Binding and Storage

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

What's the most striking difference?

```java
/**
 * A simple interface to describe finite automata. {@link GraphvizFormatter} uses
 * this to visualize arbitrary finite automata.
 */
public interface Automaton {
    /**
     * @return Something that can be iterated to visit all states.
     */
    public Iterable<State> getStates();

    /**
     * @return the start state (which must be, by definition, unique)
     */
    public State getStartState();

    /**
     * @return Something that can be iterated to visit all final states.
     */
    public Iterable<State> getFinalStates();

    /**
     * @return Return the regular expression that is equivalent to this
     *         automaton.
     */
    public String getRegex();
}
```

Java Interface definition

```
/usr/bin/java:
(__TEXT,__text) section
00001c74    pushl   $0x00
00001c76    movl    %esp,%ebp
00001c78    andl    $0xf0,%esp
00001c7b    subl    $0x10,%esp
00001c7e    movl    0x04(%ebp),%ebx
00001c81    movl    %ebx,(%esp)
00001c84    leal    0x08(%ebp),%ecx
00001c87    movl    %ecx,0x04(%esp)
00001c8b    addl    $0x01,%ebx
00001c8e    shll    $0x02,%ebx
00001c91    addl    %ecx,%ebx
00001c93    movl    %ebx,0x08(%esp)
00001c97    movl    (%ebx),%eax
00001c99    addl    $0x04,%ebx
00001c9c    testl   %eax,%eax
00001c9e    jne     0x00001c97
00001ca0    movl    %ebx,0x0c(%esp)
00001ca4    calll   0x00001cb2
00001ca9    movl    %eax,(%esp)
00001cac    calll   0x00002a42
00001cb1    hlt
00001cb2    pushl   %ebp
00001cb3    movl    %esp,%ebp
00001cb5    pushl   %edi
00001cb6    pushl   %esi
00001cb7    pushl   %ebx
```

x86 Assembly

What's the most striking difference?

```java
/**
 * A simple interface to describe finite automata. {@link GraphvizFormatter} uses
 * this to visualize arbitrary finite automata.
 */
public interface Automaton {
    /**
     * @return Something that can be iterated to visit all states.
     */
    public Iterable<State> getStates();

    /**
     * @return the start s  te (which must be, by definition, unique)
     */
    public State getStartS  te();

    /**
     * @return Something th  t can be iterated to visit all final states.
     */
    public Iterable<State>   tFinalStates();

    /**
     * @return Return the r  ular expression that is equivalent to this
     *         automaton.
     */
    public String getRegex()
}
```

Java Interface definition

```
/usr/bin/java:
(__TEXT,__text) section
00001c74    pushl   $0x00
00001c76    movl    %esp,%ebp
00001c78    andl    $0xf0,%esp
00001c7b    subl    $0x10,%esp
00001c7e    movl    0x04(%ebp),%ebx
00001c81    movl    %ebx,(%esp)
00001c84    leal    0x08(%ebp),%ecx
00001c87    movl    %ecx,0x04(%esp)
00001c8b    addl    $0x01,%ebx
00001c8e    shll    $0x02,%ebx
00001c91    addl    %ecx,%ebx
00001c93    movl    %ebx,0x08(%esp)
00001c97    movl    (%ebx),%eax
00001c99    addl    $0x04,%ebx
00001c9c    testl   %eax,%eax
00001c9e    jne     0x00001c97
00001ca0    mo      %ebx,0x0c(%esp)
00001ca4            0x00001cb2
00001ca9    movl    %eax,(%esp)
00001cac    calll   0x00002a42
0000  b1    hlt
0   1cb2    pushl   %ebp
    1cb3    movl    %esp,%ebp
    1cb5    pushl   %edi
    1cb6    pushl   %esi
    1cb7    pushl   %ebx
```

x86 Assembly

Names!

High-level languages have rich facilities for naming "**things**."

Tuesday, February 23, 2010

Names

Required for abstraction.

- Assembly only has **values** & **addresses** & registers.
  - Machine dependence!
- **Names enable abstraction**.
  - Can refer to something **without knowing the details** (e.g., exact address, exact memory layout).
  - Let the compiler worry about the details.
- Can refer to things that **do not yet exist**!
  - E.g., during development, we can (and often do) write code for (Java) interfaces that have not yet been implemented.

Abstraction

Control Abstraction vs. Data Abstraction

```java
/**
 * A simple interface to describe finite automata. {@link GraphvizFormatter} uses
 * this to visualize arbitrary finite automata.
 */
public interface Automaton {
    /**
     * @return Something that can be iterated to visit all states.
     */
    public Iterable<State> getStates();

    /**
     * @return the start state (which must be, by definition, unique)
     */
    public State getStartState();

    /**
     * @return Something that can be iterated to visit all final states.
     */
    public Iterable<State> getFinalStates();

    /**
     * @return Return the regular expression that is equivalent to this
     *         automaton.
     */
    public String getRegex();
}
```

Abstraction

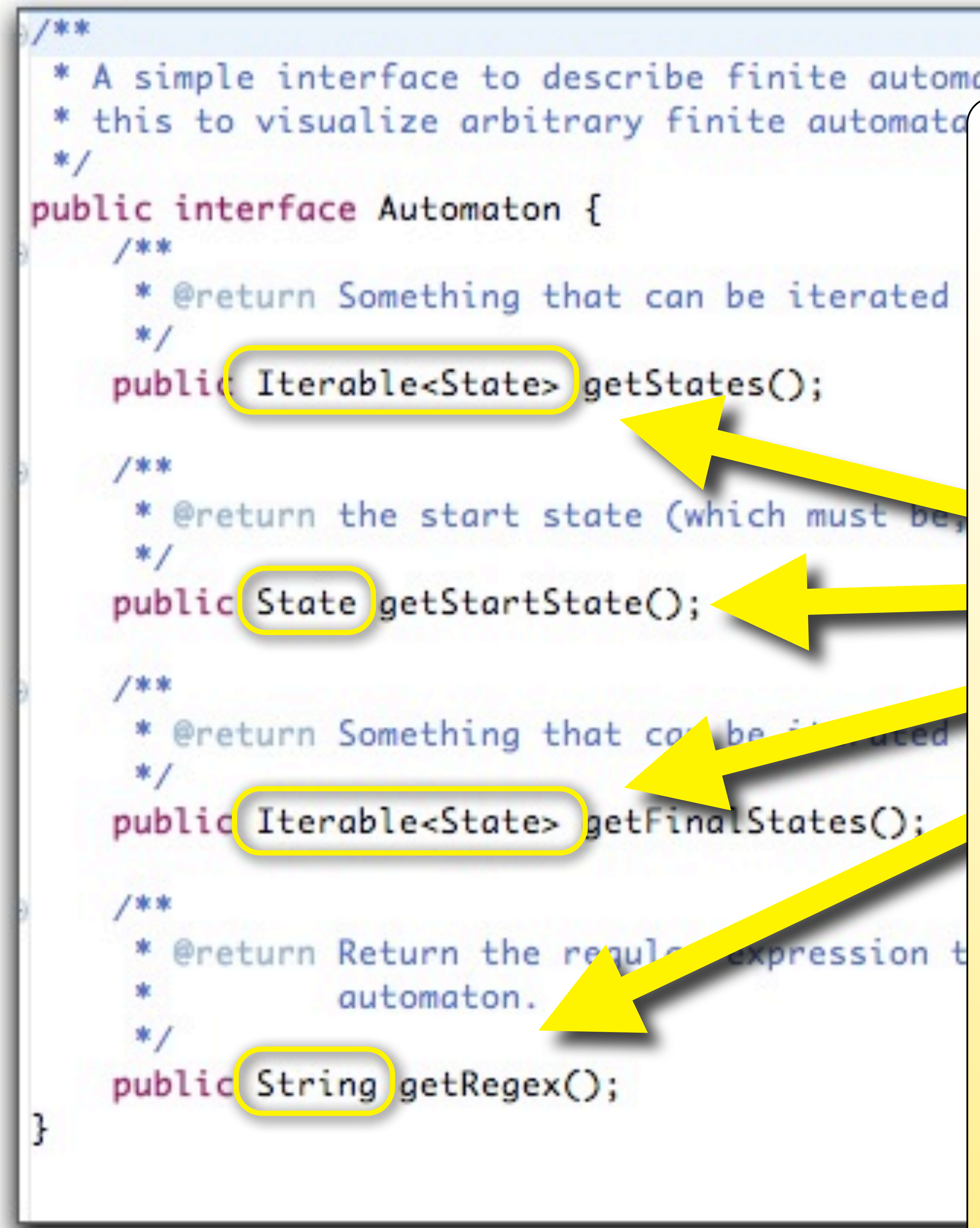
Control Abstraction

Can **hide arbitrary complex code** behind a simple name.
For example, addition can be simple (int) or “difficult” (vector).

```
public interface Automaton {
    /**
     * @return Something that can be iterated to visit all states.
     */
    public Iterable<State> getStates();

    /**
     * @return the start state (which must be, by definition, unique)
     */
    public State getStartState();

    /**
     * @return Something that can be iterated to visit all final states.
     */
    public Iterable<State> getFinalStates();

    /**
     * @return Return the regular expression that is equivalent to this
     *         automaton.
     */
    public String getRegex();
}
```

Abstraction

Control Abstraction vs. Data Abstraction



Data Abstraction

Abstract Data Types (ADTs)

Reason about **concepts** **instead of impl. details**.

Programmer doesn't know memory layout, address, whether other interfaces are implemented, what invariants need to be ensured, etc.

Binding

Associating a **name** with some **entity**.
(or “object,” but not the Java notion of an object)

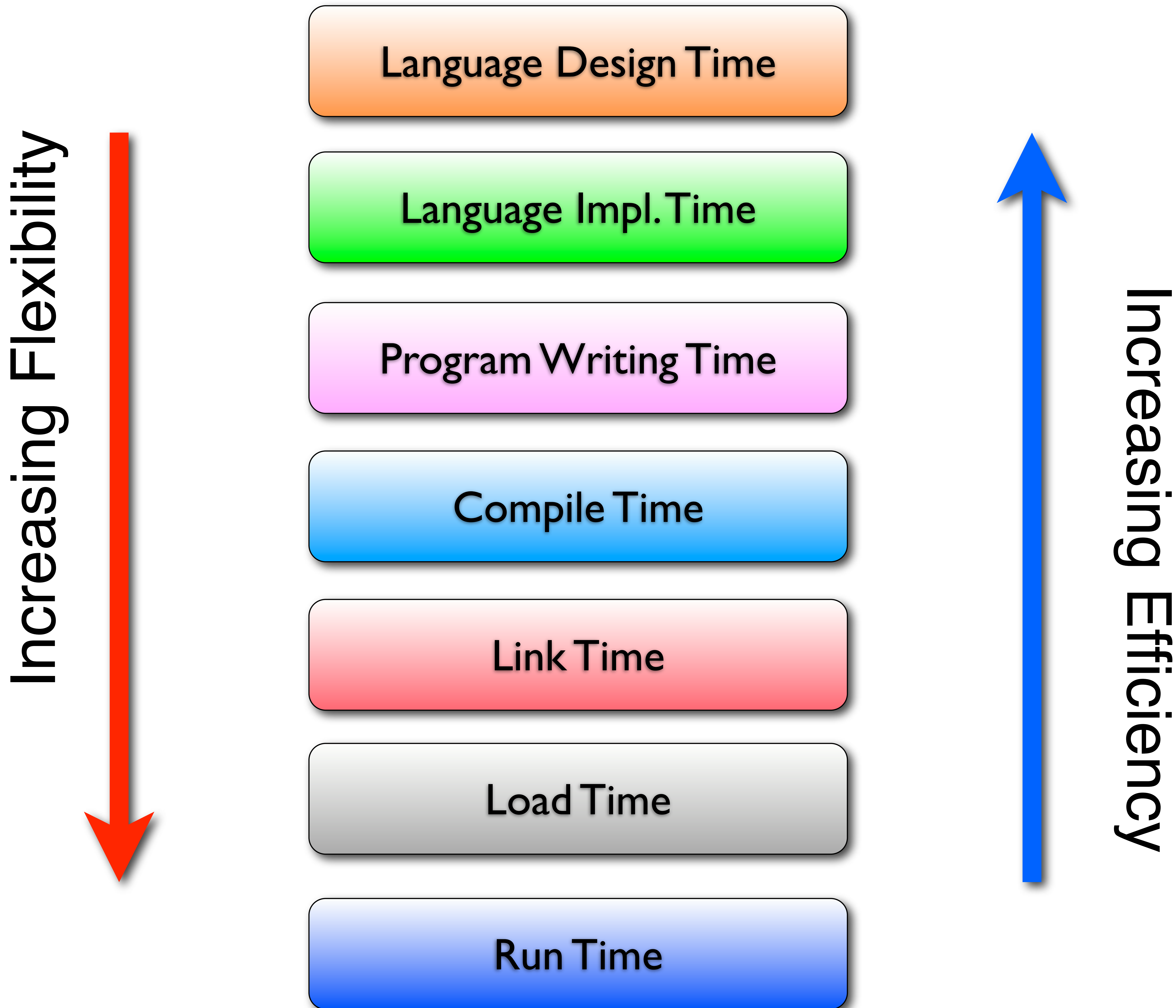
Binding vs. Abstraction.

- ➡ Introducing **a name creates an abstraction**.
- ➡ **Binding** a name to an entity **resolves an abstraction**.

Binding time:

- ➡ When is a name resolved?

Binding Time



Binding Time

Keywords, e.g. “if”	Language Design Time
Tool vendor chooses **predefined symbols** and std. library impl.	Language Impl. Time
Programmer chooses names, data structures, and algorithms.	Program Writing Time
Compiler determines exact **memory layout**, code order, etc.	Compile Time
Resolve names in **static library**, e.g., “printf” in the C library.	Link Time
OS loads **dynamic libraries**, e.g., Windows loads DLLs, Unix SOs.	Load Time
Name resolved **based on input**. E.g., plugins, interfaces.	Run Time

Increasing Efficiency

Binding Time

Binding may change…

Keywords, e.g. "if"	Language Design Time	if language is revised, e.g., Java 1.1, 1.2, …, 1.6
Tool vendor chooses **predefined symbols** and std. library impl.	Language Impl. Time	if new tool version is released, e.g., GCC 2.95 vs. GCC 4.0.
Programmer chooses names, data structures, and algorithms.	Program Writing Time	if source code is changed.
Compiler determines exact **memory layout**, code order, etc.	Compile Time	on every compile (but hopefully doesn't).
Resolve names in **static library**, e.g., "printf" in the C library.	Link Time	every time the final program is linked after a **library update**.
OS loads **dynamic libraries**, e.g., Windows loads DLLs, Unix SOs.	Load Time	every time the app is launched. (e.g., Windows compatibility mode, DLL injection, debug libraries).
Name resolved **based on input**. E.g., plugins, interfaces.	Run Time	at any time during execution.

Binding Time

Called **static** or **early** binding.

Increasing Flexibili

Language Design Time

Language Impl. Time

Program Writing Time

Compile Time

Link Time

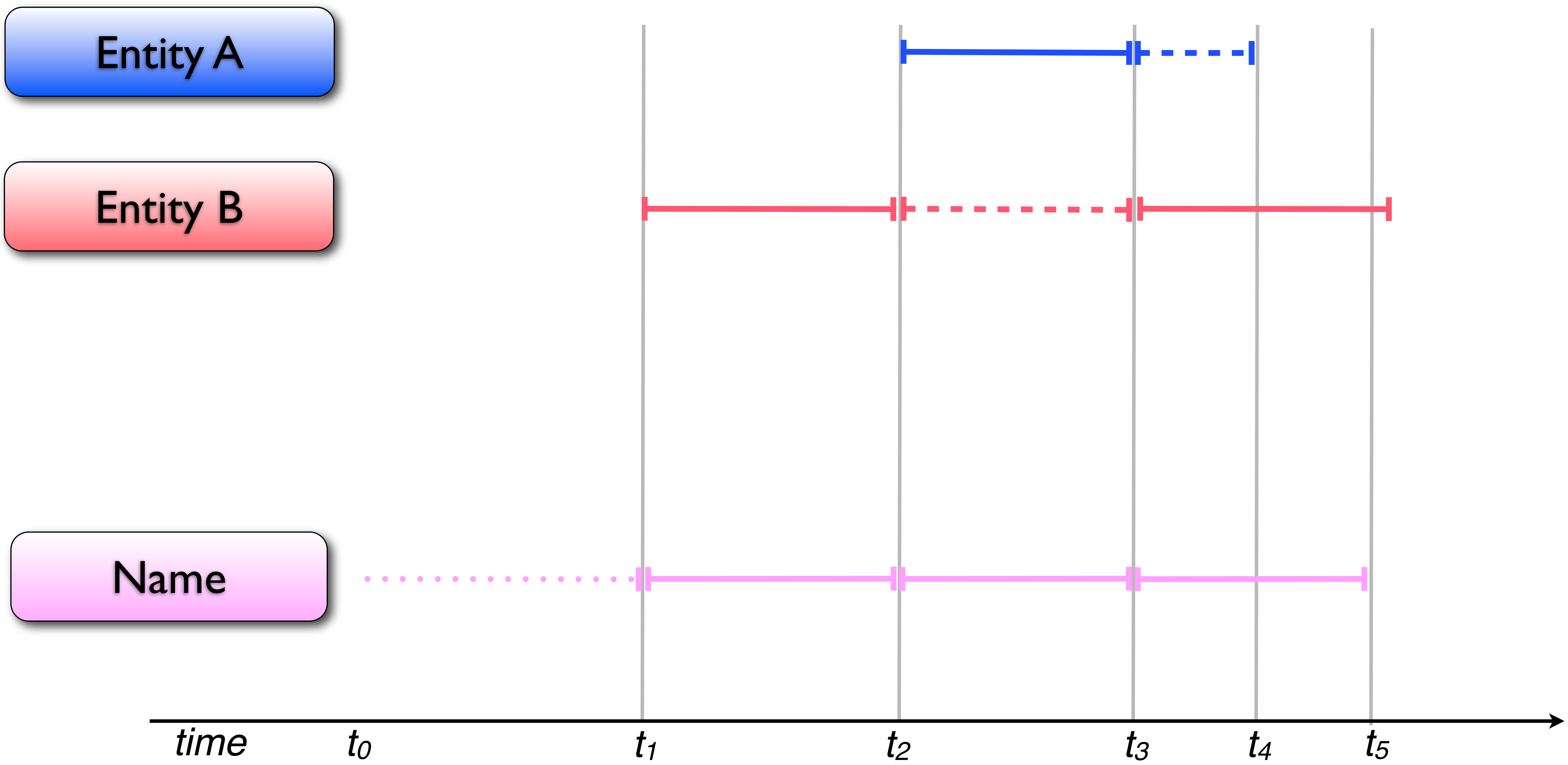
Load Time

Run Time

Increasing Efficienc

Called **dynamic** or **late** binding.

Object Lifetime vs. Binding Lifetime

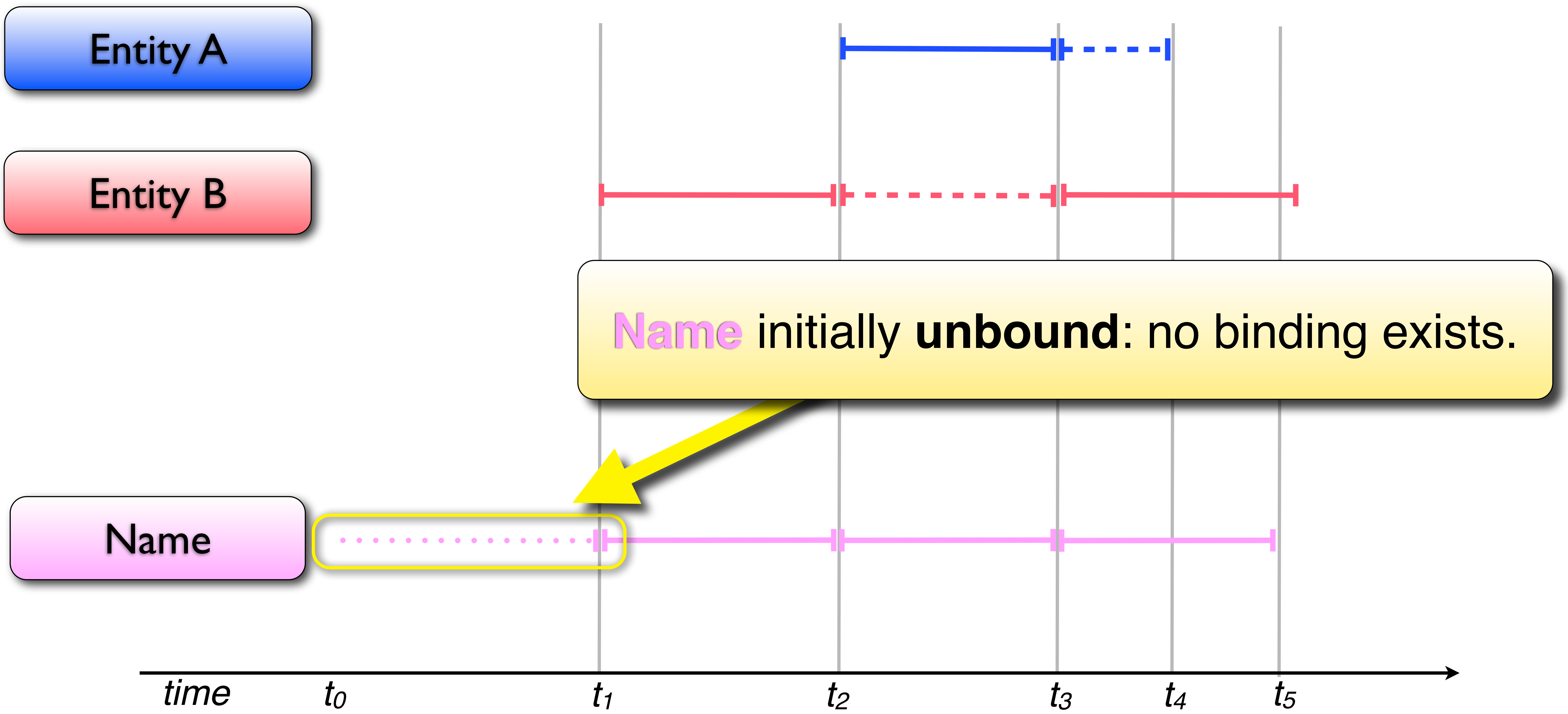


Lifetime

➡ Entity: “alife” if memory is allocated (and initialized).

➡ Binding: “alife” if the name refers to some entity.

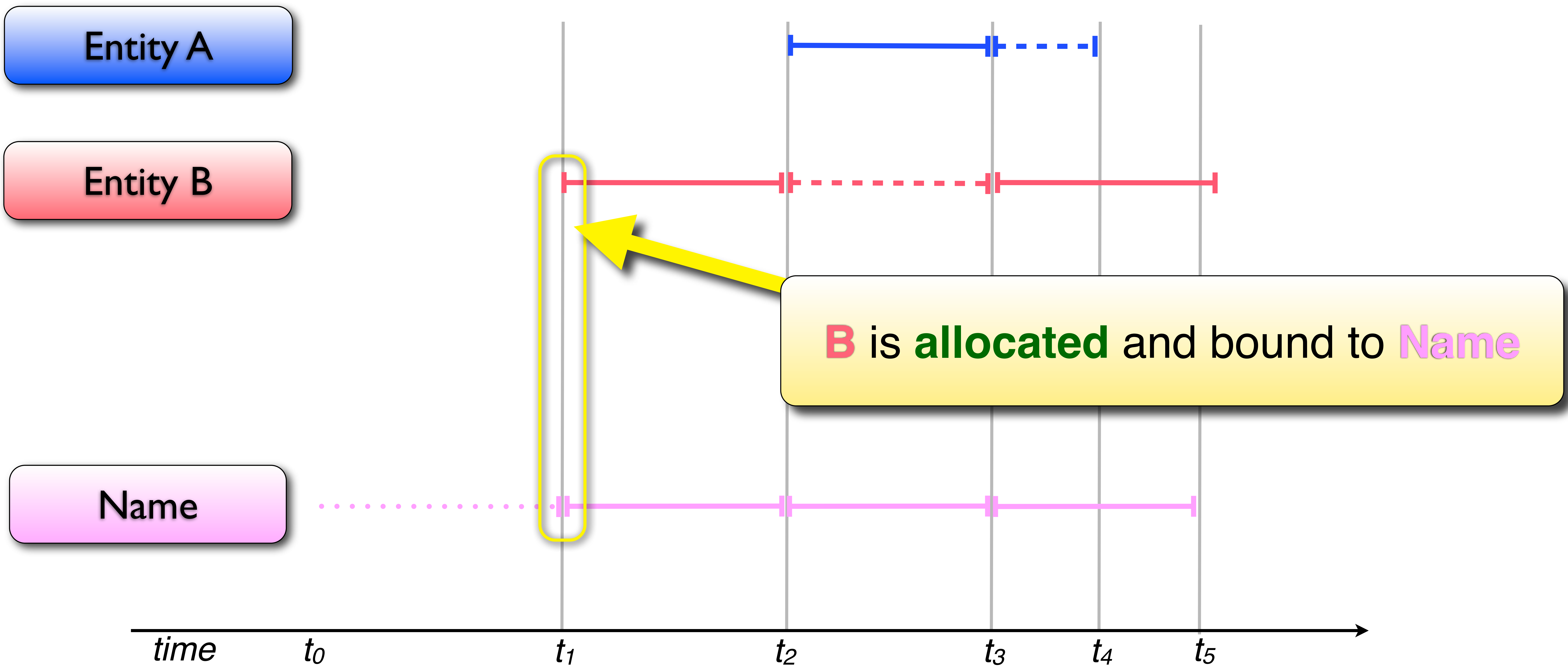
Object Lifetime vs. Binding Lifetime



Lifetime

- ➡ Entity: “alife” if memory is allocated (and initialized).
- ➡ Binding: “alife” if the name refers to some entity.

Object Lifetime vs. Binding Lifetime

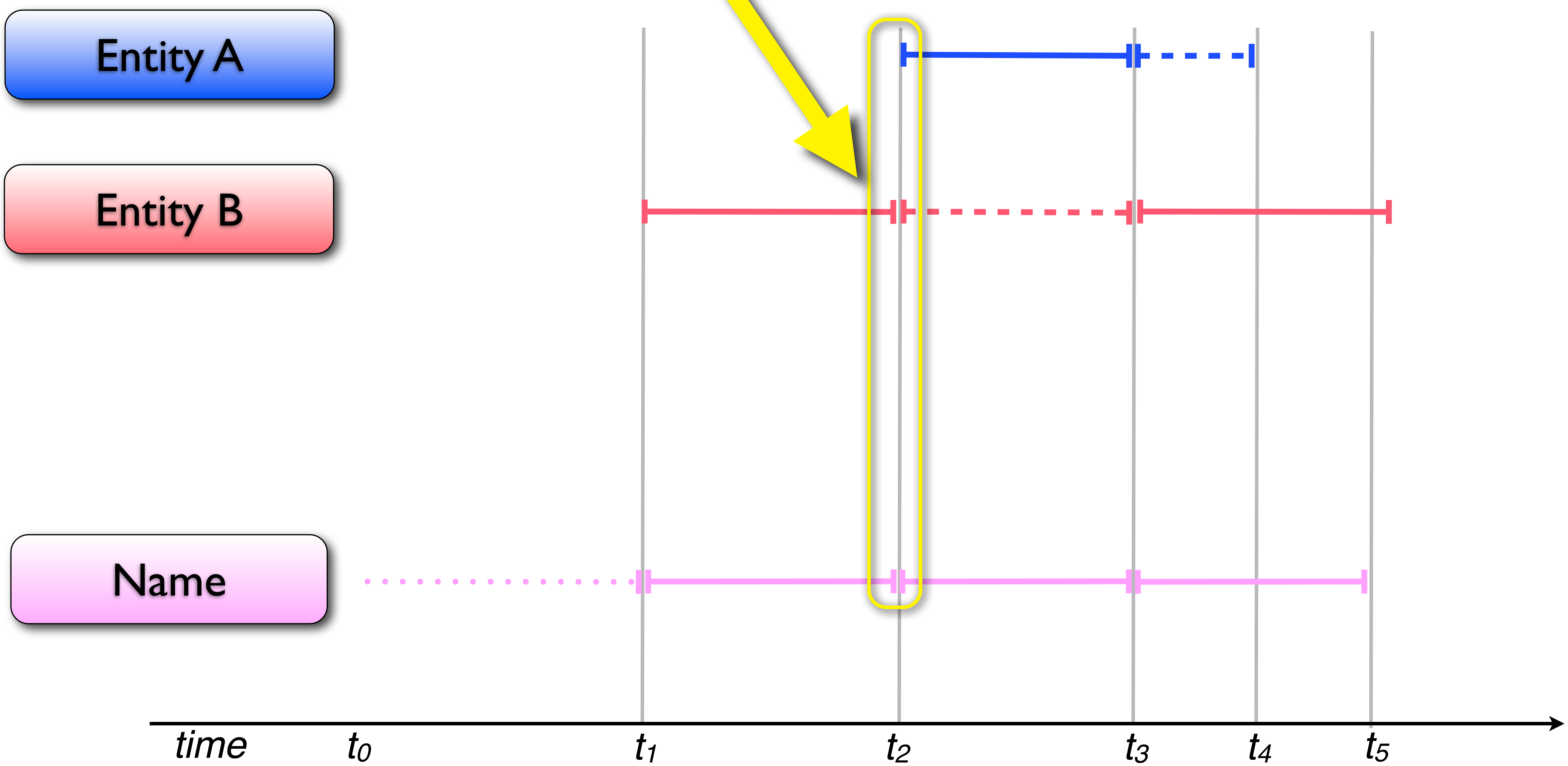


Lifetime

- ➡ Entity: “alife” if memory is allocated (and initialized).
- ➡ Binding: “alife” if the name refers to some entity.

Ob ding Lifetime

A is **allocated** and bound to **Name**.
B still exists!

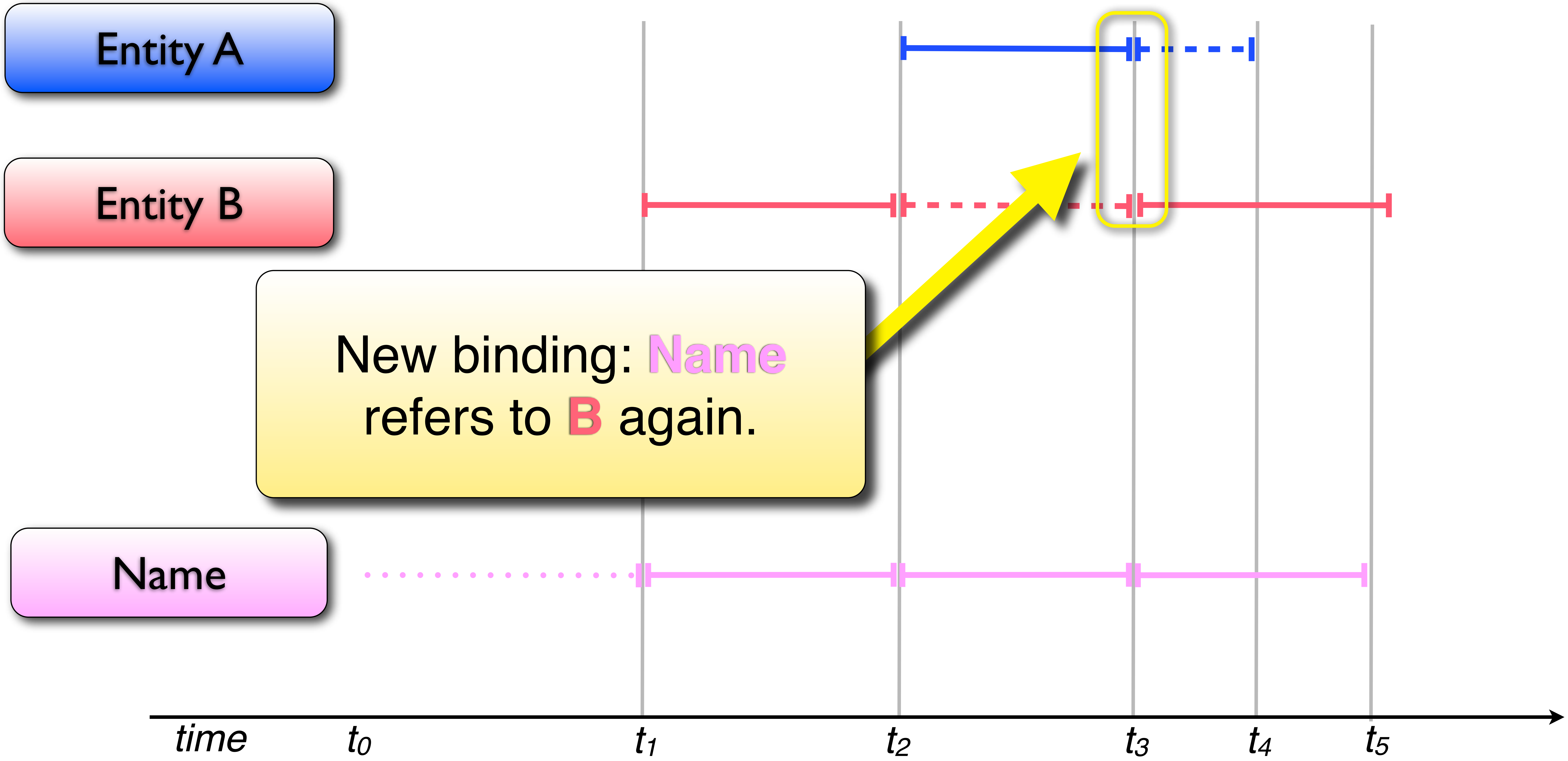


Lifetime

➡ Entity: “alife” if memory is allocated (and initialized).

➡ Binding: “alife” if the name refers to some entity.

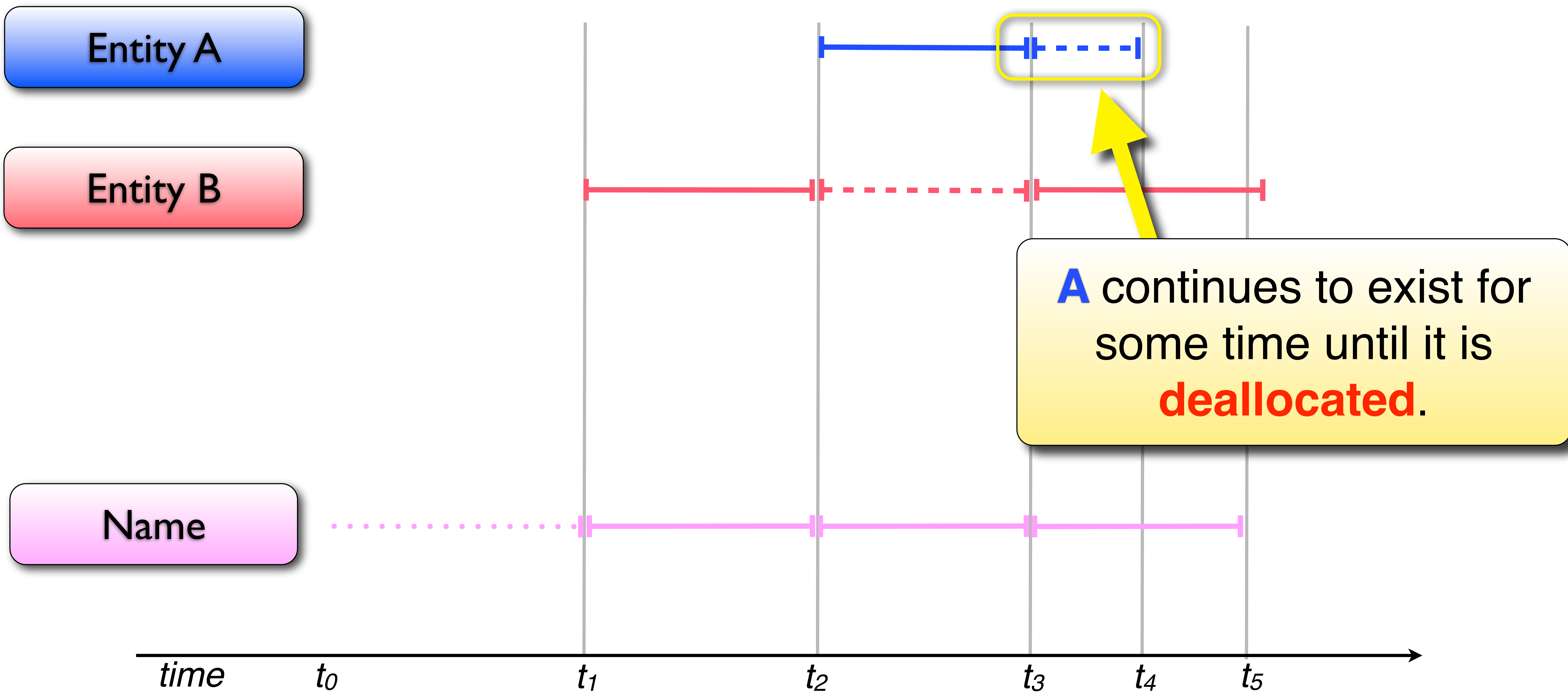
Object Lifetime vs. Binding Lifetime



Lifetime

- ➡ Entity: “alife” if memory is allocated (and initialized).
- ➡ Binding: “alife” if the name refers to some entity.

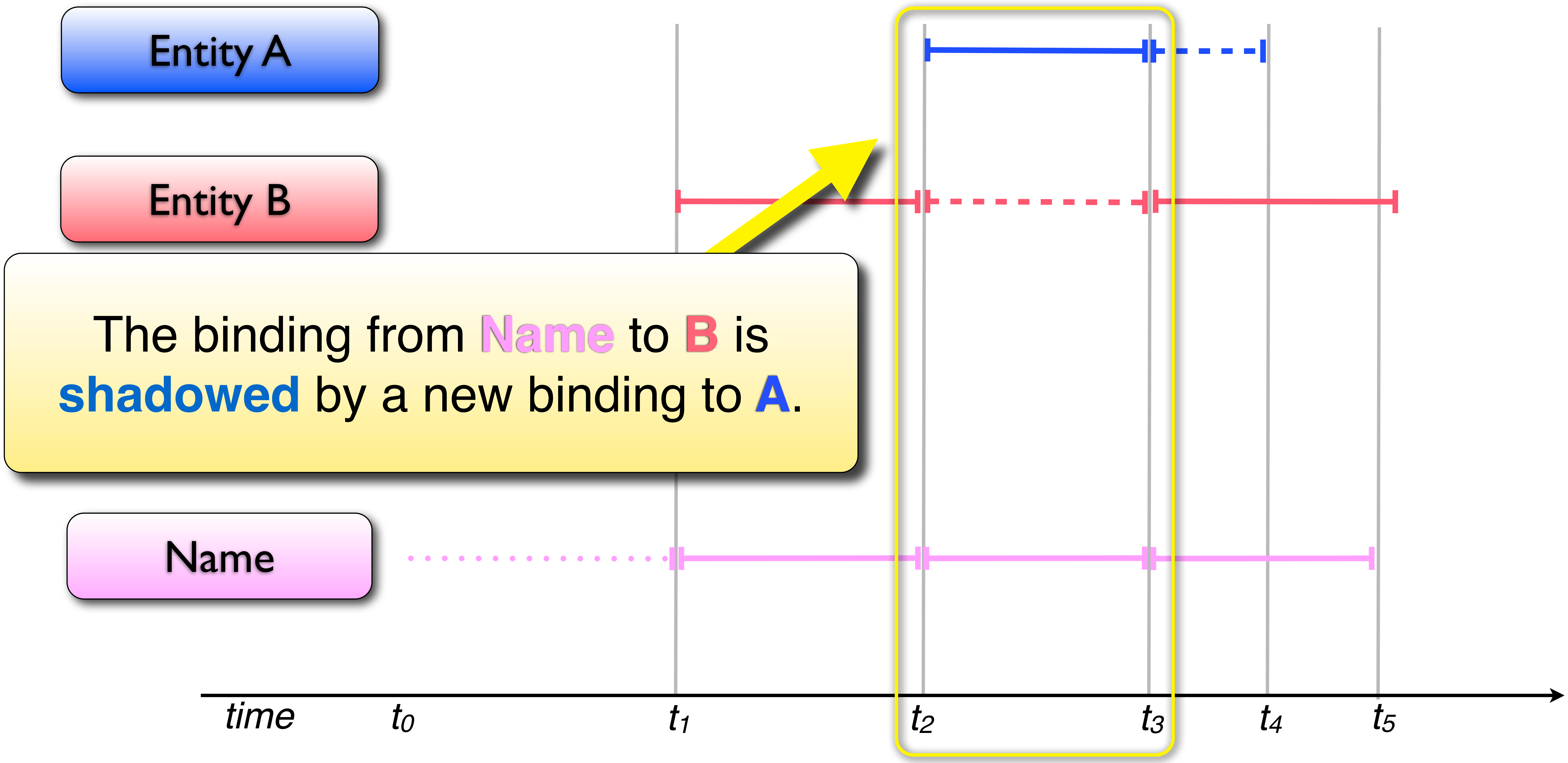
Object Lifetime vs. Binding Lifetime



Lifetime

- ➡ Entity: “alife” if memory is allocated (and initialized).
- ➡ Binding: “alife” if the name refers to some entity.

Object Lifetime vs. Binding Lifetime



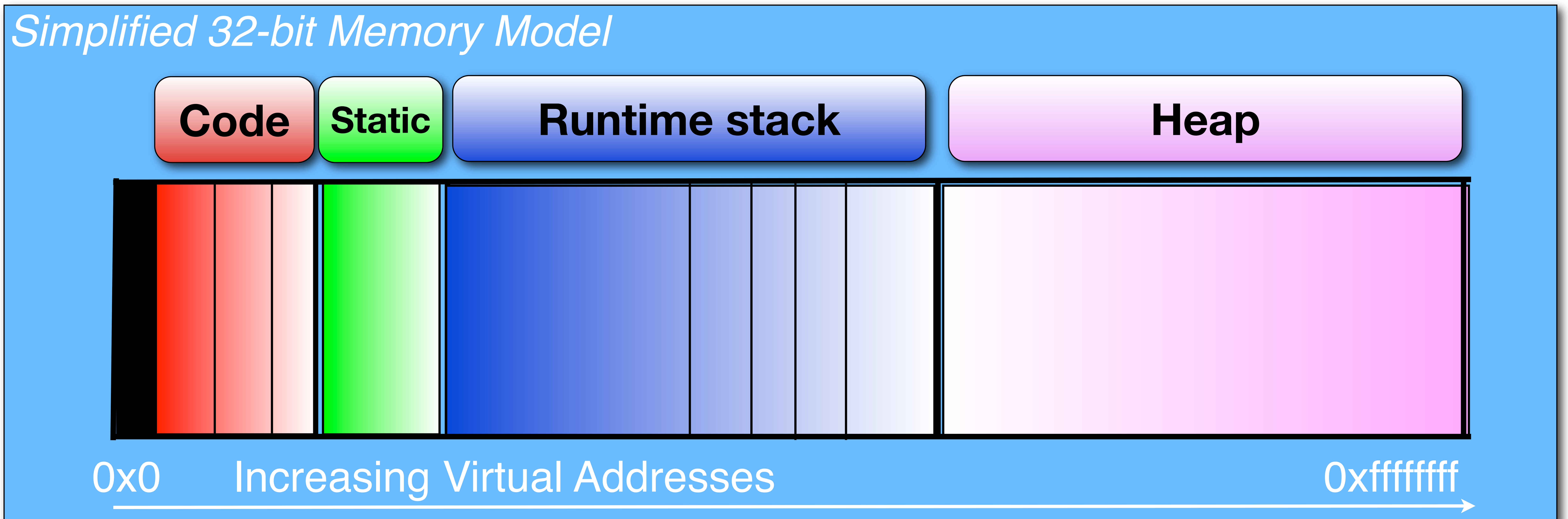
Lifetime

- ➡ Entity: "alife" if memory is allocated (and initialized).
- ➡ Binding: "alife" if the name refers to some entity.

Object Lifetimes

Defined by three principal storage allocation mechanisms:

- ➡ **Static** objects, which have a **fixed address** and are not de-allocated before program termination.
- ➡ **Stack** objects, which are allocated and deallocated in a **Last-In First-Out** (LIFO) order.
- ➡ **Heap** objects, which are allocated and deallocated at **arbitrary times**.



Static Allocation

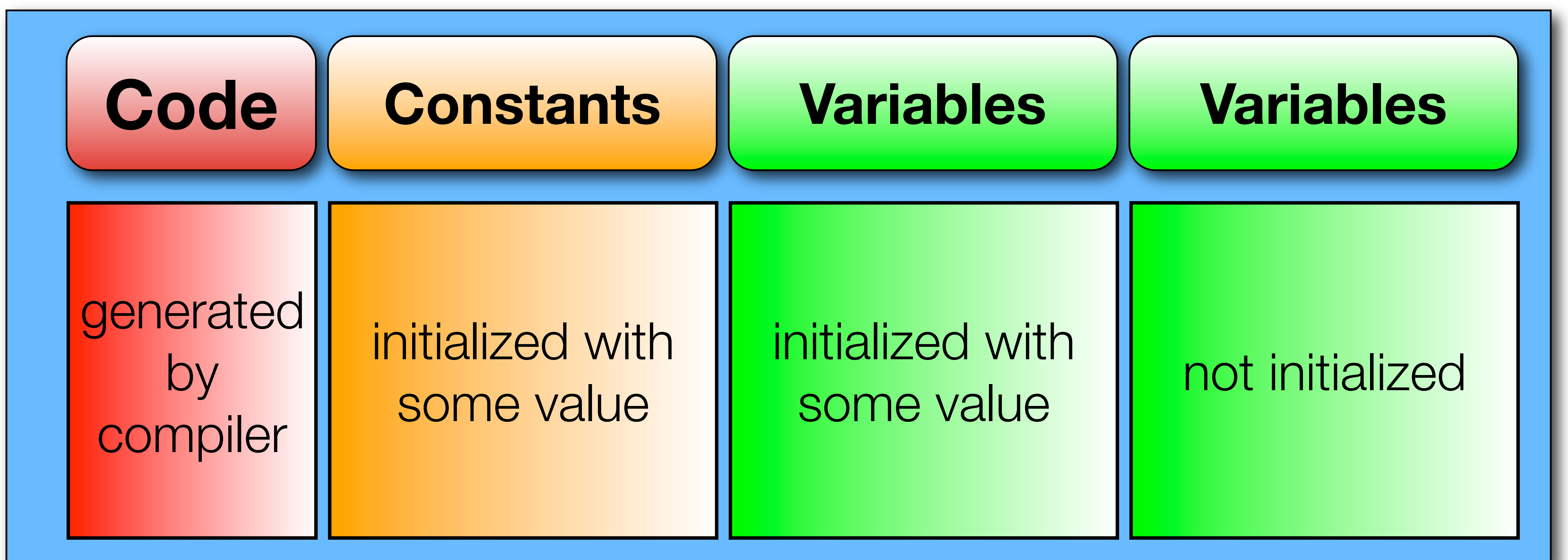
Some memory is required throughout program execution.

- ➡ Multi-byte **constants**.
  - Strings (“hello world”).
  - Lookup tables.
- ➡ **Global variables**.
- ➡ The program **code**.

> **Caution**: this is not the same as Java’s **`static`**.

Must be allocated before program execution starts.

- ➡ Requirements specified in program file.
- ➡ **Allocated by OS** as part of program loading.
- ➡ The **size of static allocation is constant** between runs.



Static Allocation

Some memory is required throughout program execution.

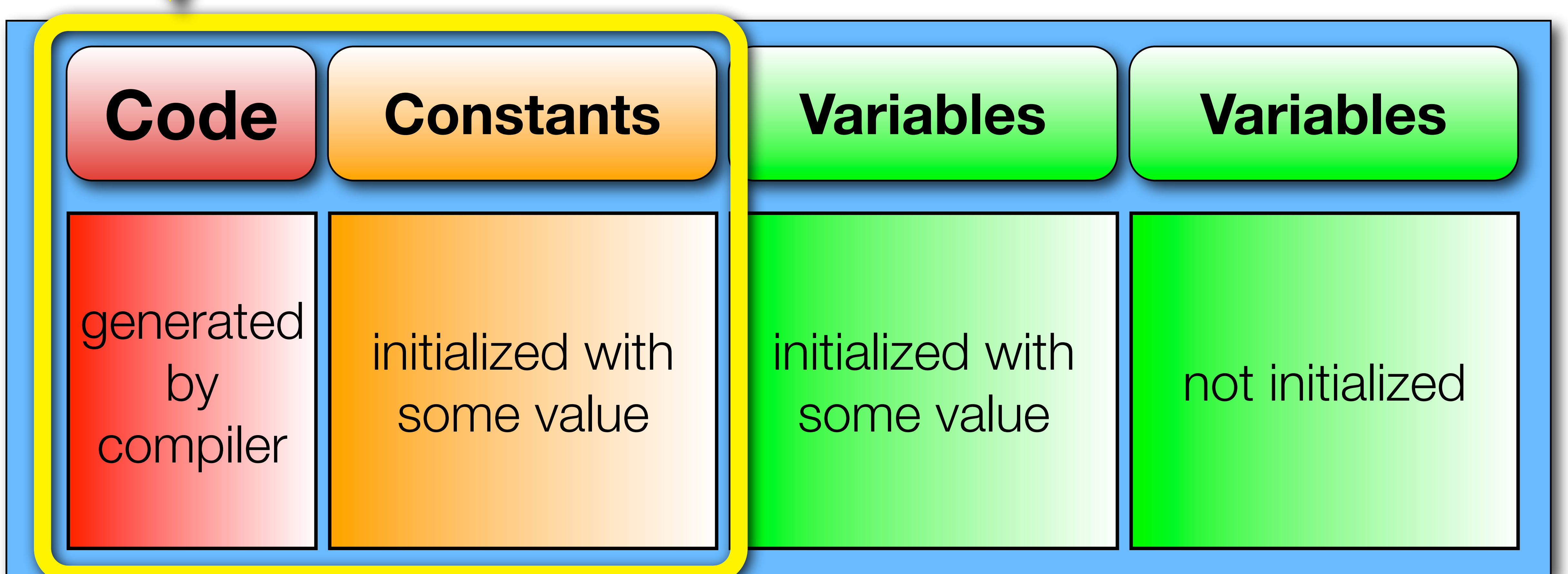
- Multi-byte **constants**.
  - Str
  - Lo
- **Glo**
- The

is not the same as Java's **`static`**.

Read-only data.
Attempt to update illegal
in many operating systems.

Must be allocated before program execution starts.

- Requirements specified in program file.
- **Allocated by OS** as part of program loading.
- The **size of static allocation is constant** between runs.



Static Allocation

Some memory is required throughout program execution.

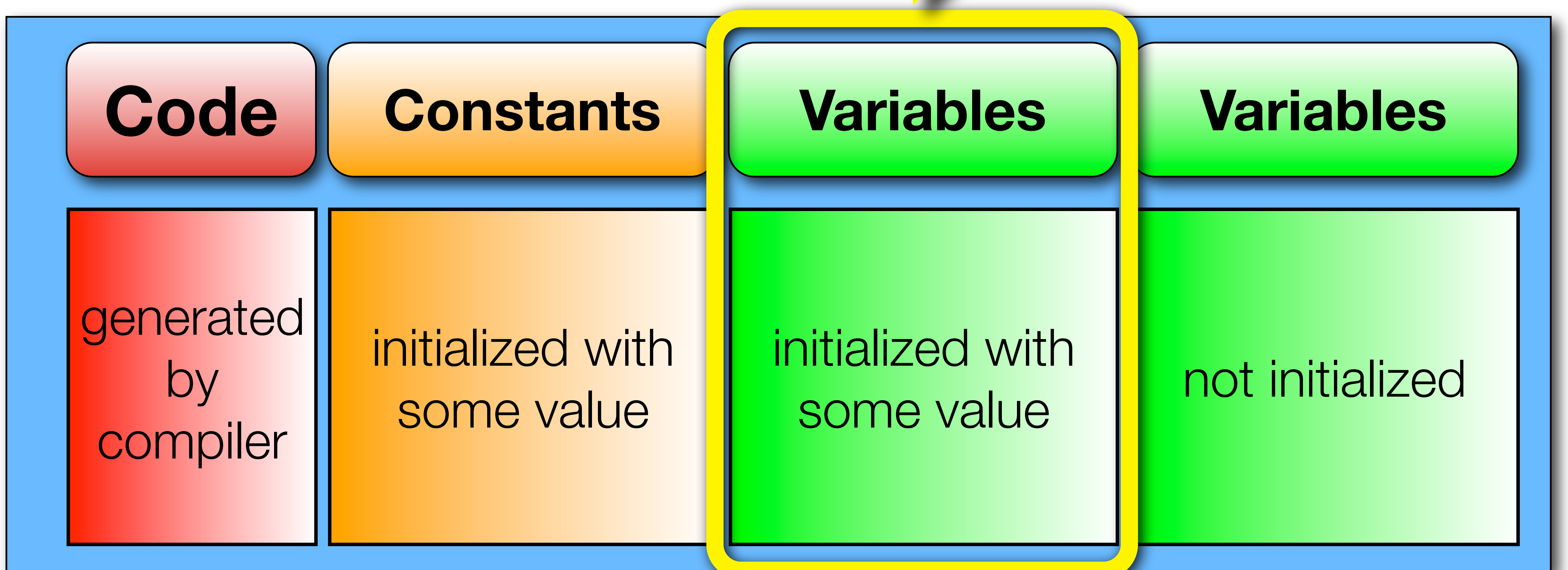
- Multi-byte **constants**.
  - Strings (“hello world”).
  - Lookup tables.
- **Global variables**.
- The program **code**.

Caution ... ic.

Global Initialized Variables.
E.g., `int meaning = 42;`

Must be allocated before program exe...

- Requirements specified in program file.
- **Allocated by OS** as part of program loading.
- The **size of static allocation is constant** between runs.



Static Allocation

Some memory is required throughout program execution.

- ➡ Multi-byte **constants**.
  - Strings (“hello wo
  - Lookup tables.
- ➡ **Global variables**.
- ➡ The program **code**

atic.

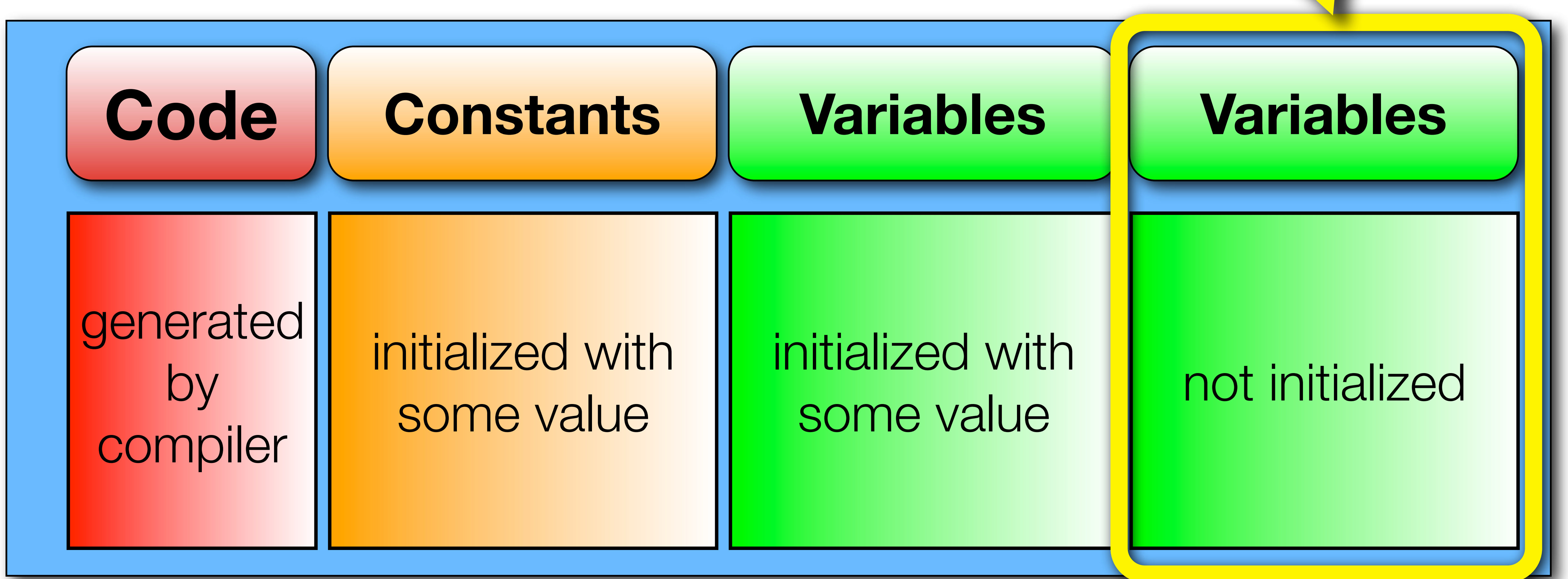
Must be allocated be

- ➡ Requirements specified in program file.
- ➡ **Allocated by OS** as part of program loading.
- ➡ The **size of static allocation is constant** between runs.

Global Uninitialized Variables.

E.g., `int last_error;`

(For historic reasons called the **.bss** segment.)



Static Allocation

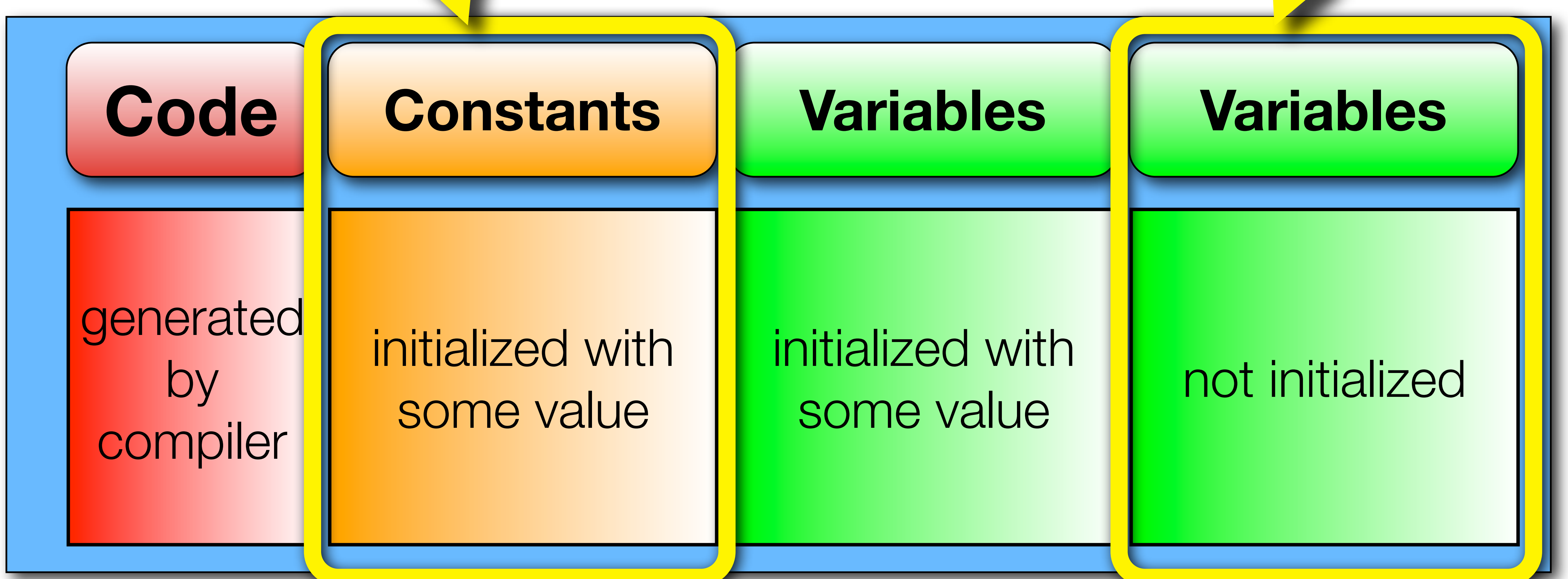
Some memory is required throughout program execution.
- ➡ Multi-byte **constants**
- ➡ The program **code**.

his is not the same as Java's `static`.

Compile-time constants.
Value must be known at compile time.

Elaboration-time constants.
Value computed at runtime; compiler disallows subsequent updates.

Must be allocated before program execu
- ➡ Requirements specified in program file.
- ➡ **Allocated by OS** as part of program loa
- ➡ The **size of static allocation is constant** between runs.



Advantages & Disadvantages

Advantages.

- **No** allocation/deallocation **runtime overhead**.
- Static addresses.
- Compiler can **optimize** accesses.

Limitations.

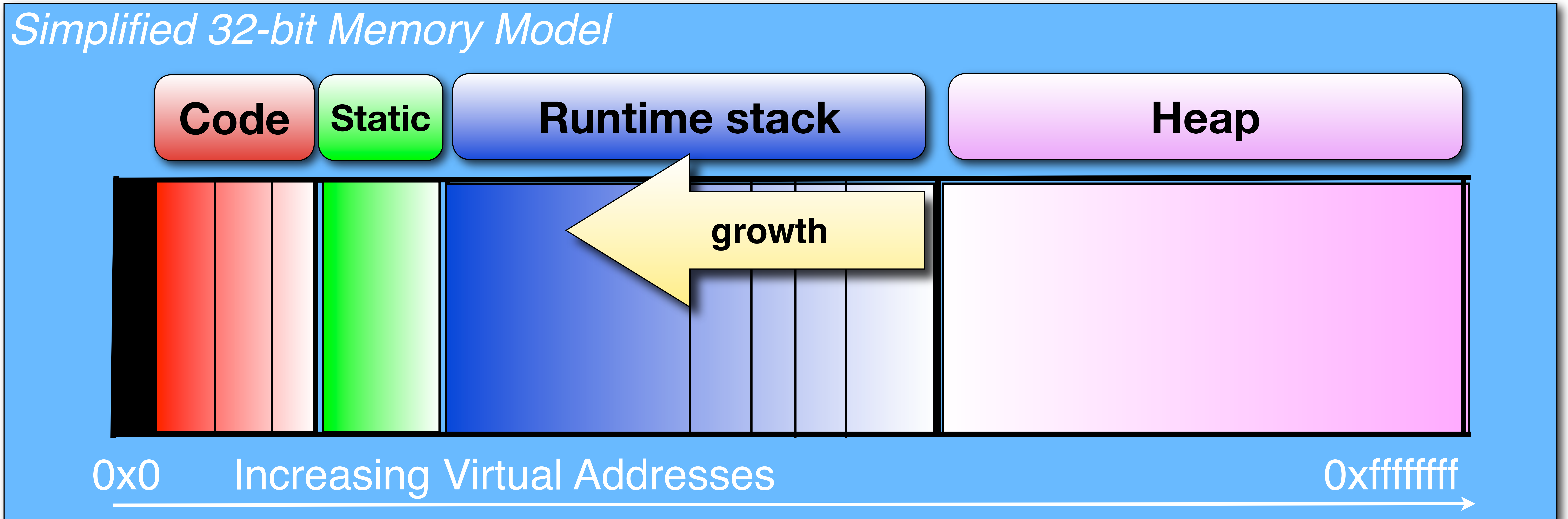
- Allocation **size fixed**; cannot depend on input.
- **Wasteful**; memory is always allocated.
- Global variables are **error-prone**.

Advice: avoid global variables.

Runtime Stack

Hardware-supported allocation area.

- ➡ Essential for **subprogram calls**.
- ➡ Grows **top-down** in many architectures.
- ➡ **Size limit**: stack overflow if available space is exhausted.
- ➡ Max. size of stack can be **adjusted at runtime**.
- ➡ OS is often involved in stack management.



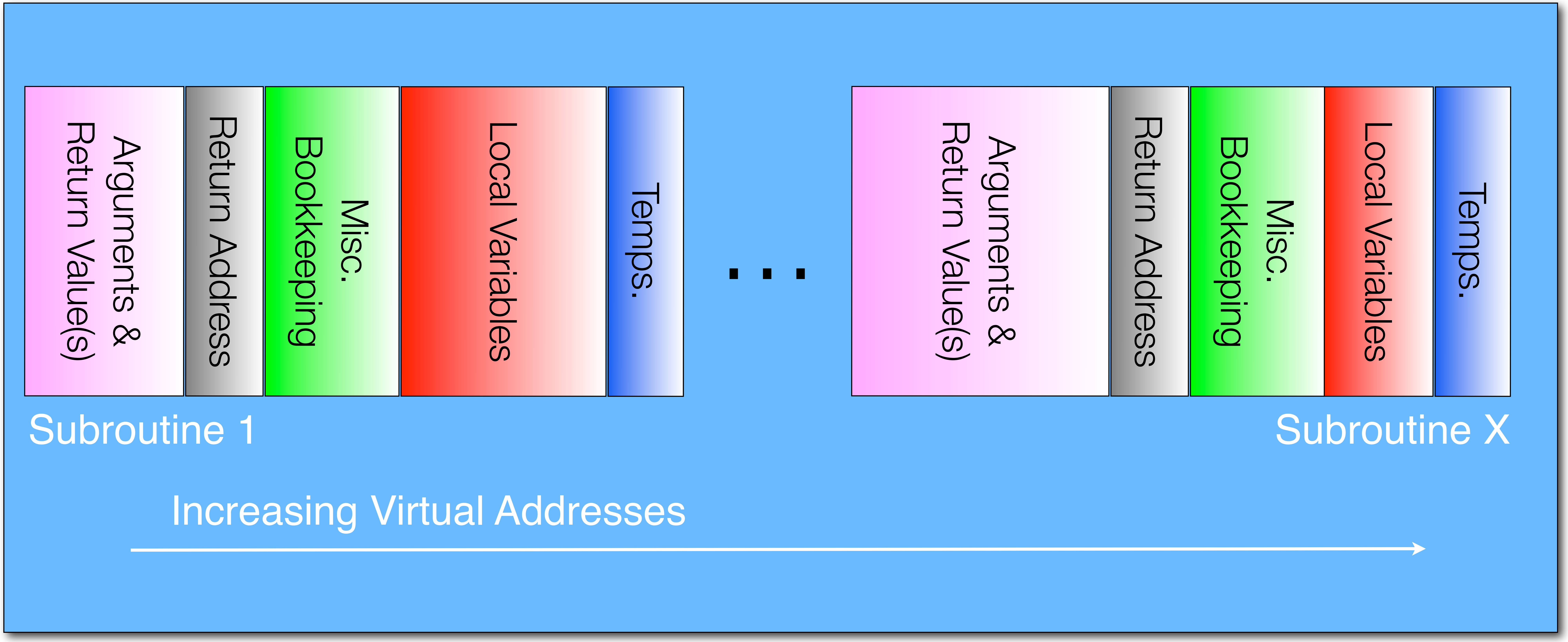
Subroutines & Static Allocation

Calling a function/method/subroutine requires memory.

- ➡ **Arguments**.
- ➡ **Local variables**.
- ➡ **Return address**
  (where to continue execution after subroutine completes).
- ➡ Some **bookkeeping** information.
  - E.g., to support **exceptions** and call **stack traces**.

Where should this memory be allocated?

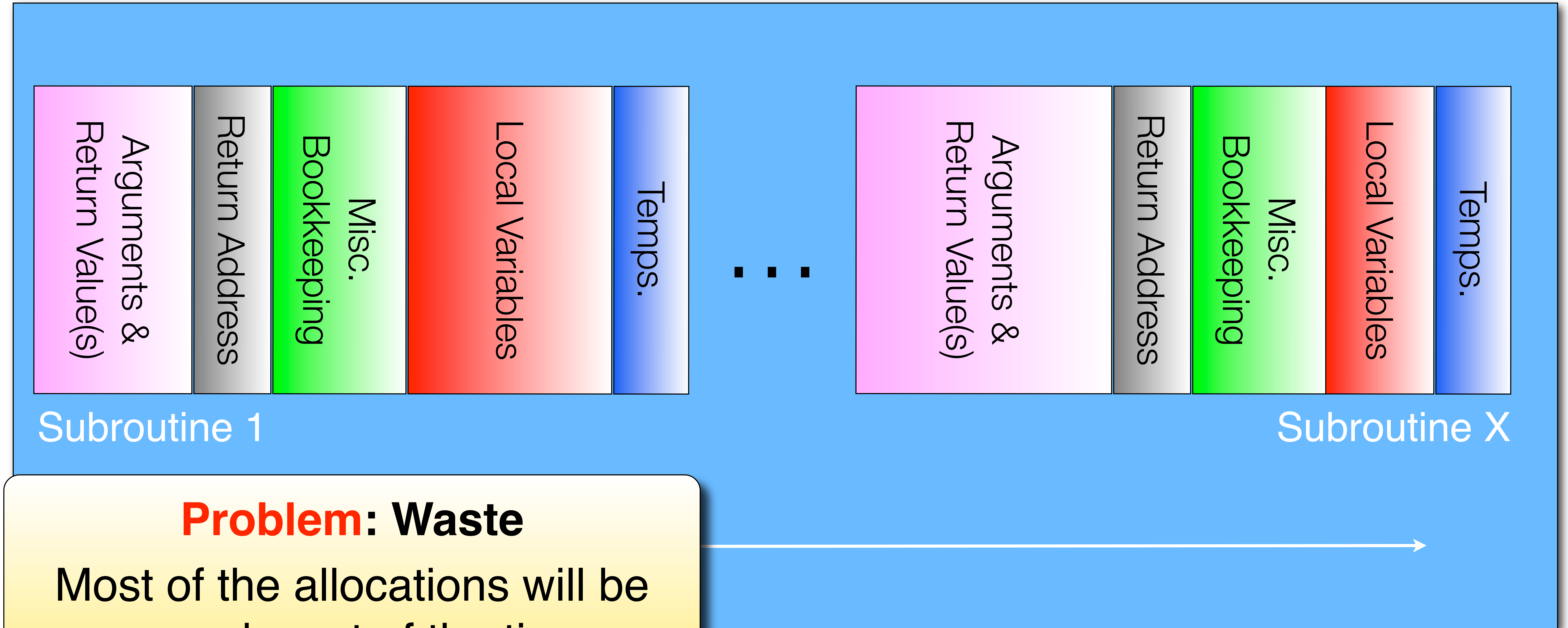
Static Allocation of Subroutine Memory



One approach: statically allocate memory for each subroutine.

(e.g. early versions of Fortran)

Static Allocation of Subroutine Memory



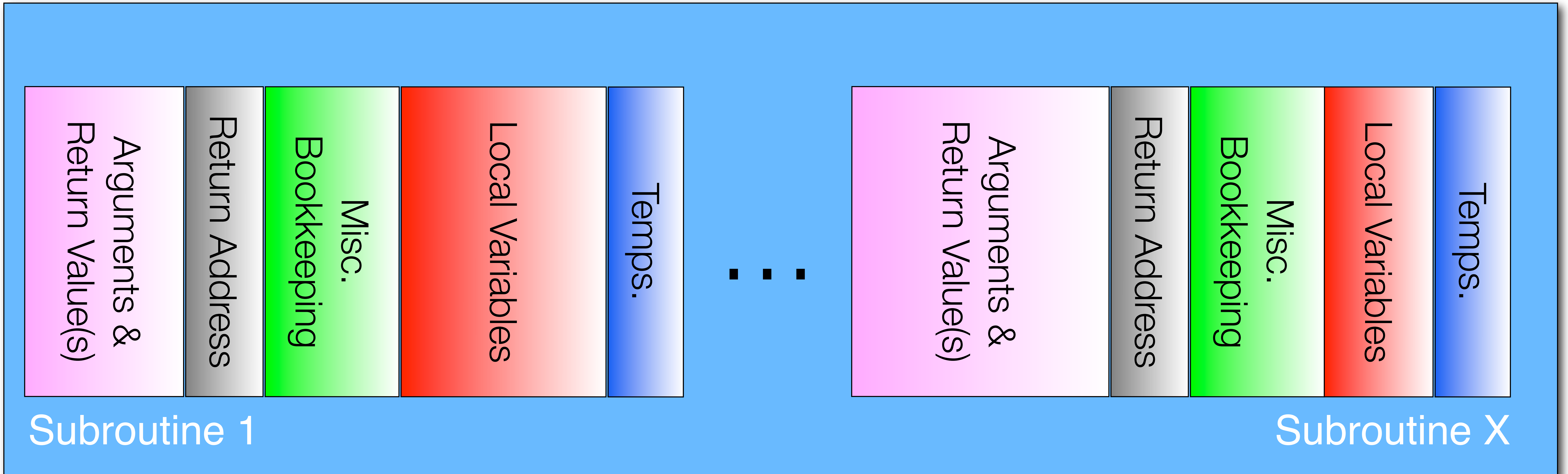
Problem: Waste

Most of the allocations will be unused most of the time.

One approach: statically allocate memory for each subroutine.

(e.g. early versions of Fortran)

Static Allocation of Subroutine Memory



Problem: Waste

Most of the allocations will be unused most of the time.

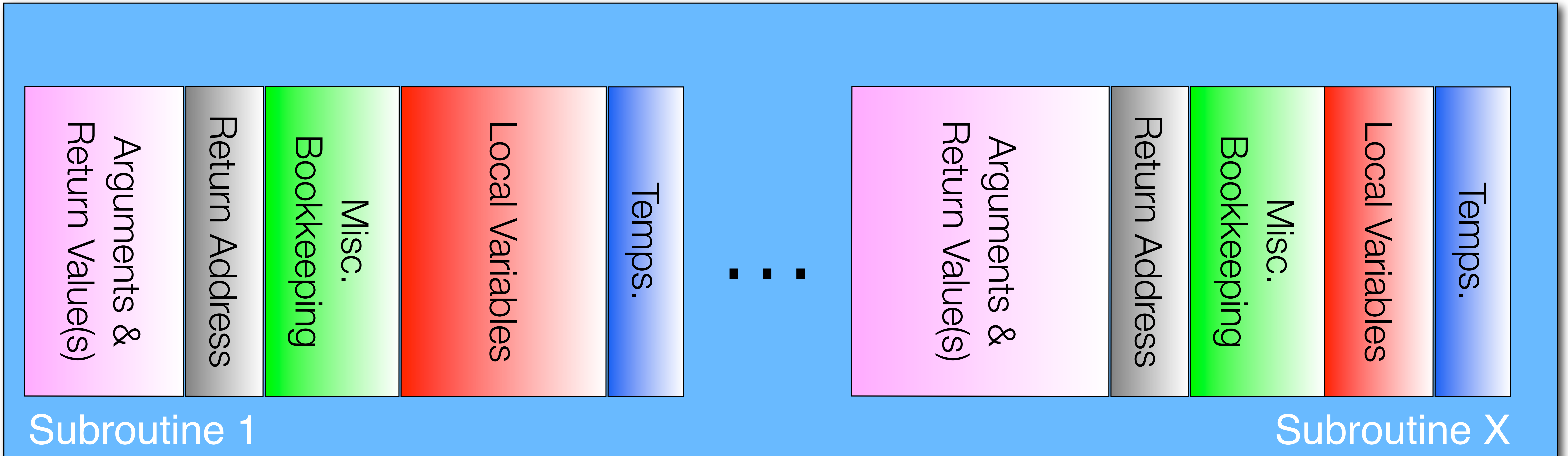
Problem: No Recursion

Subroutines may not be called while their memory is already in use.

One approach: statically allocate memory for each subroutine.

(e.g. early versions of Fortran)

Static Allocation of Subroutine Memory



Problem: Waste

Most of the allocations will be unused most of the time.

Problem: No Recursion

Subroutines may not be called while their memory is already in use.

Limited recursion depth can be allowed by allocating memory for multiple subroutine ***instances***. But this **increases waste**…

Runtime Stack

Idea: allocate memory for subroutines on demand.

- ➡ Reserve (large) area for subroutine calls. Allocate new ***frame*** for each call. Deallocate on return.
- ➡ Subroutine calls must be **fast**: need **efficient** memory management.

Observation: last-in first-out allocation pattern.

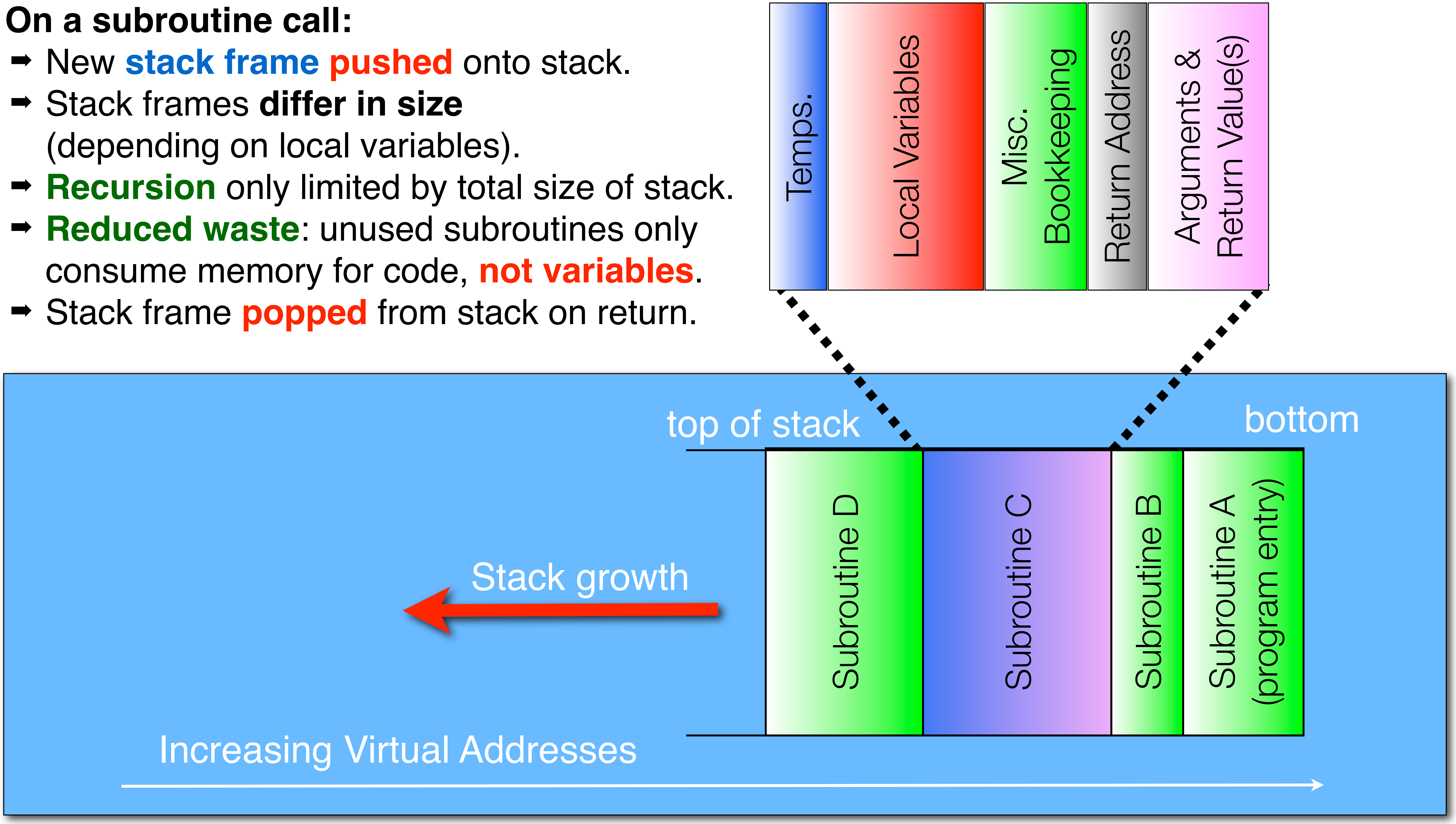
- ➡ A routine returns only after all called subroutines have returned.
- ➡ Thus, allocations can be "piled" on top of each other.

Subroutine memory is allocated on-demand from the runtime stack.

Stack Frames

On a subroutine call:

- New **stack frame pushed** onto stack.
- Stack frames **differ in size** (depending on local variables).
- **Recursion** only limited by total size of stack.
- **Reduced waste**: unused subroutines only consume memory for code, **not variables**.
- Stack frame **popped** from stack on return.



Calling Sequence

Compilers generate code to manage the runtime stack.

- ➡ **Setup,** before call to subroutine.
- ➡ **Prologue,** before subroutine body executes.
- ➡ **Epilogue,** after subroutine body completes (the **return**).
- ➡ **Cleanup,** right after subroutine call.

```
private void checkForKleeneClosure(NFA fa) throws IOException {
    if (readNextChar() == '*') {
        // indeed, it's a closure.
        fa.createKleeneClosure();
    } else
        unreadLastChar();
}
```

Calling Sequence

Compilers generate code to manage the runtime stack.

- ➡ **Setup,** before call to subroutine.
- ➡ **Prologue,** before subroutine body executes.
- ➡ **Epilogue,** after subroutine body completes (the **return**).
- ➡ **Cleanup,** right after subroutine call.

```
private void checkForKleeneClosure(NFA fa) throws IOException {
    if (readNextChar() == '*') {
        // indeed, it's a closure.
        fa.createKleeneClosure();
    } else
        unreadLastChar();
}
```

Calling Sequence Example

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}


int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

C program

Calling Sequence Example

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}


int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

C program, x86-64 assembly

Calling Sequence Example

```
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}


int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

Call of `add()` from `main()`.

Prologue
push stack & load arguments

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}


int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

C program, x86-64 assembly

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

Prologue

push stack & load arguments

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}


int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

C program, x86-64 assembly

Epilogue

setup return value & restore stack

Prologue

push stack & load arguments

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
```

Call Setup

prepare arguments

```c
int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

C program, x86-64 assembly

Epilogue

setup return value & restore stack

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

Prologue
push stack & load arguments

```
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
```

Call Setup
prepare arguments

```
int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

Cleanup
save return value

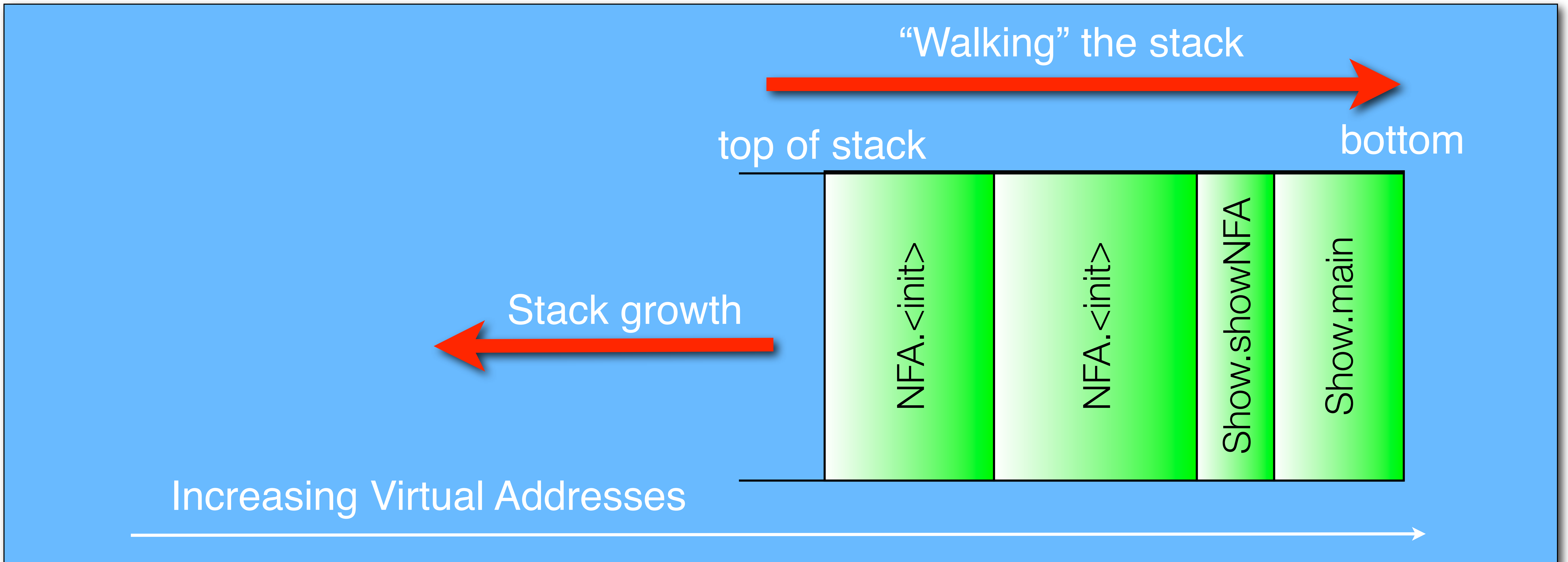
Epilogue
setup return value & restore stack

```
_add:
0000000100000ea0    pushq   %rbp
0000000100000ea1    movq    %rsp,%rbp
0000000100000ea4    movl    %edi,0xec(%rbp)
0000000100000ea7    movl    %esi,0xe8(%rbp)
0000000100000eaa    movl    0xe8(%rbp),%eax
0000000100000ead    addl    0xec(%rbp),%eax
0000000100000eb0    movl    %eax,0xfc(%rbp)
0000000100000eb3    movl    0xfc(%rbp),%eax
0000000100000eb6    leave
0000000100000eb7    ret
```

```
_main:
0000000100000eb8    pushq   %rbp
0000000100000eb9    movq    %rsp,%rbp
0000000100000ebc    subq    $0x20,%rsp
0000000100000ec0    movl    %edi,0xec(%rbp)
0000000100000ec3    movq    %rsi,0xe0(%rbp)
0000000100000ec7    movl    $0x00000014,%esi
0000000100000ecc    movl    $0x0000000a,%edi
0000000100000ed1    callq   0x00000ea0
0000000100000ed6    movl    %eax,0xfc(%rbp)
0000000100000ed9    movl    0xfc(%rbp),%esi
0000000100000edc    leaq    0x00000041(%rip),%rdi
0000000100000ee3    movl    $0x00000000,%eax
0000000100000ee8    callq   0x00000efa
0000000100000eed    movl    $0x00000000,%eax
0000000100000ef2    leave
0000000100000ef3    ret
```

Stack Trace

```
Exception in thread "main" NotYetImplementedException:
    at NFA.<init>(NFA.java:25)
    at NFA.<init>(NFA.java:16)
    at Show.showNFA(Show.java:68)
    at Show.main(Show.java:23)
```



Advantages & Disadvantages

Advantages.

- **Negligible** (in most cases) **runtime overhead**.
- **Efficient** use of space.
- **Recursion** possible.
- **Offset** of local variable **within frame** usually **constant**.

Limitations.

- Stack space is a **limited resource**.
- Stack frame **size fixed** (in many languages).
- Some **offset computations** required at runtime.
- **Object lifetime limited** to one subroutine invocation.

Advice: use stack allocation when possible.

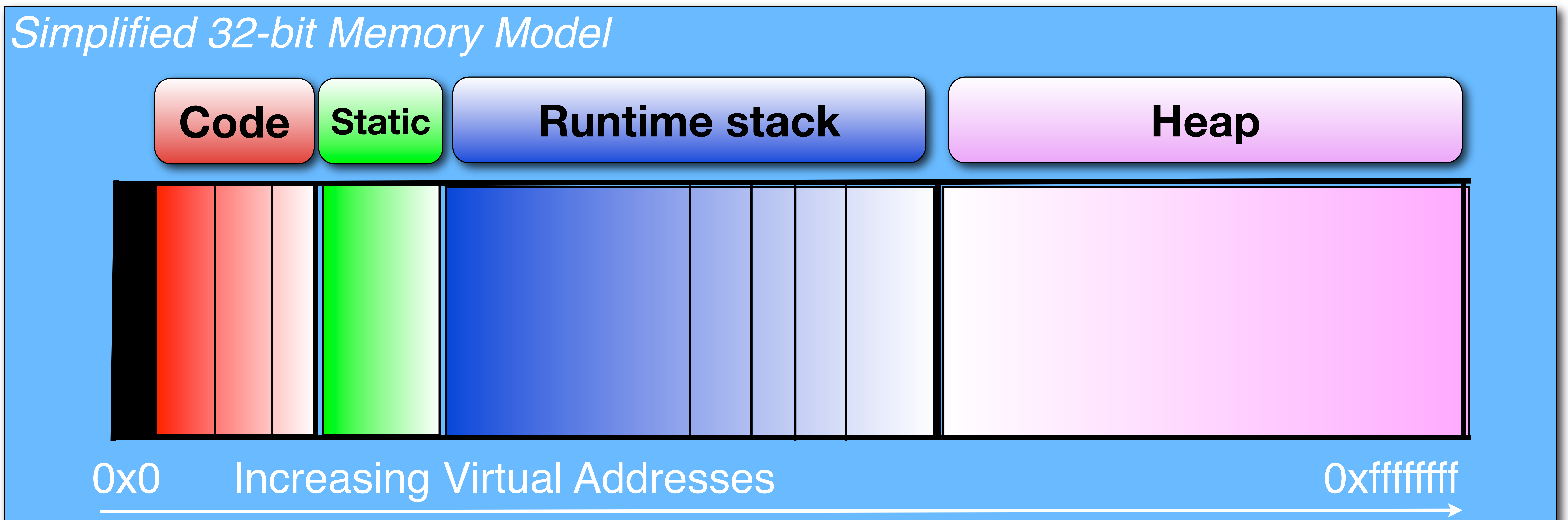
(except for large buffers)

The Heap

(no relation to the data structure of the same name)

Arbitrary object lifetimes.

- ➡Allocation and deallocation **at any point in time**.
- ➡Can persists after subroutine completion.
- ➡Very **flexible**; required for **dynamic allocations**.
- ➡**Most expensive to manage**.



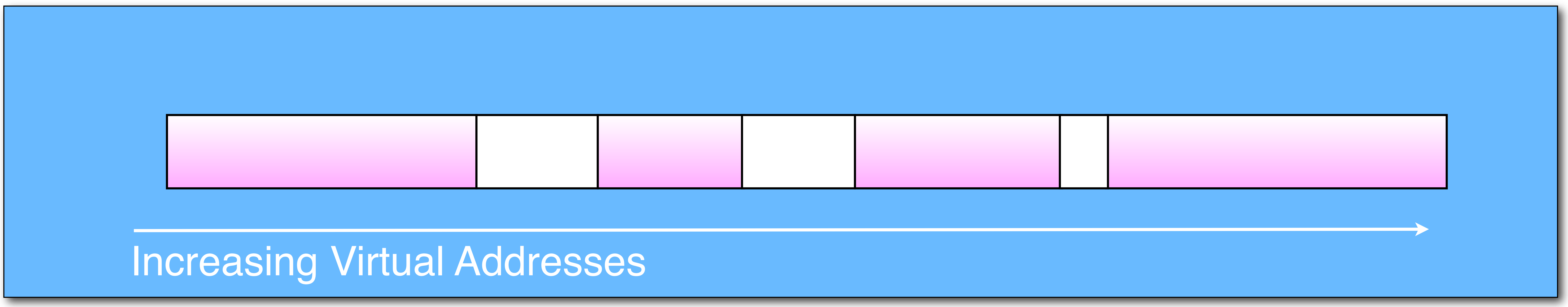
Memory Management

Allocation.

- ➡Often explicit.
  - C++: `new`
- ➡Compiler can generate implicit calls to **allocator**.
  - E.g., Prolog.

Deallocation.

- ➡Often explicit.
  - C++: `delete`
- ➡Sometimes done automatically.



Tuesday, February 23, 2010

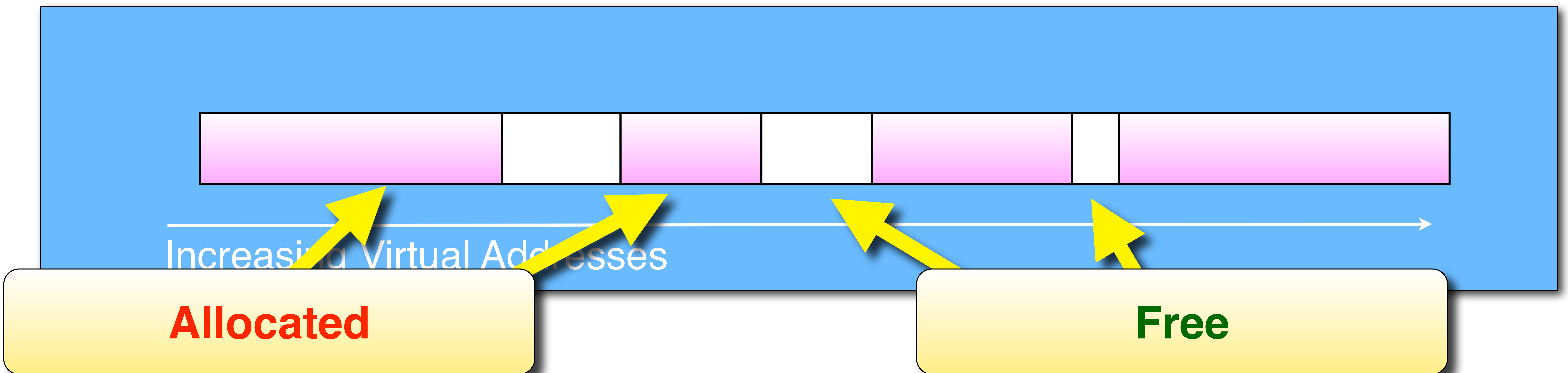
Memory Management

Allocation.

- ➡ Often explicit.
  - C++: `new`
- ➡ Compiler can generate implicit calls to **allocator**.
  - E.g., Prolog.

Deallocation.

- ➡ Often explicit.
  - C++: `delete`
- ➡ Sometimes done automatically.



Memory Management

Allocation.

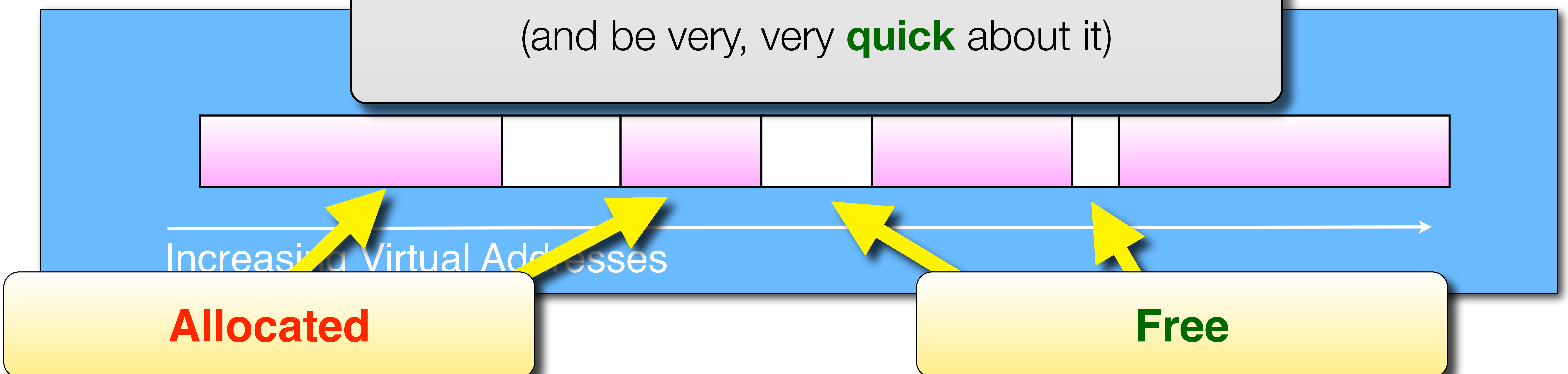
- ➡ Often explicit.
  - C++: `new`
- ➡ Compiler can generate implicit calls to **allocator**.
  - E.g., Prolog

Deallocation.

- ➡ Often explicit.
  - C++: `delete`
- ➡ Sometimes done automatically.

Allocation Problem:
Given a size ***S***, find a **contiguous** region of unallocated memory of length at least ***S***.

(and be very, very **quick** about it)



Common Techniques

Allocator implementation.

- ➡ **Variable size**.
  - Heuristics: First-fit, best-fit, last-fit, worst-fit.
  - List traversals, slow coalescing when deallocated.
- ➡ **Fixed-size blocks**.
  - **2^n** or **Fibonacci** sequence.
  - "**Buddy allocator**," "**slab allocator**"
    - Memory blocks are split until desired block size is reached.
    - Quick coalescing: on free block is merged with its "buddy."

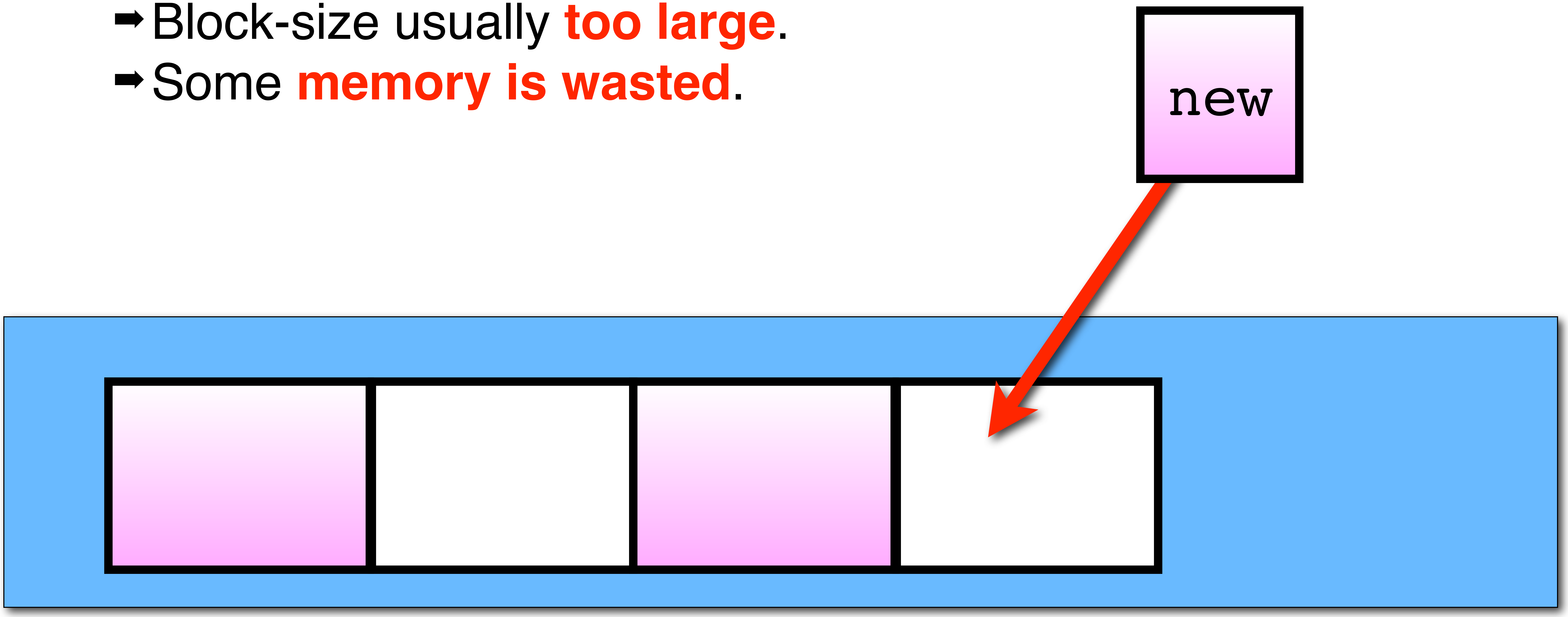
In practice.

- ➡ Most modern OSs use fixed-size blocks.
- ➡ Allocator **performance crucial to many workloads**.
- ➡ Allocators for multicore systems are still being researched.

Internal Fragmentation

Negative impact of fixed-size blocks.

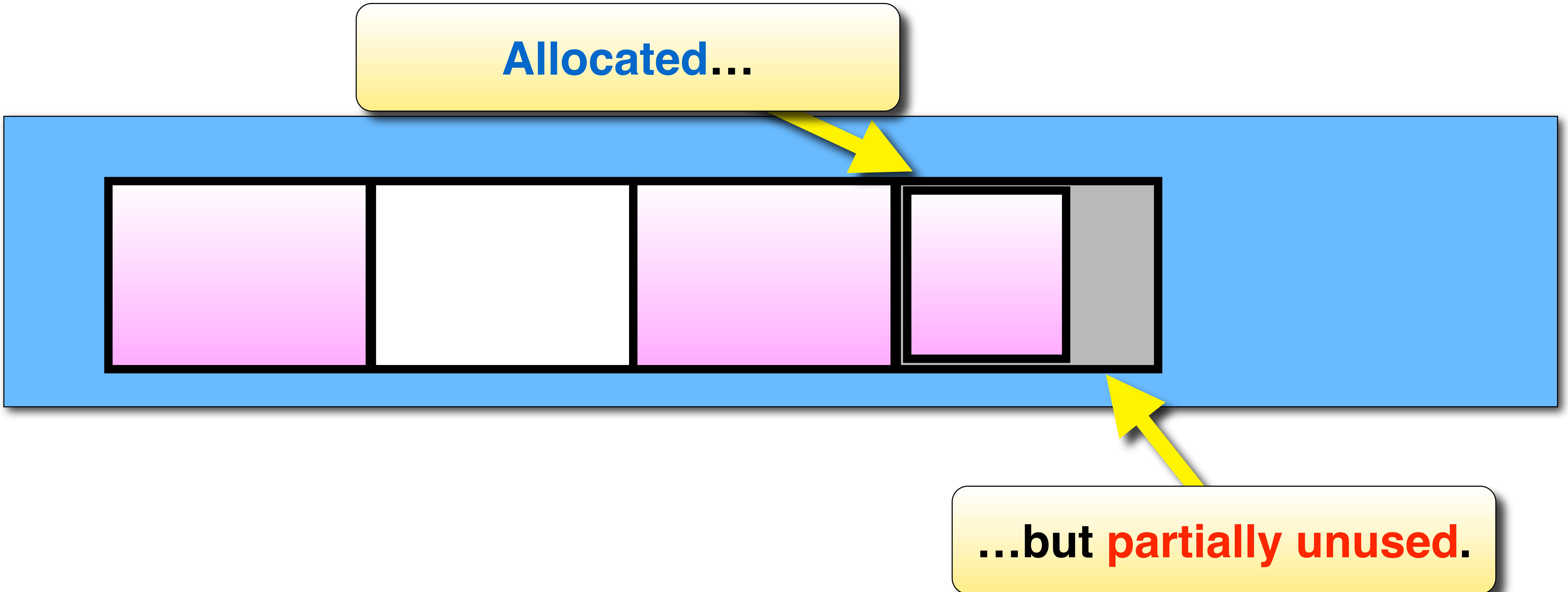
- ➡ Block-size usually **too large**.
- ➡ Some **memory is wasted**.



Internal Fragmentation

Negative impact of fixed-size blocks.

- ➡ Block-size usually **too large**.
- ➡ Some **memory is wasted**.

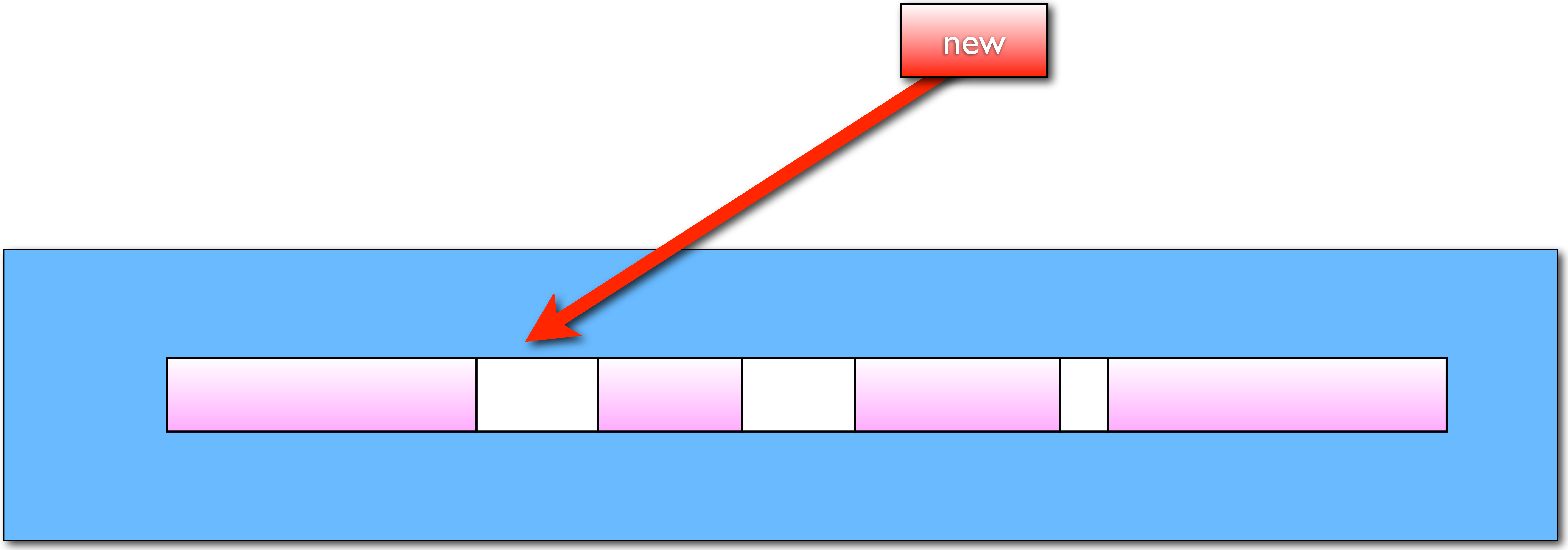


External Fragmentation

Non-contiguous free memory.

➡In total, there is **sufficient available space**…

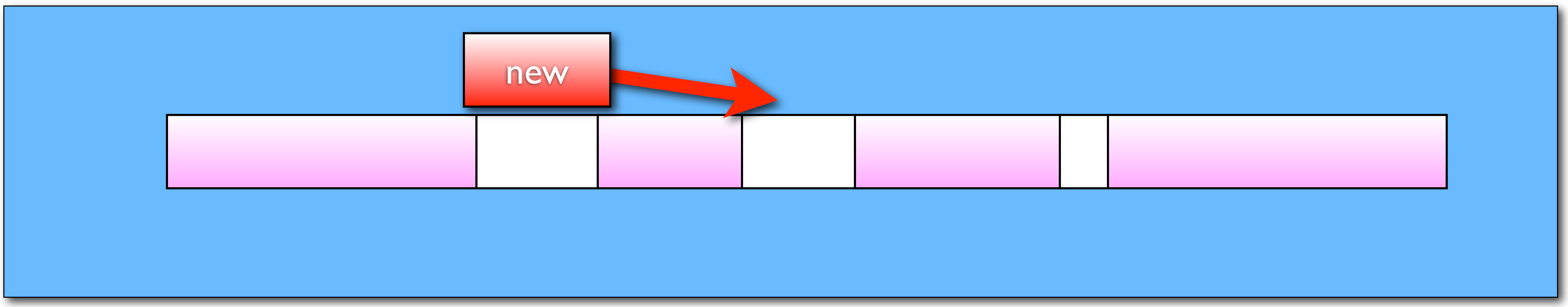
➡…but there is none of the free blocks is large enough by itself.



External Fragmentation

Non-contiguous free memory.

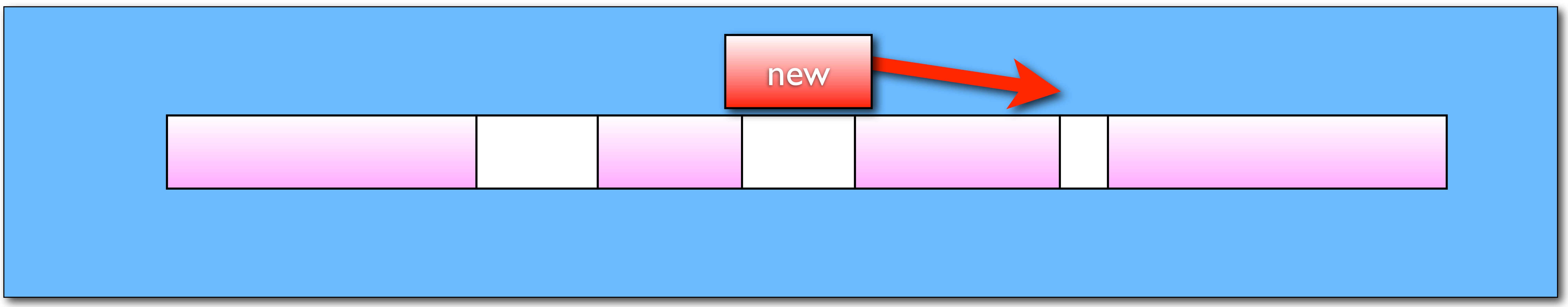
- ➡In total, there is **sufficient available space...**
- ➡...but there is none of the free blocks is large enough by itself.



External Fragmentation

Non-contiguous free memory.

- ➡In total, there is **sufficient available space**…
- ➡…but there is none of the free blocks is large enough by itself.

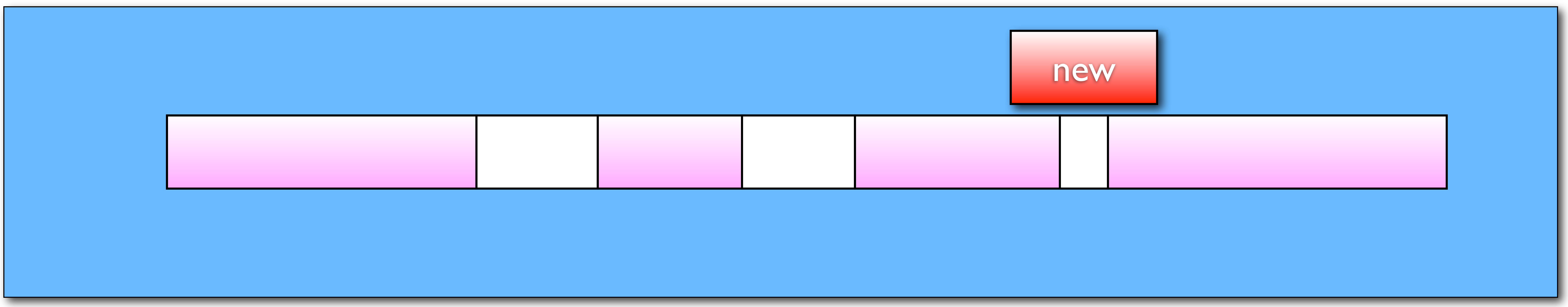


External Fragmentation

Non-contiguous free memory.

➡In total, there is **sufficient available space**…

➡…but there is none of the free blocks is large enough by itself.

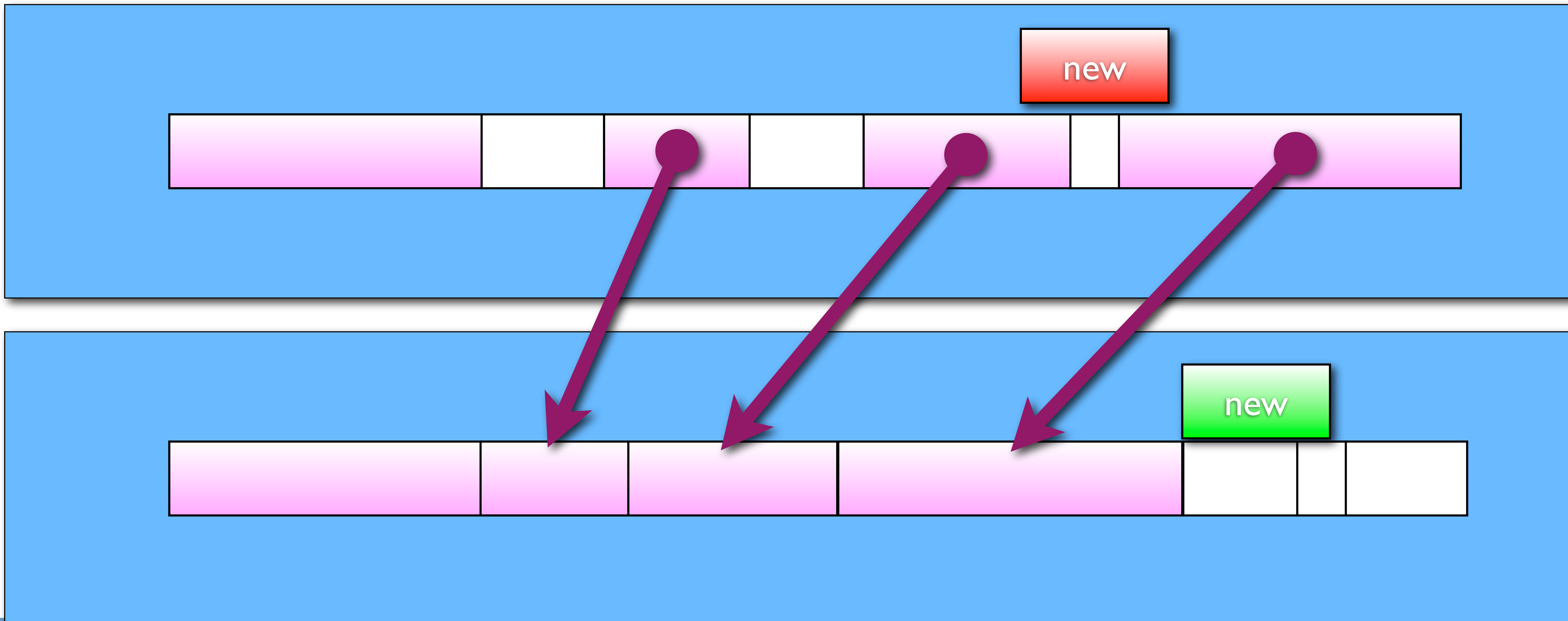


Compacting the Heap

Merge free space.

➡**Copy** existing allocations & **update all references**.

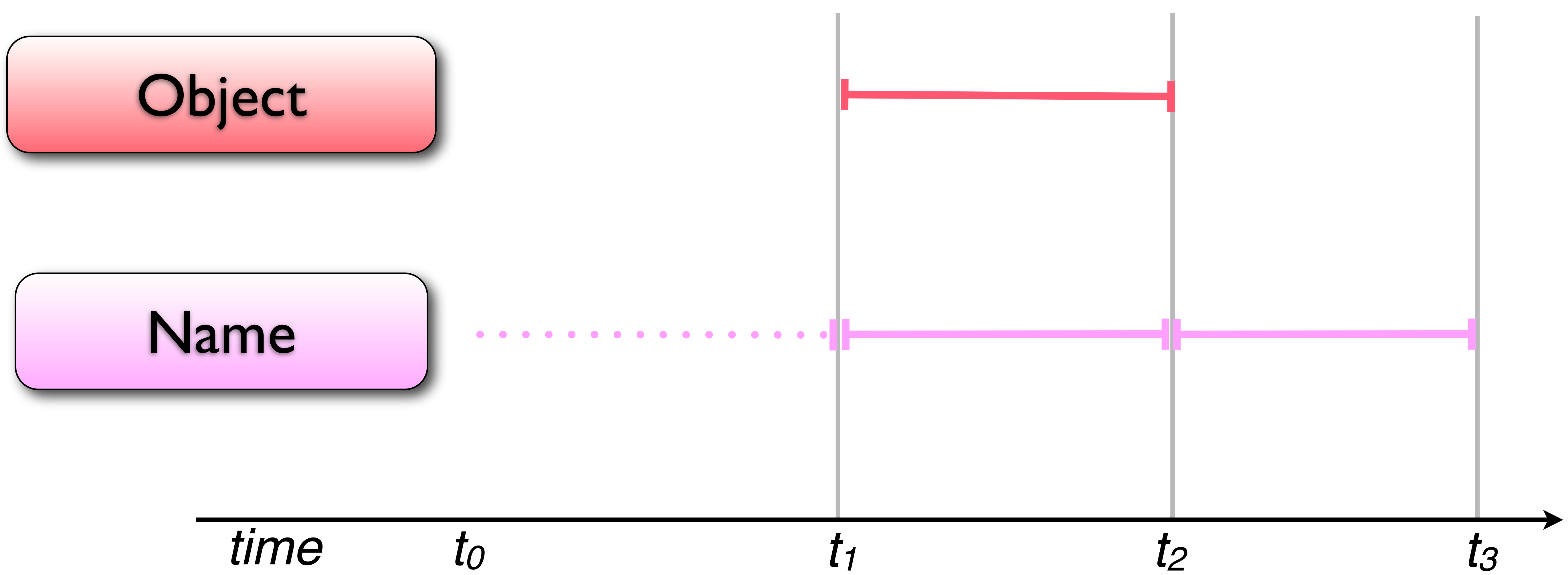
➡Very difficult to implement…



"Dangling" References

Binding / object lifetime mismatch.

- ➡**Binding exists longer than object**.
- ➡Object de-allocated too early; **access** now **illegal**.
- ➡"Use-after-free bug"
  (`free` is the C deallocation routine)
- ➡"Dangling" pointer or reference.



"Dangling" References

Binding / object lifetime mismatch.

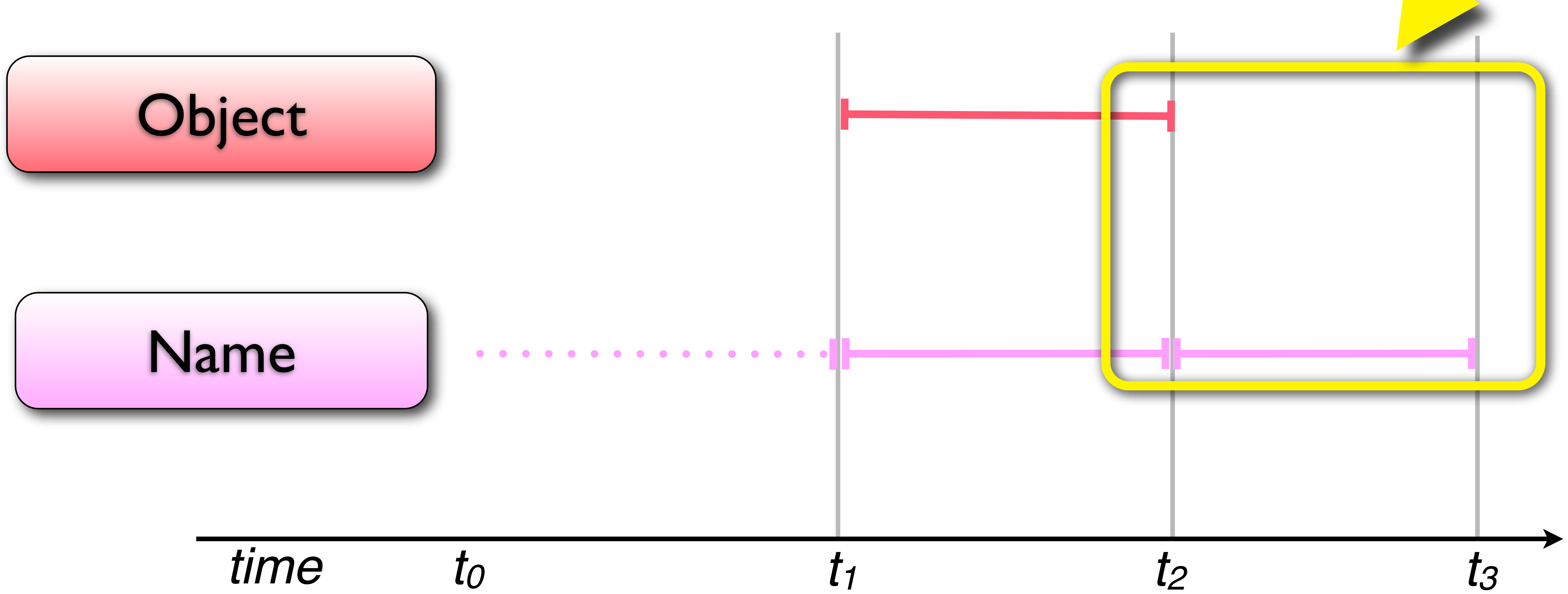
➡ **Binding exists long**

➡ Object de-allocated

➡ "Use-after-free bug"
(`free` is the C deallocation routine)

➡ "Dangling" pointer or reference.

Name bound, but object **no longer exists**.
Reference is "stale" and "dangles."

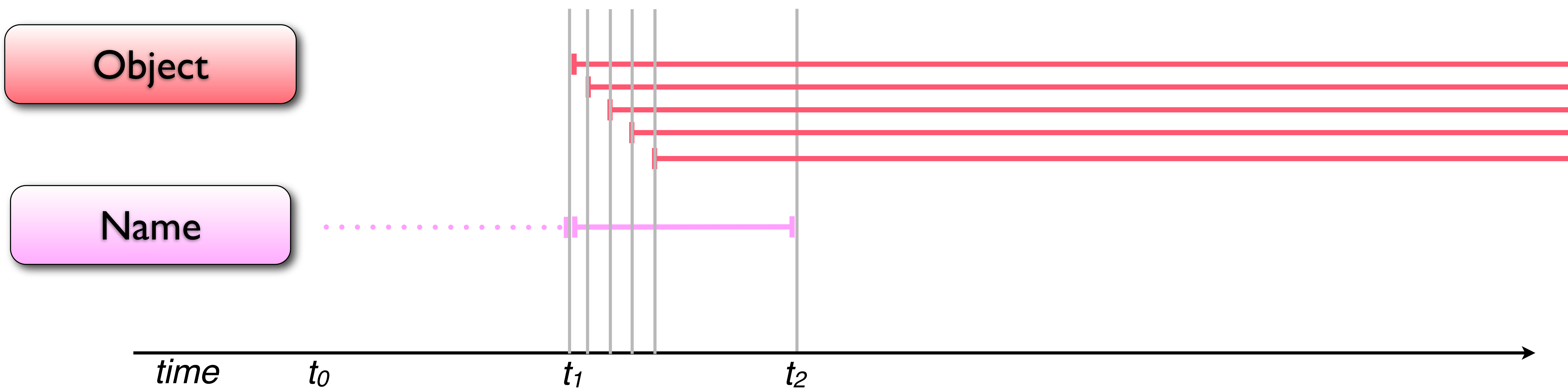


Tuesday, February 23, 2010

Memory "Leaks"

Omitted deallocation.

- ➡Objects that "**live forever**."
- ➡Even **if no longer required**.
  - Possibly no longer referenced.
- ➡**Waste memory**; can bring system down.
- ➡A problem in virtually every non-trivial project.



Memory "Leaks"

Omitted deall

➡Objects that

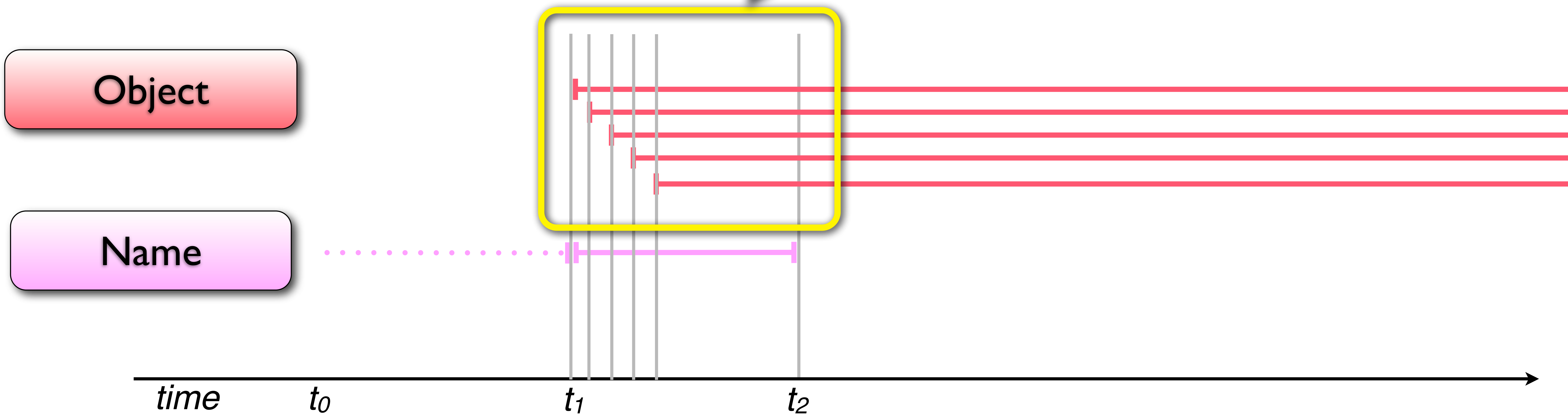
➡Even **if no l**

‣Possibly no longer referenced.

➡**Waste memory**; can bring system down.

➡A problem in virtually every non-trivial project.

Objects "forgotten" about, but "**stick around**" and waste space until program termination.



Garbage Collection

Manual deallocation is error-prone.

- ➡"Dangling references."
- ➡"**Use after free**."
- ➡Possibly unnecessarily **conservative**.
- ➡"Memory leaks."

Garbage collection.

- ➡**Automatically** deallocates objects **when it is safe to do so**.
- ➡Automated heap management; programmer can **focus on solving real problem**.
- ➡We will focus on garbage collection techniques later in the semester.

Summary & Advise

Static Allocation

Not dynamically sizable; lifetime spans virtually whole program execution; use only sparingly.

Stack Allocation

Lifetime restricted to subroutine invocation; allocation and deallocation is cheap; use whenever possible.

Heap Allocation

Arbitrary lifetimes; **use garbage collection whenever possible**; use for large buffers and long-living objects.