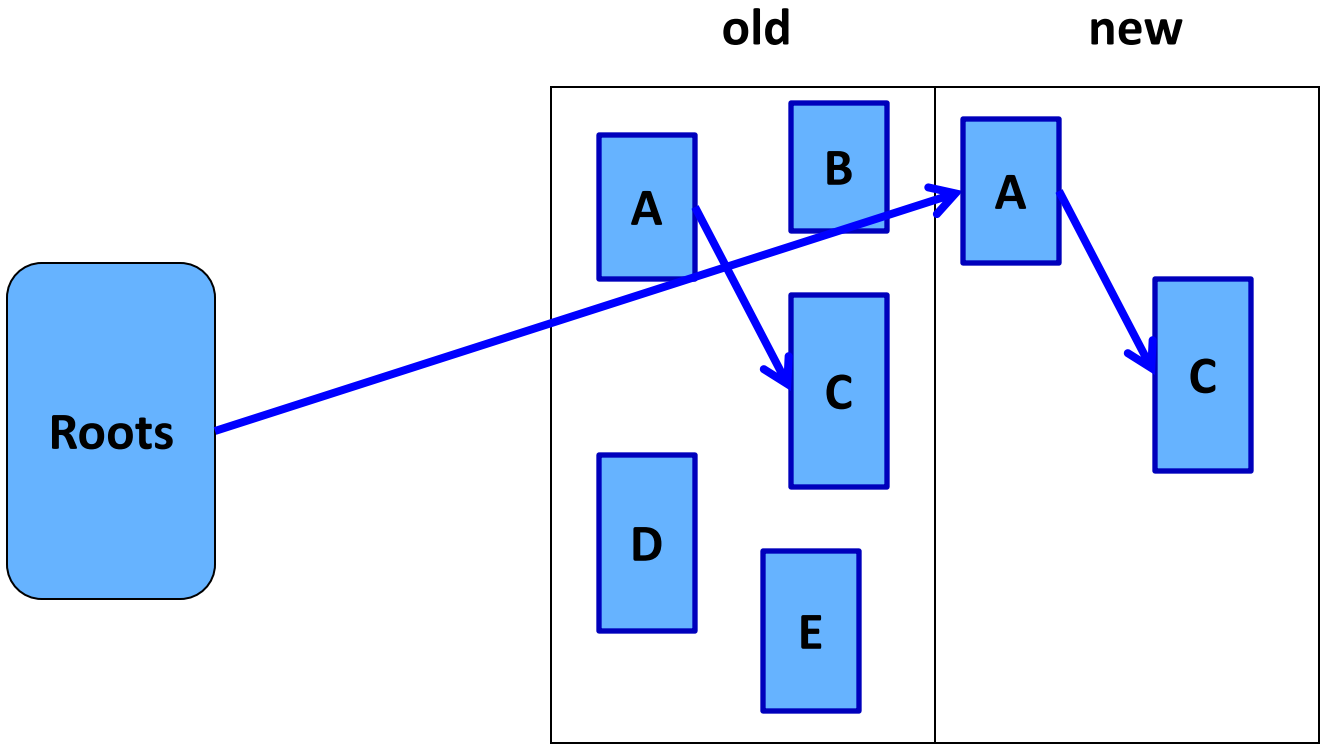
- partition the heap into two parts
  - old space
  - new space

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

Example



Example



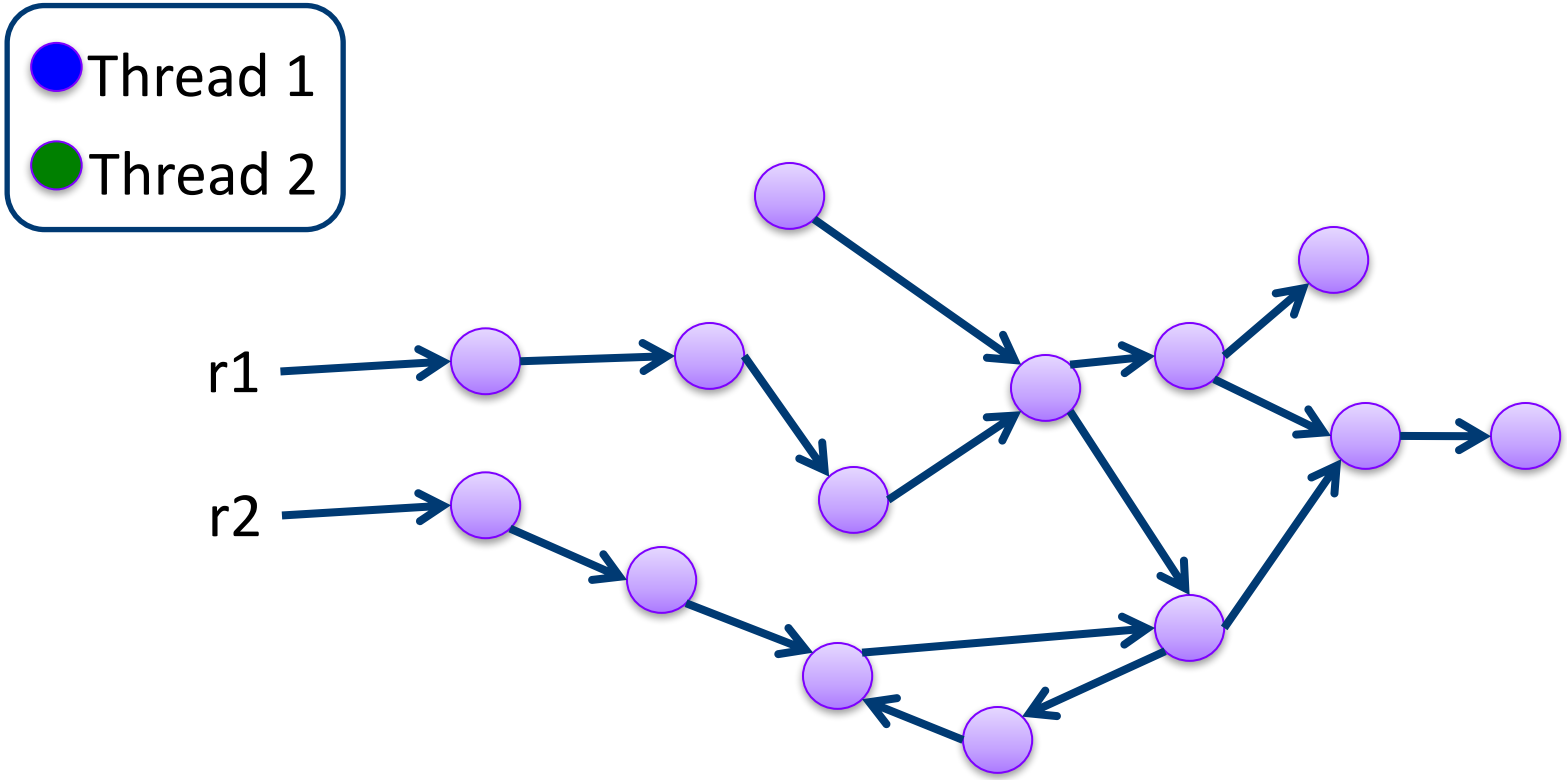
Properties of Copying Collection

- Compaction for free
- Major disadvantage: **half of the heap is not used**
- "Touch" only the live objects
  - Good when most objects are dead
  - Usually most new objects are dead
    - Some methods use a small space for young objects and collect this space using copying garbage collection

A very simplistic comparison

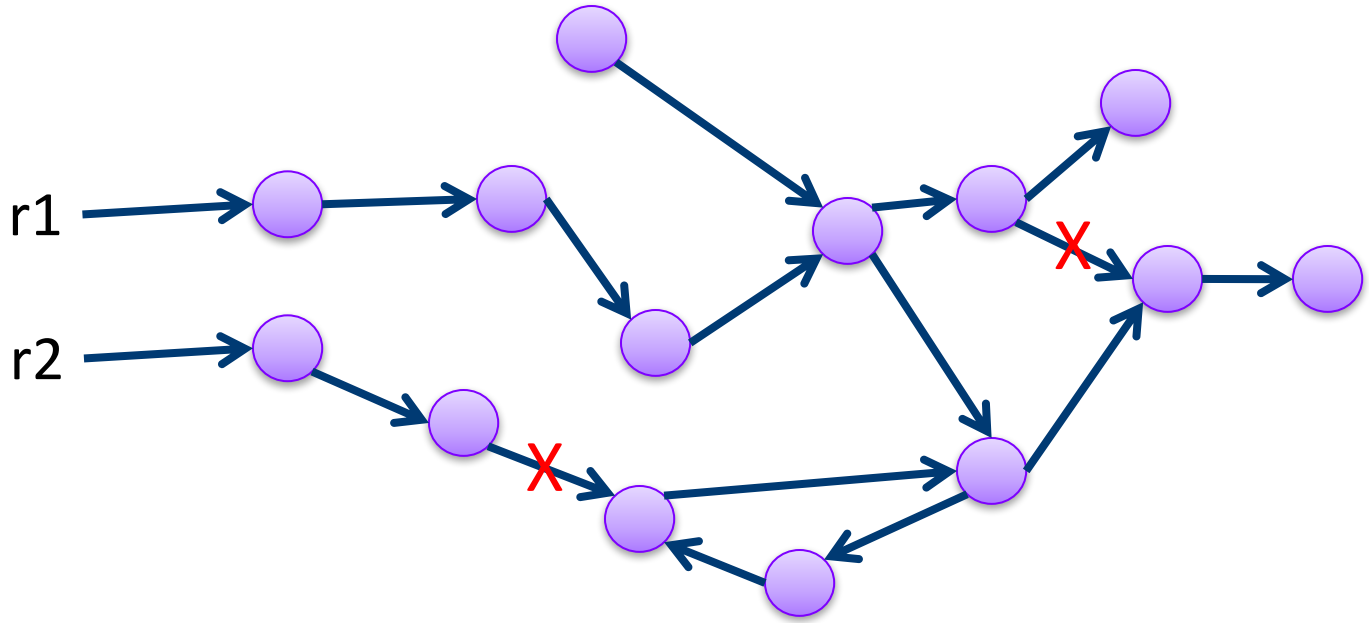
	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Mark = live objects Sweep = Size of heap	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

Parallel Mark&Sweep GC



Parallel GC: mutator is stopped, GC threads run in parallel

Concurrent Mark&Sweep Example



Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator

Conservative GC

- How do you track pointers in languages such as C ?
  - Any value can be cast down to a pointer
- **How can you follow pointers in a structure?**

- Easy – be conservative, consider anything that can be a pointer to be a pointer
- Practical! (e.g., Boehm collector)

Conservative GC

- Can you implement a conservative **copying GC**?
- What is the problem?
- **Cannot update pointers to the new address… you don't know whether the value is a pointer, cannot update it**

Modern Memory Management

- Considers standard program properties
- Handle parallelism
  - Stop the program and collect in parallel on all available processors
  - Run collection concurrently with the program run
- Cache consciousness
- Real-time

Terminology Recap

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, compaction, tracing algorithms
- Fragmentation