Lecture 7: Binding Time and Storage

COMP 524 Programming Language Concepts
Stephen Olivier
February 5, 2009

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts

Goal of Lecture

- The Goal of this lecture is to discuss **object binding** and **memory management**.

High-Level Programming Languages

- High-Level Programming languages are defined by two characteristics
  - Machine “independence”
  - Ease of programming

Machine "Independence"

- With few exceptions, the **code of a high-level language can be compiled on any system**
  - For example `cout << "hello world"<< endl;` means the same thing on any machine
- However, few languages are **completely machine independent**.
- Generally, the more machine dependent a language is the more "efficient" it is.

Ease of Programming

- Names
- Control Flow
- Types
- Subroutines
- Object Orientation
- Concurrency
- Declarative Programming

Naming

- Naming is the process by which a programer associates a name with a potentially complicated program fragment.
  - Purpose is to hide complexity.
  - For example, to designate variables, types, classes, operators, etc ...

Naming

- Naming is the process by which a programer associates a name with a potentially complicated program fragment.
  - Purpose is to hide
  - For example, to d s, classes, operators, etc ...

By naming an object we make **an abstraction**

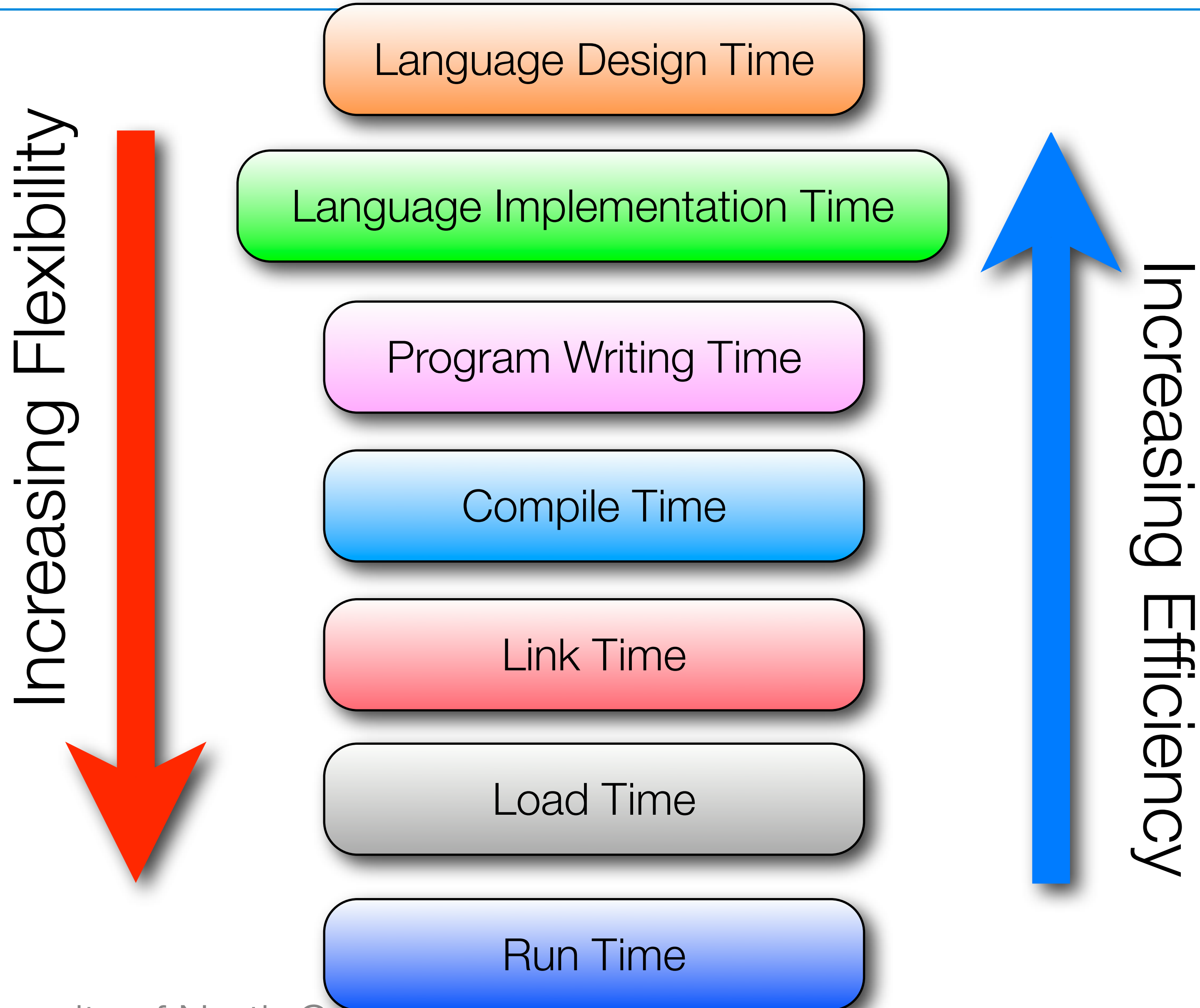
Abstractions

- **Control abstractions** allows programs to “hide” complex code behind simple interface
  - Subroutines and functions
  - Classes.
- **Data abstraction** allow the programer to hide data representation details behind abstract operations
  - Abstract Data Types (ADTs)
  - Classes.

Binding Time

- A **binding** is an association between **any two things**
  - Name of an object and the object.
  - A Dook student to a loosing basketball team.
- **Binding Time** is the time at which a **binding is created**.

Binding Time



Object Lifetime

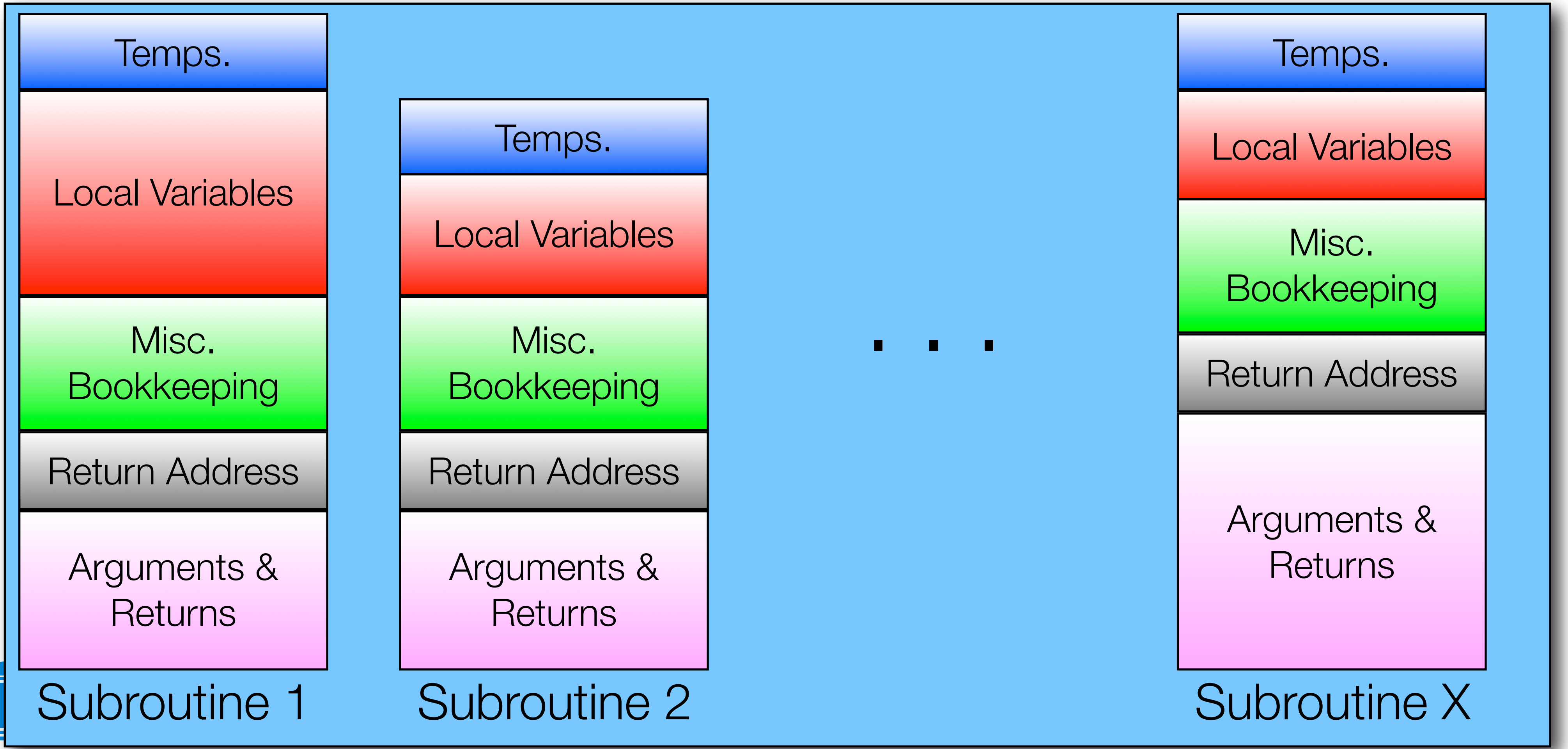
- Object lifetimes have two components
  - Lifetime of the object.
  - Lifetime of the binding.
- These two don’t necessarily correspond.
  - For example in C++, when a variable is passed by “reference”, i.e., using “&”, then the name of the object does not exist even though the binding does.
  - For example in C++, when the value pointed to by an object is deleted the binding is gone before the object.

Object Lifetimes

- Object Lifetimes correspond to three principal storage allocation mechanisms,
  - **Static** objects, which have an **absolute address**
  - **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**.

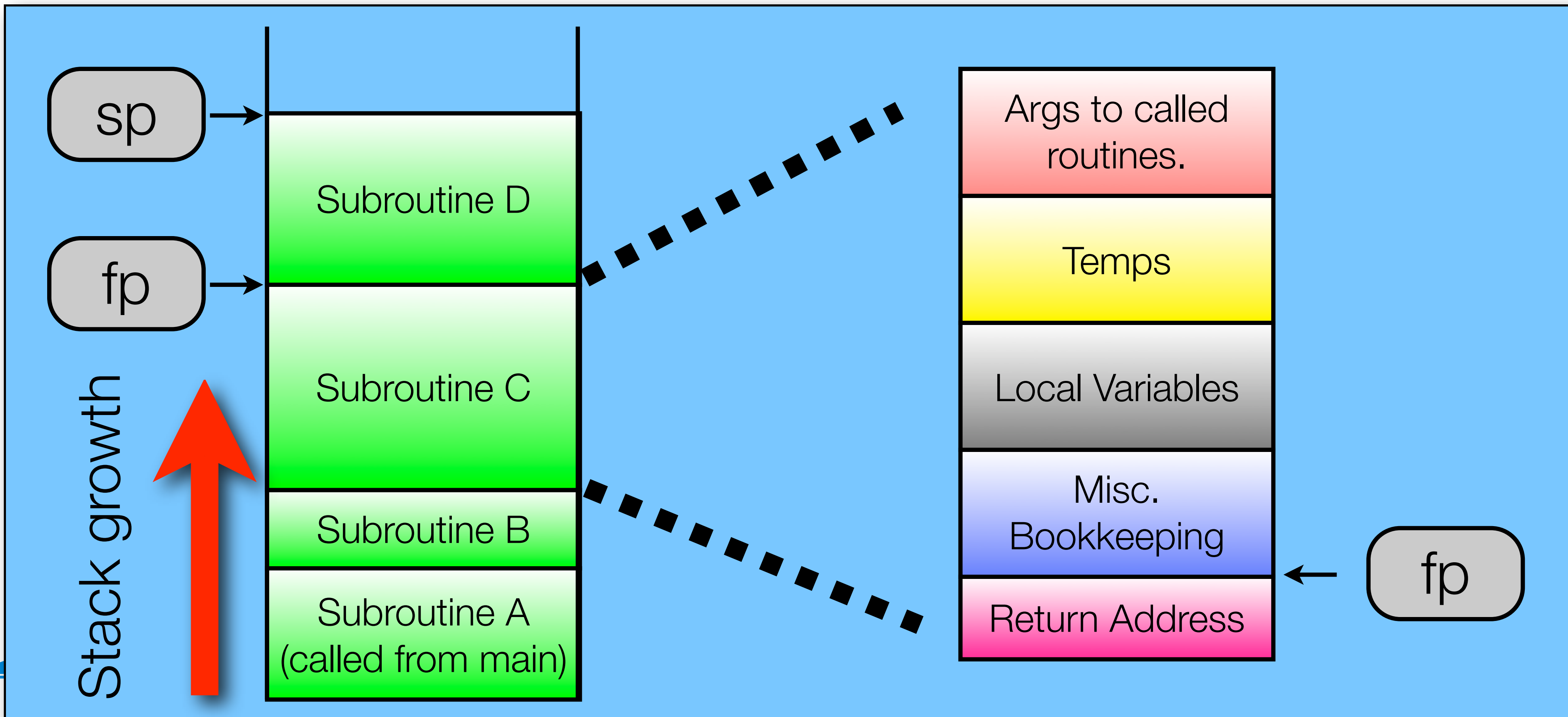
Static Allocation

- Under **static allocation**, objects are given an absolute address that is retained through the program's execution
  - *e.g.*, global variables



Stack Allocation

- Under **stack-based allocation**, objects are allocated in a **Last-In First-Out** (**LIFO**) basis called a stack.
  - e.g., recursive subroutine parameters.

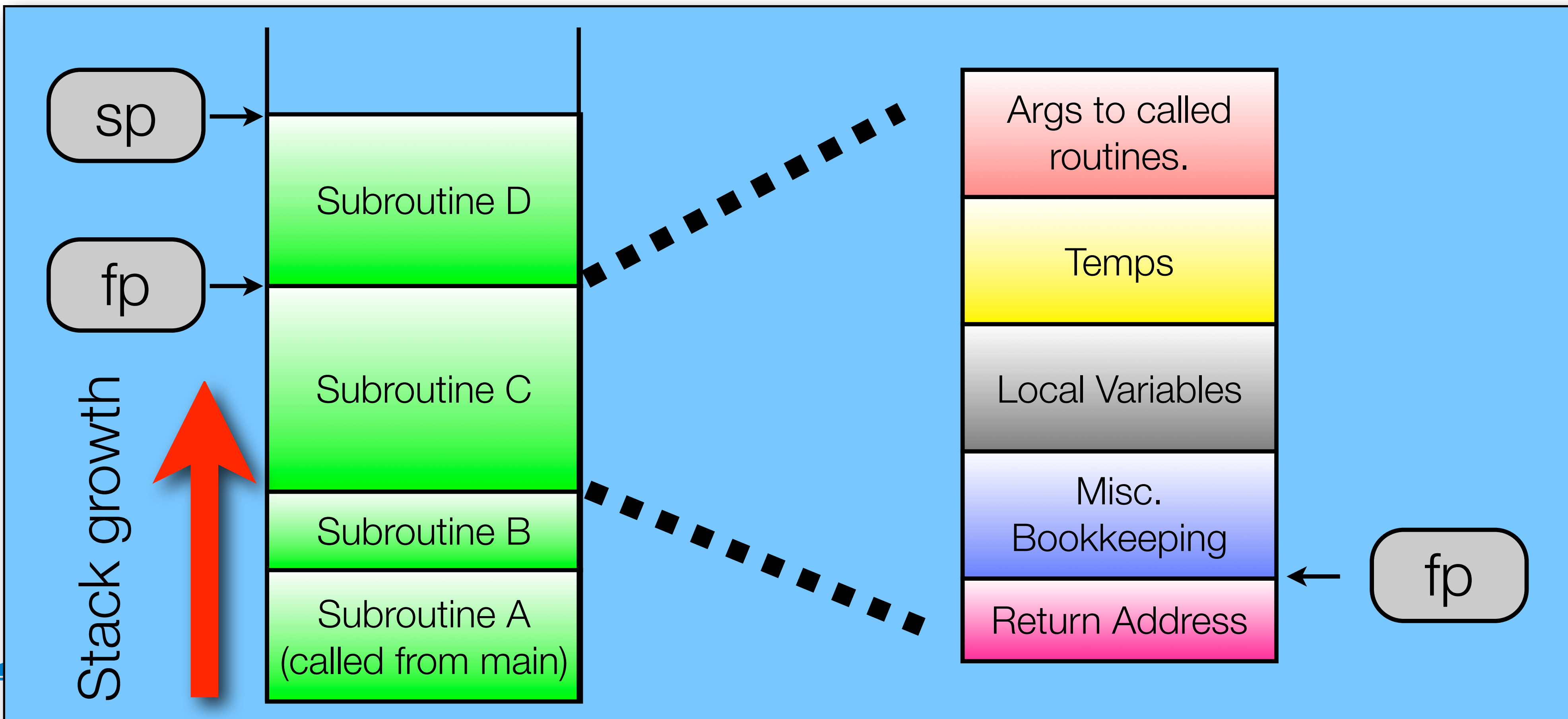


Stack Allocation

fp is the "**Frame Pointer**"

sp is the "**Stack Pointer**"

- Under **s**… allocated in a **Last-In First-Out** (**LIFO**) basis called a stack.
  - e.g., recursive subroutine parameters.



Calling Sequence

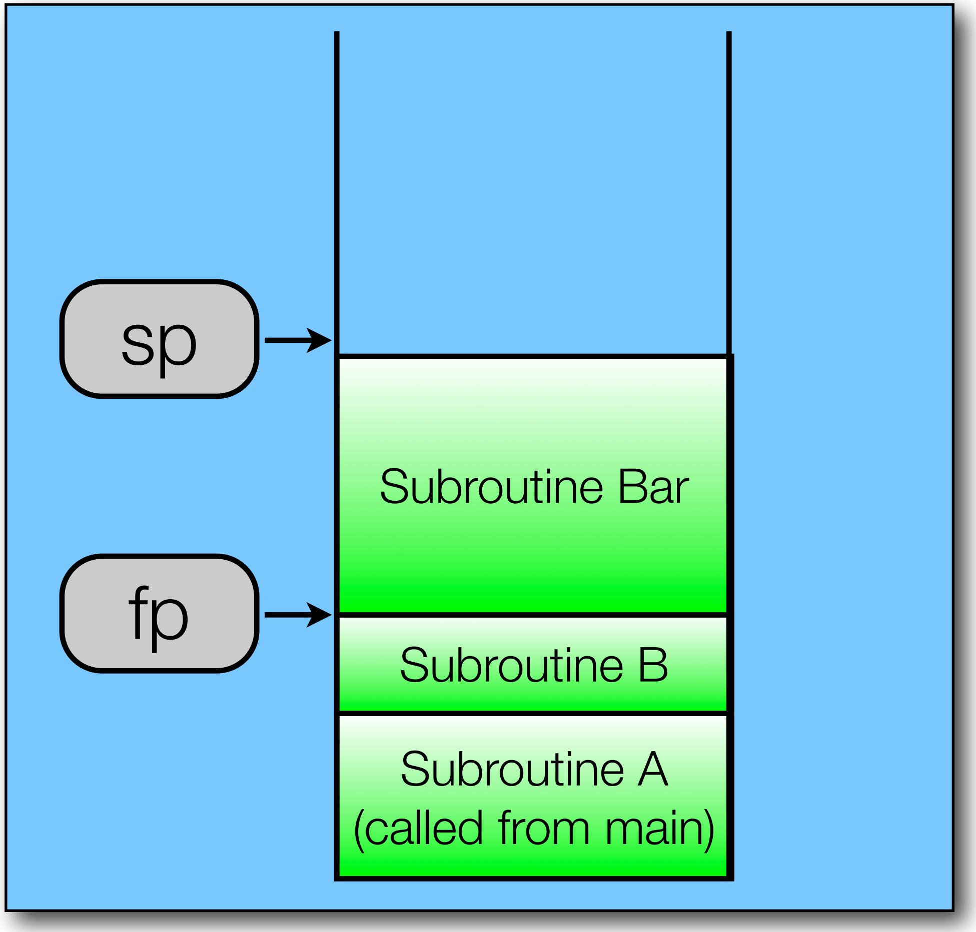
- On procedure call and return compilers generate code that execute to manage the runtime stack.
  - **Setup** at call to procedure foo(a,b).
  - **Prologue** before foo code executes.
  - **Epilogue** at the end of foo code.
  - **“Teardown”** right after calling the code.

Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** a,b into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address

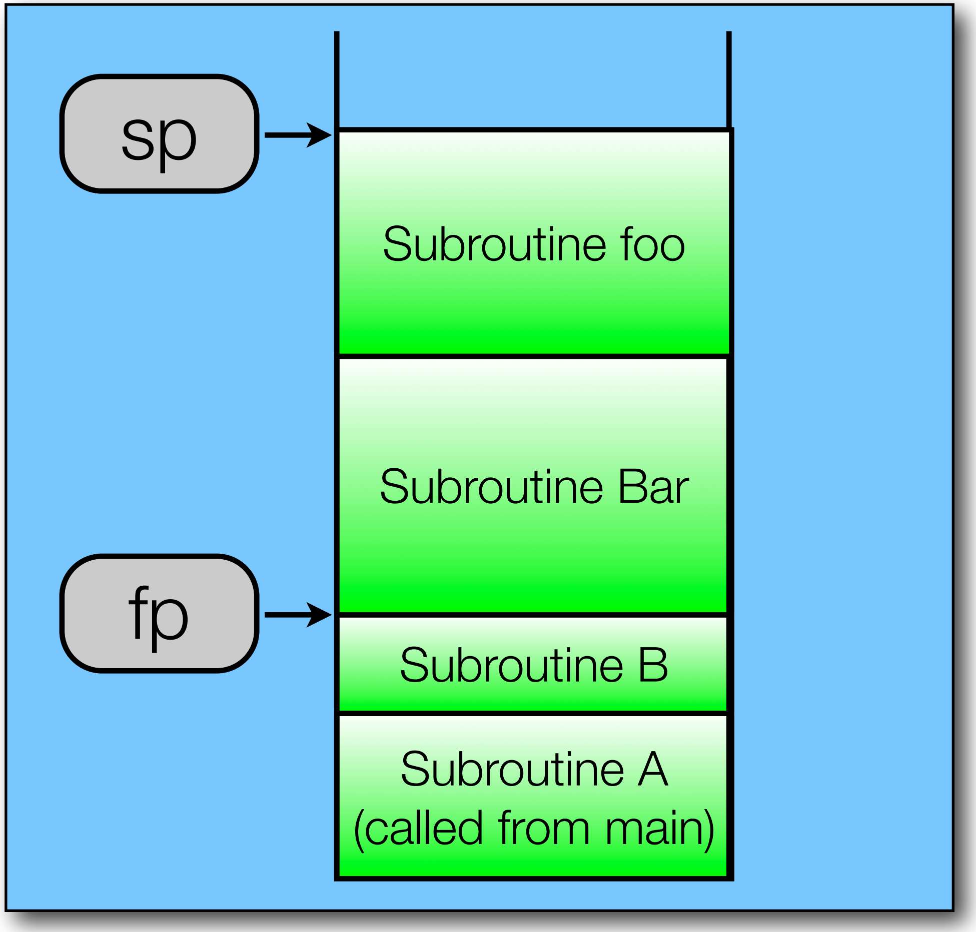
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



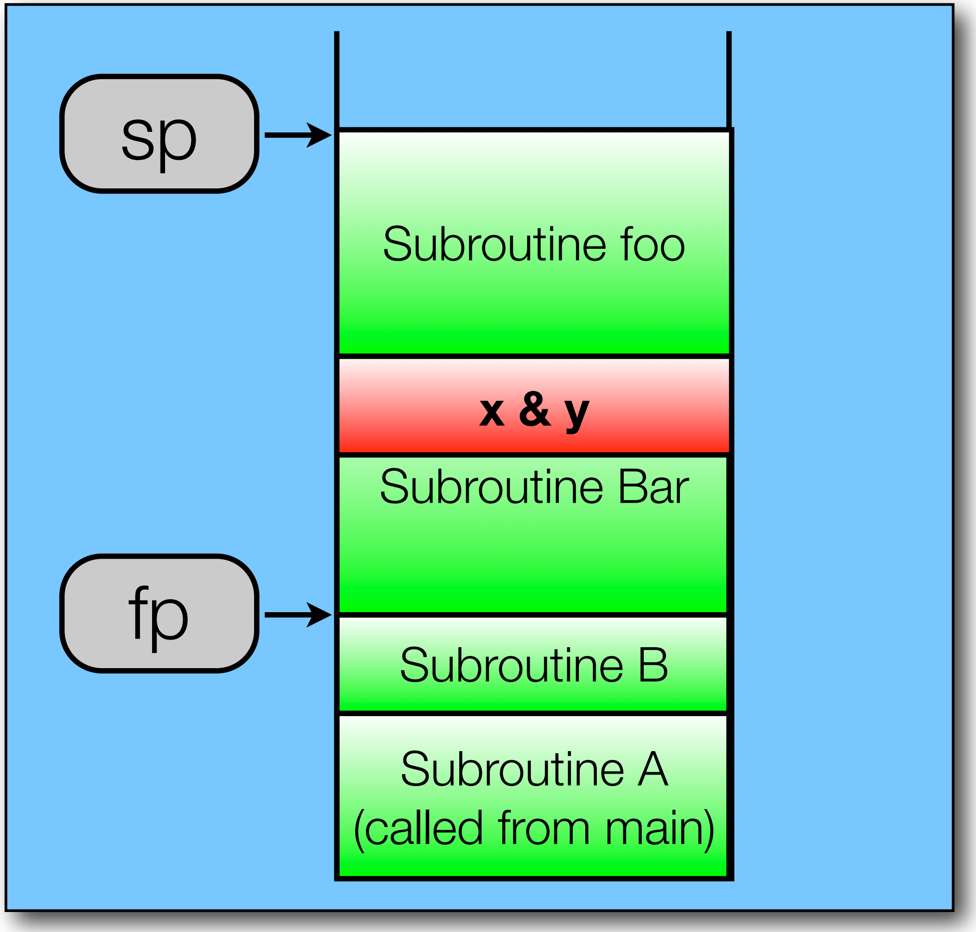
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



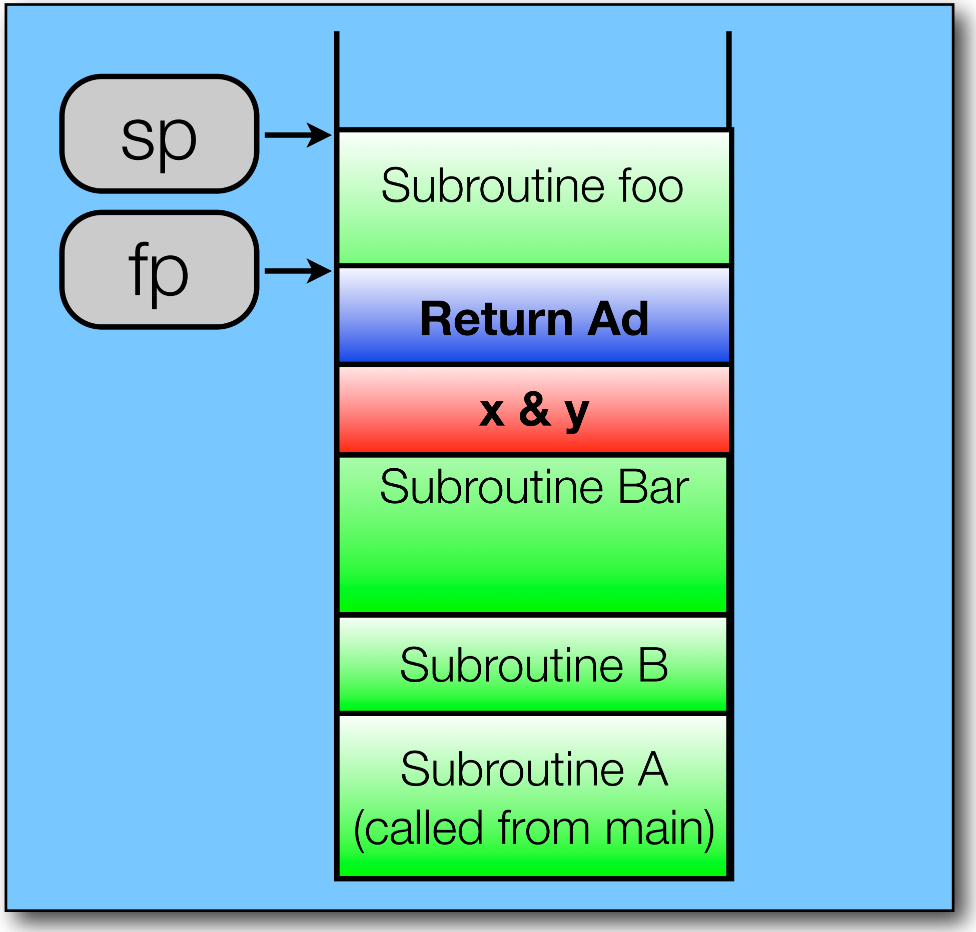
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



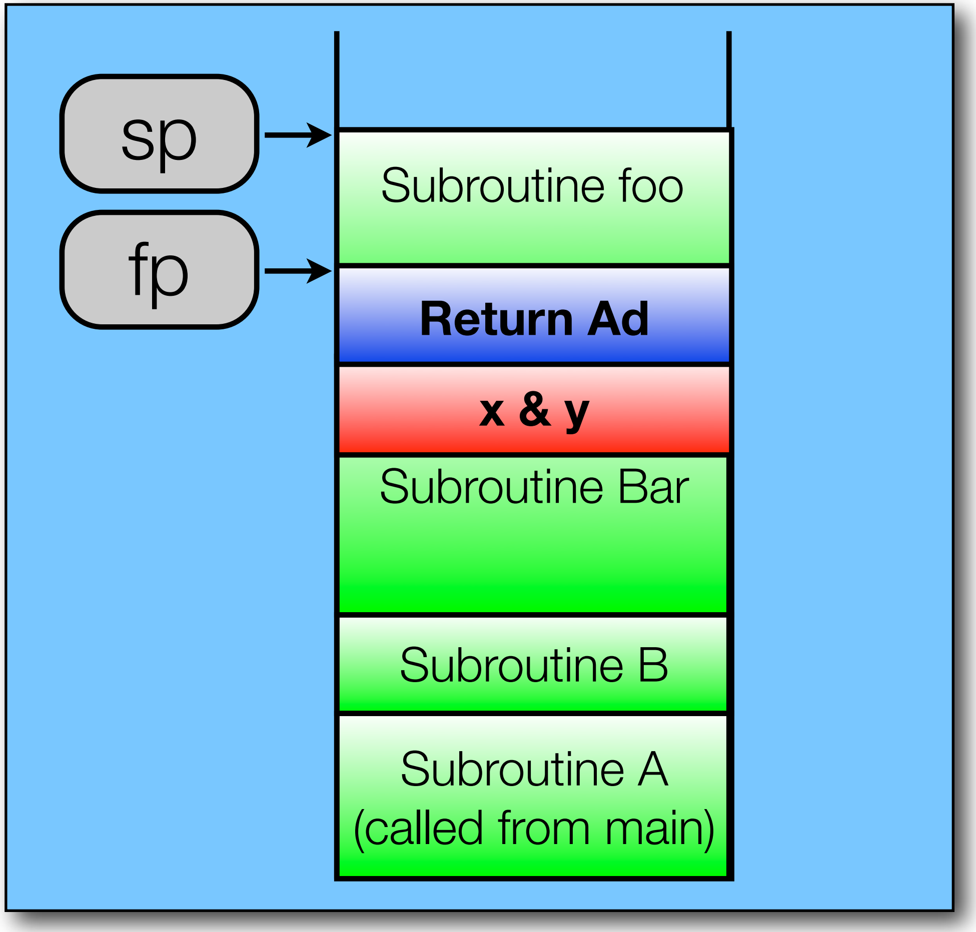
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



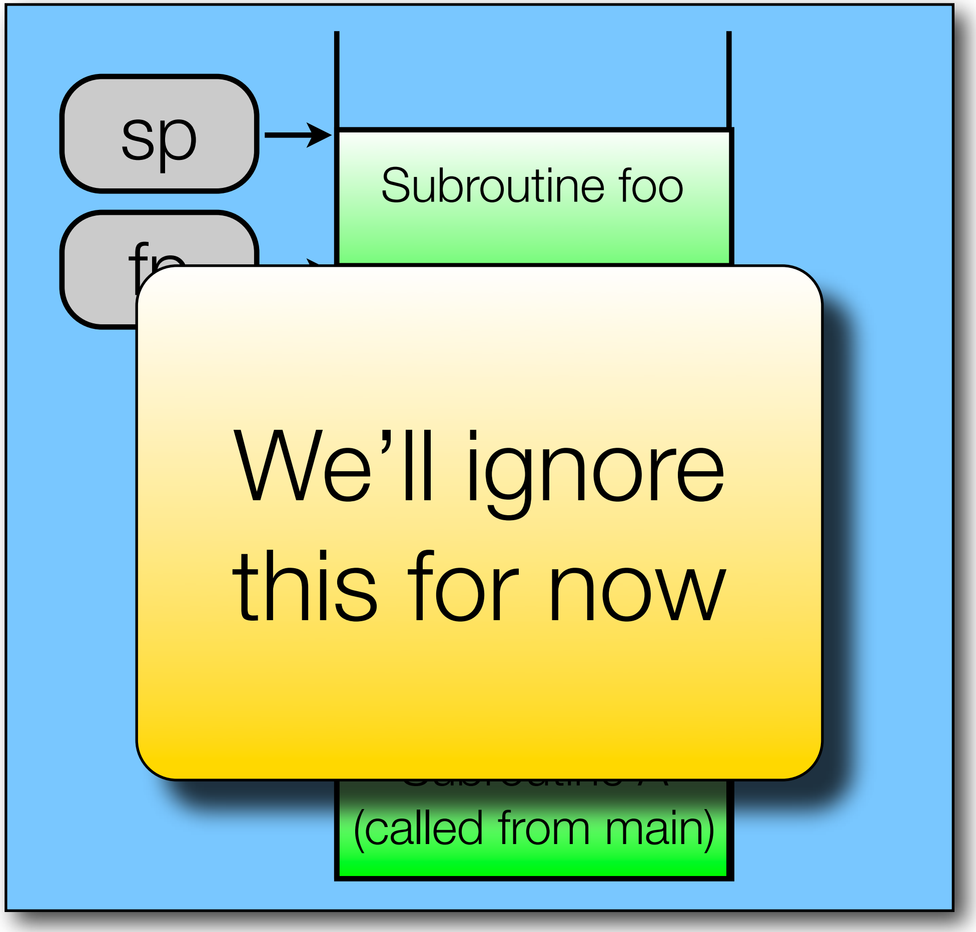
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



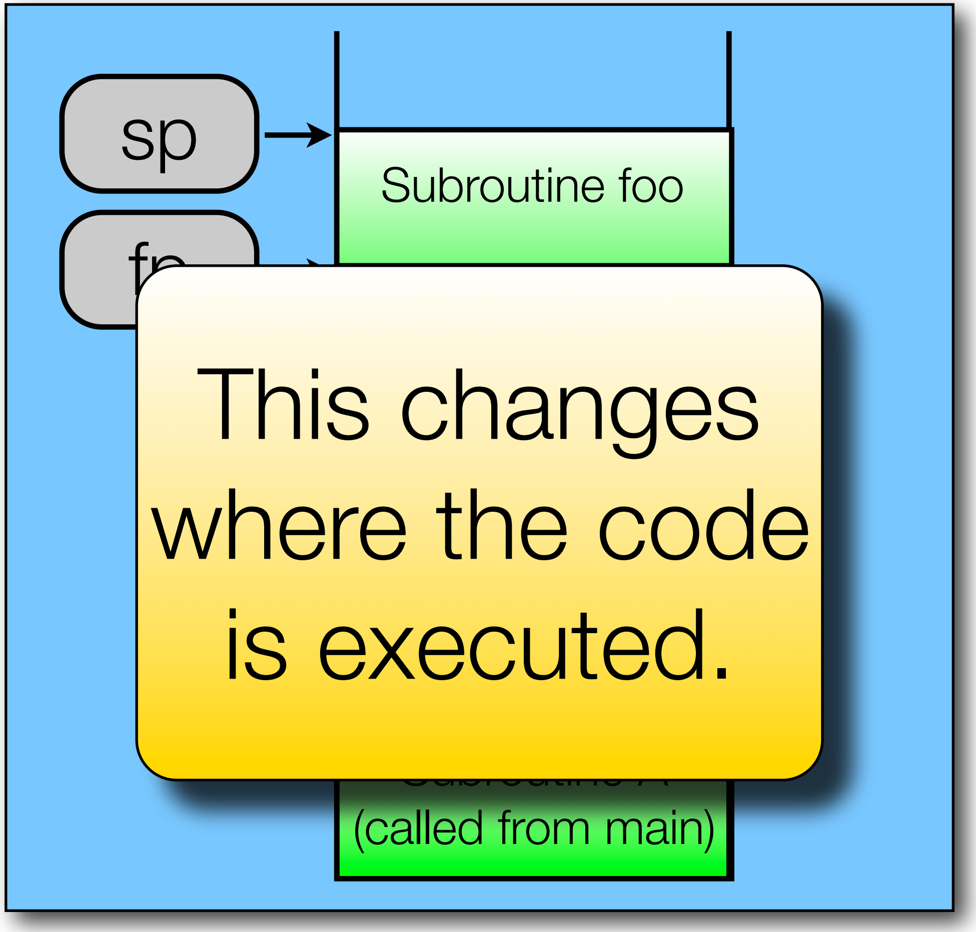
Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address



Setup *foo(a,b)*

- **Move sp** to allocate a new stack frame
- **Copy args** x,y into frame
- **Copy return** address into frame
- **Set fp** to point to new frame
- **Maintain static chain** or display
- **Move PC** to procedure address

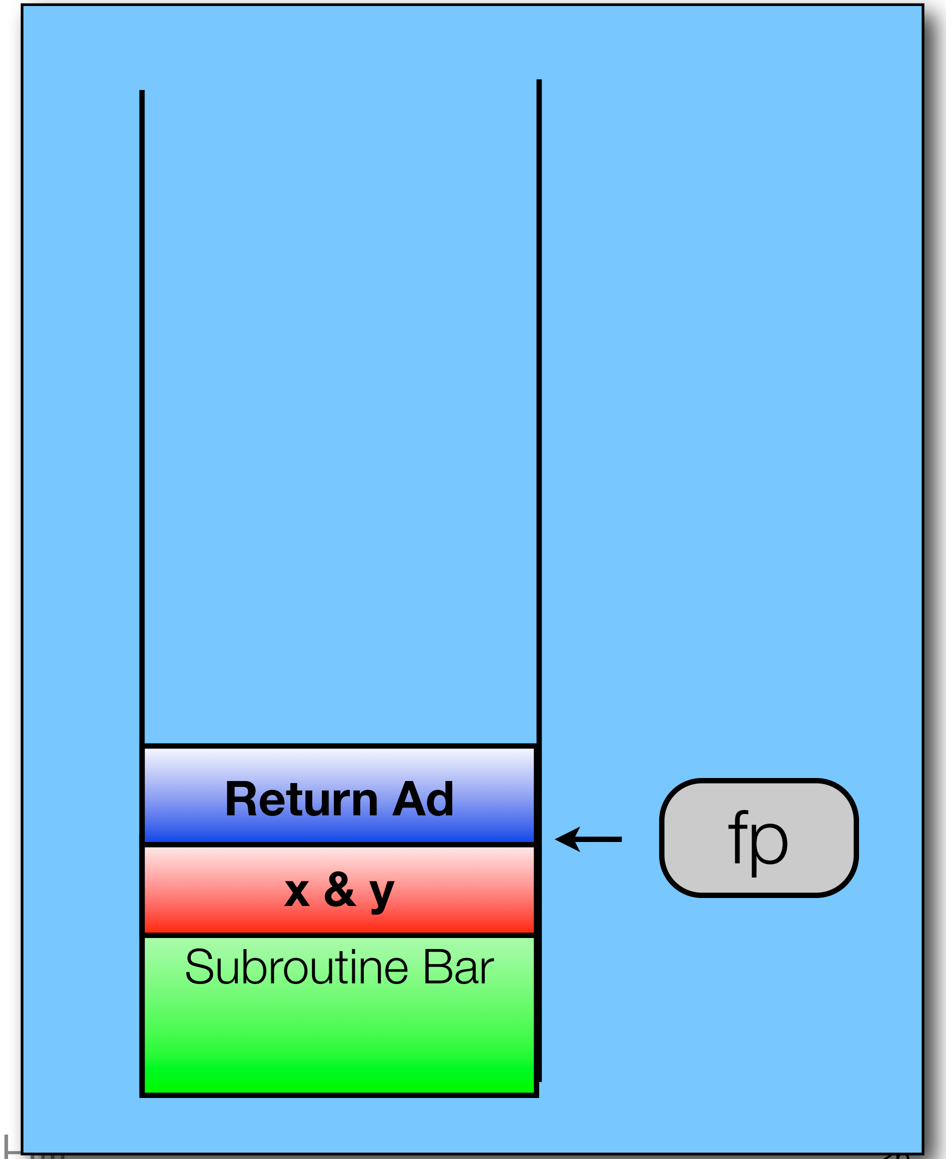


Prologue

- **Copy registers** into local slots
- **Object initialization**.

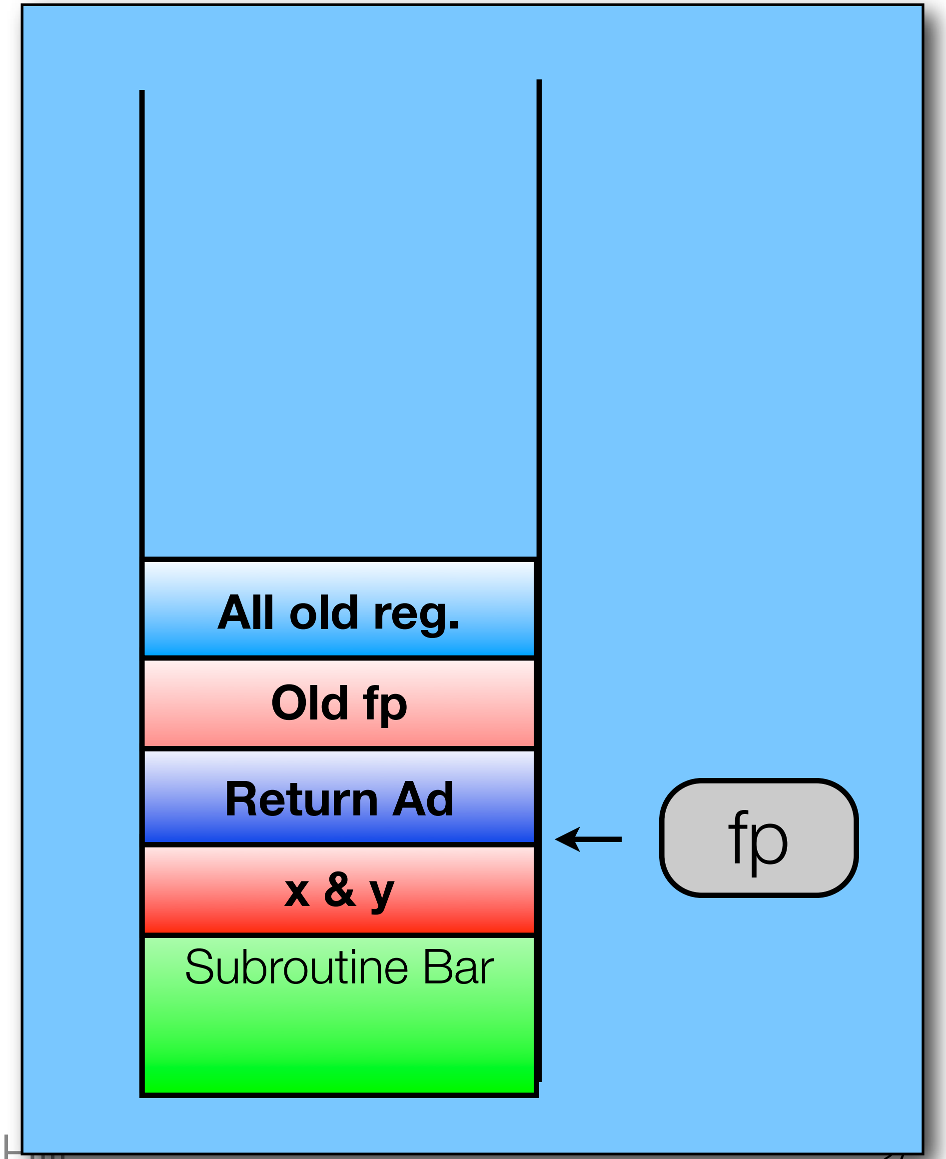
Prologue

- **Copy registers** into local slots
- **Object initialization**.



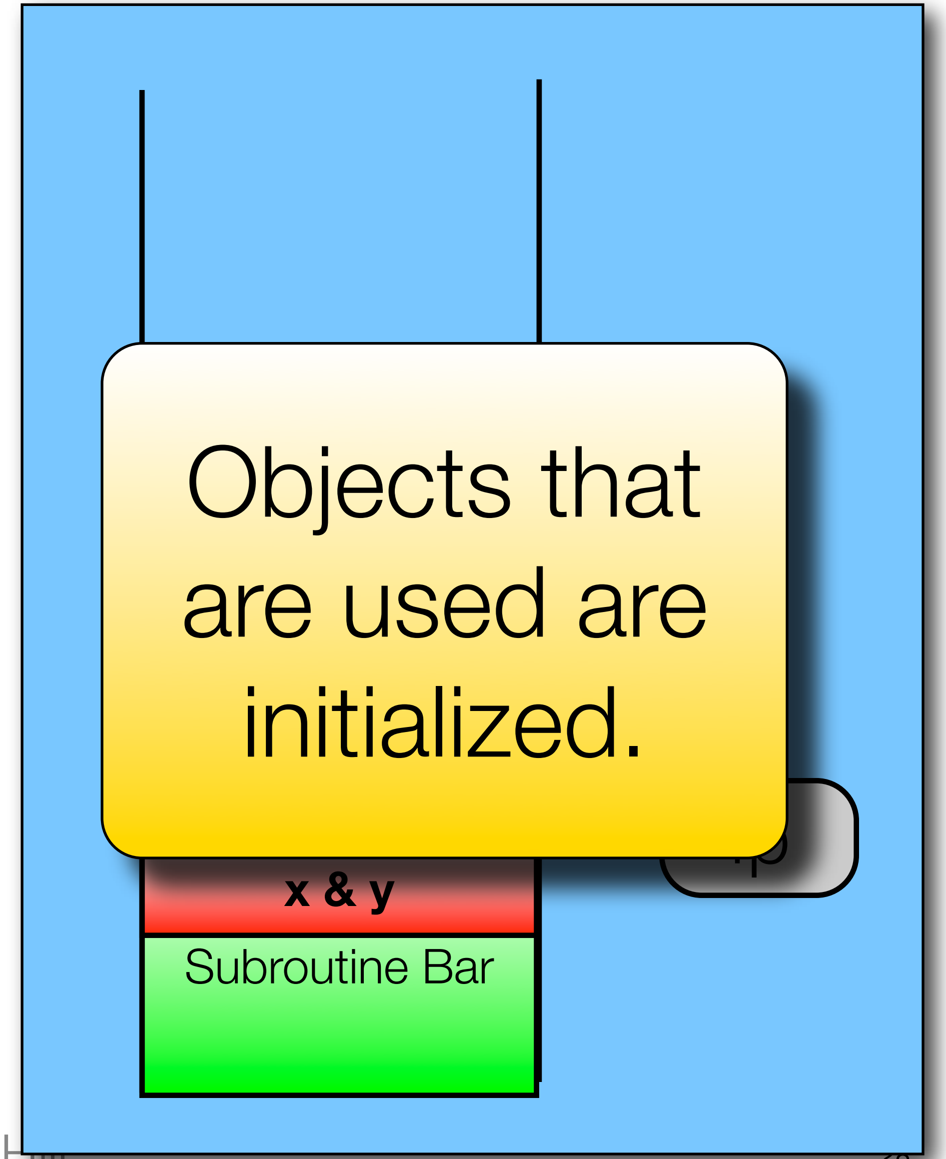
Prologue

- **Copy registers** into local slots
- **Object initialization.**



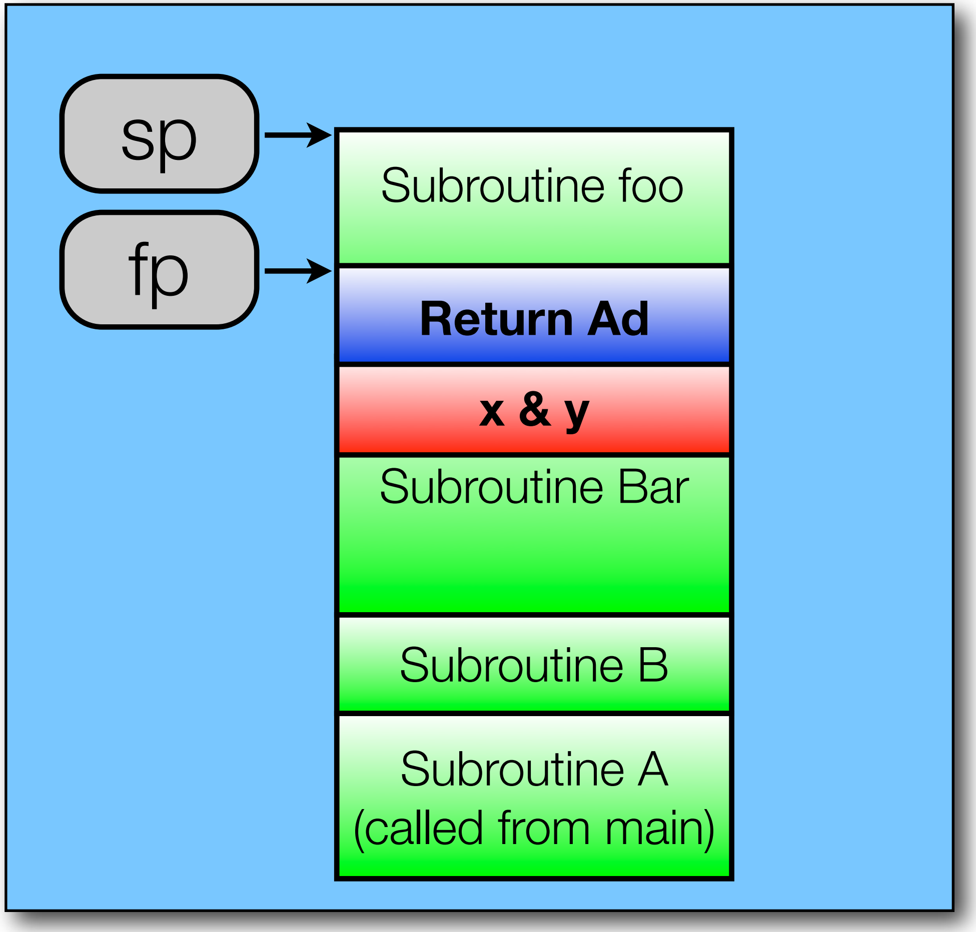
Prologue

- **Copy registers** into local slots
- **Object initialization**.



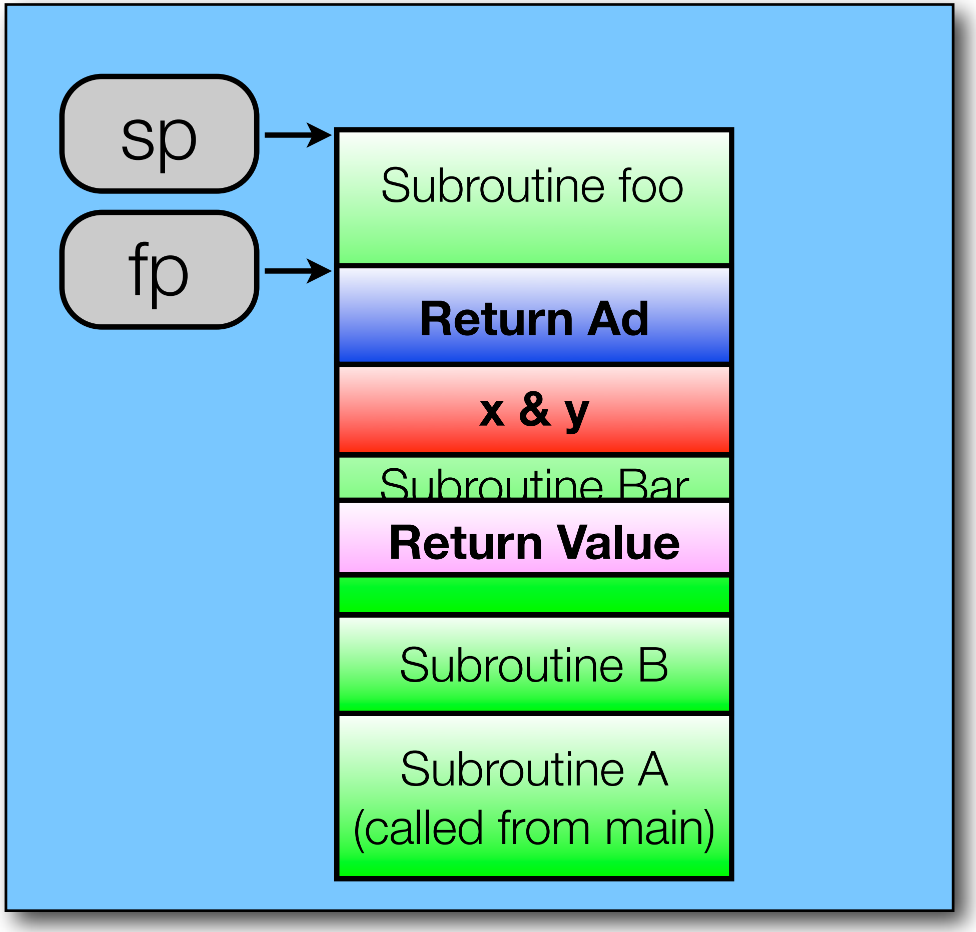
Epilogue

- **Place return value** into slot in frame.
- **Restore registers**.
- **Restore PC** to return address.



Epilogue

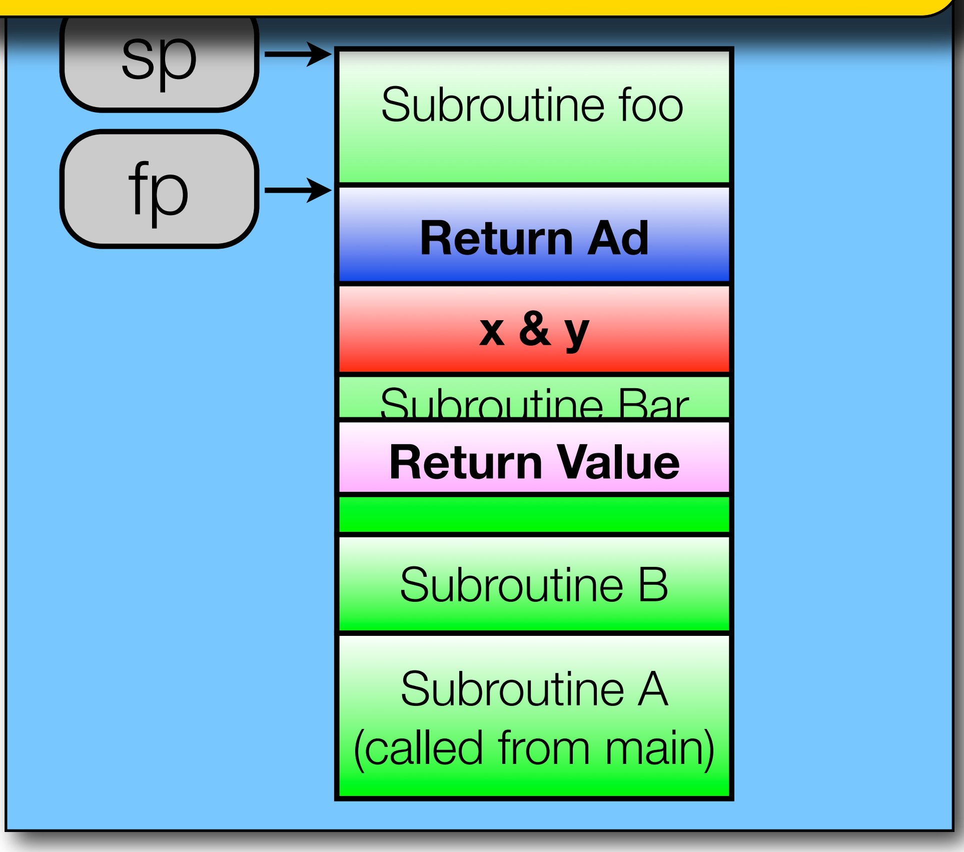
- **Place return value** into slot in frame.
- **Restore registers.**
- **Restore PC** to return address.



Epilogue

- **Place return value** into slo in frame.
- **Restore registers**.
- **Restore PC** to return address.

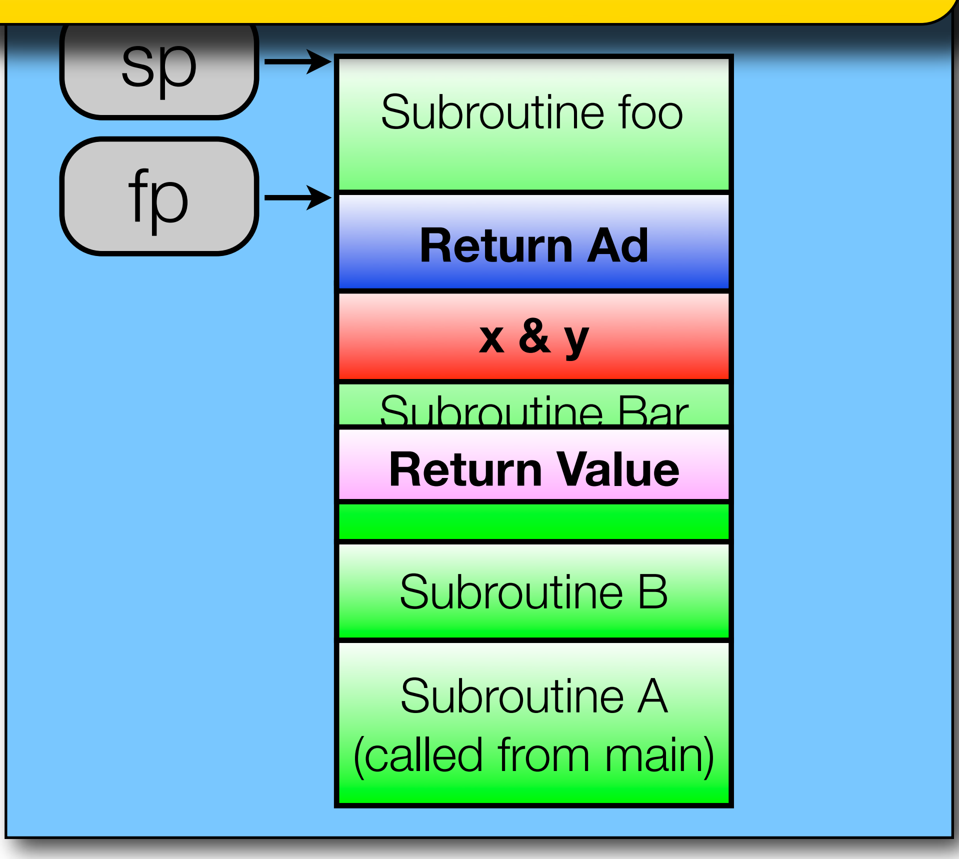
Registers stored from "foo"'s subroutine are registered.



Epilogue

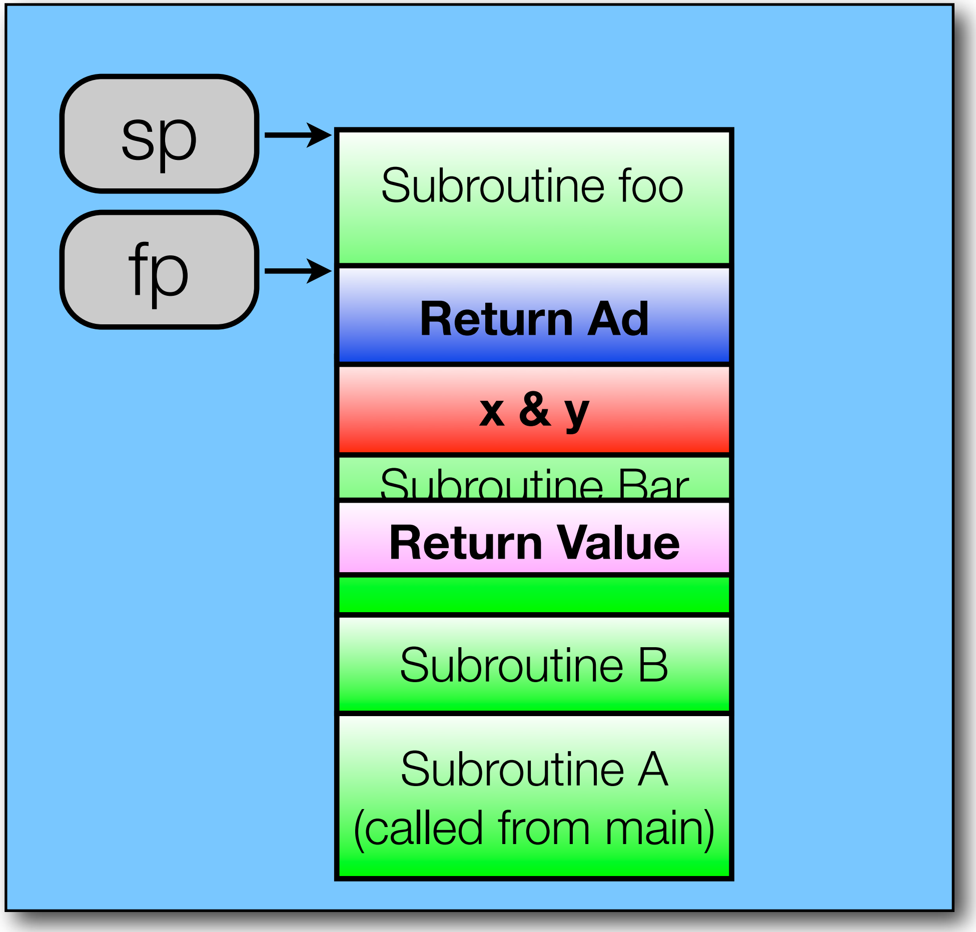
- **Place return value** into slo
  in frame.
- **Restore registers.**
- **Restore PC** to return address.

The program resumes from where it began.



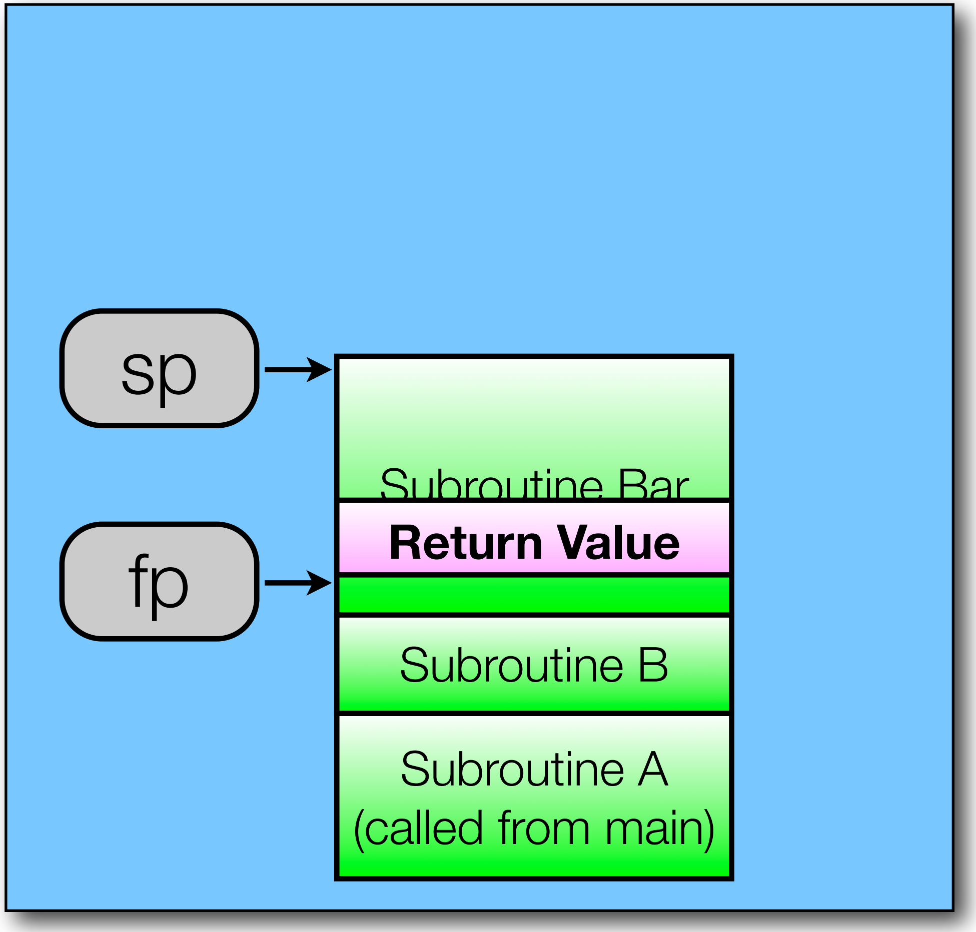
"Teardown"

- **Move sp & fp** (deallocate frame)
- **Move return** values (if in registers)



“Teardown”

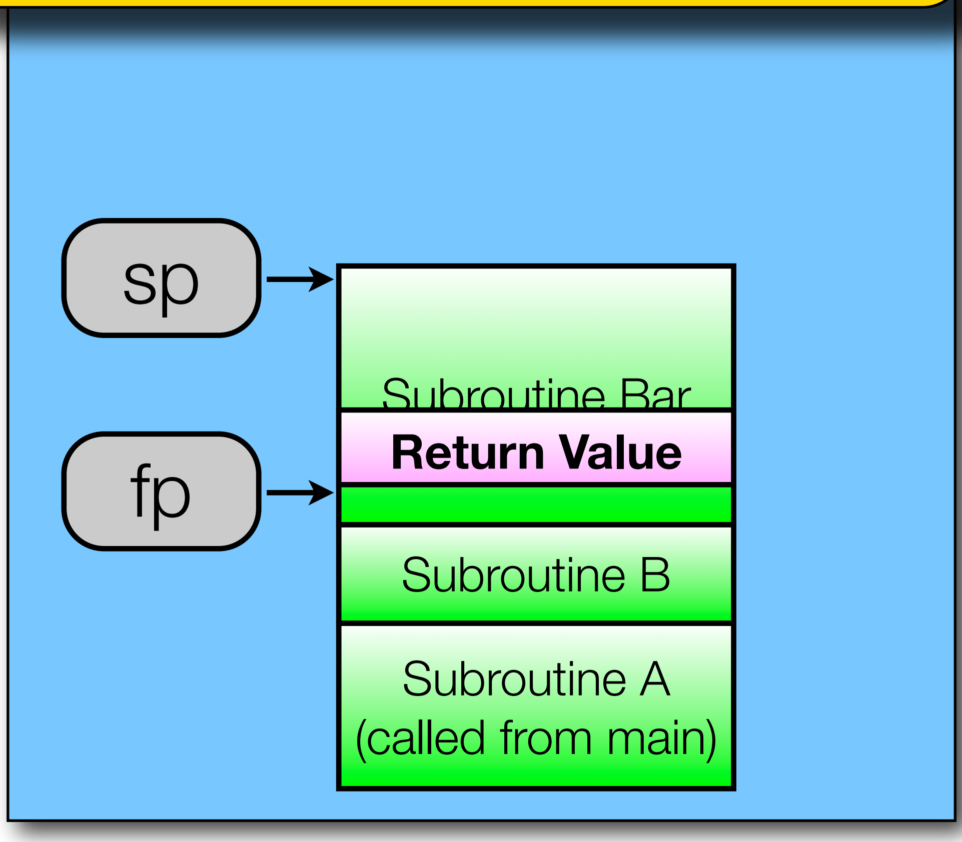
- **Move sp & fp** (deallocate frame)
- **Move return** values (if in registers)



“Teardown”

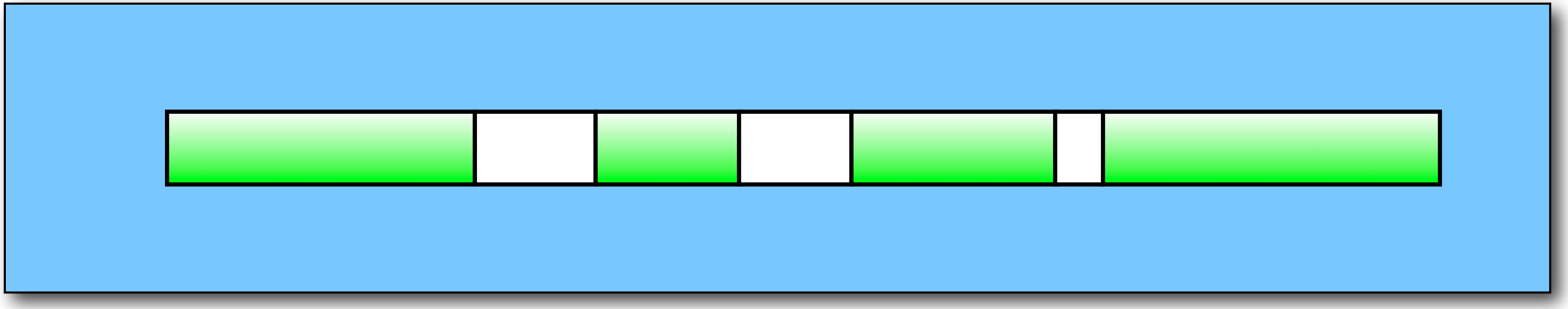
- **Move sp & fp** (deallocate frame)
- **Move return** values (if in registers)

If the return value was placed in a register, put it in the stack.



Heap-based allocation

- In heap-based allocation, **objects may be allocated and deallocated at arbitrary times**.
  - For example, objects created with C++ new and delete.

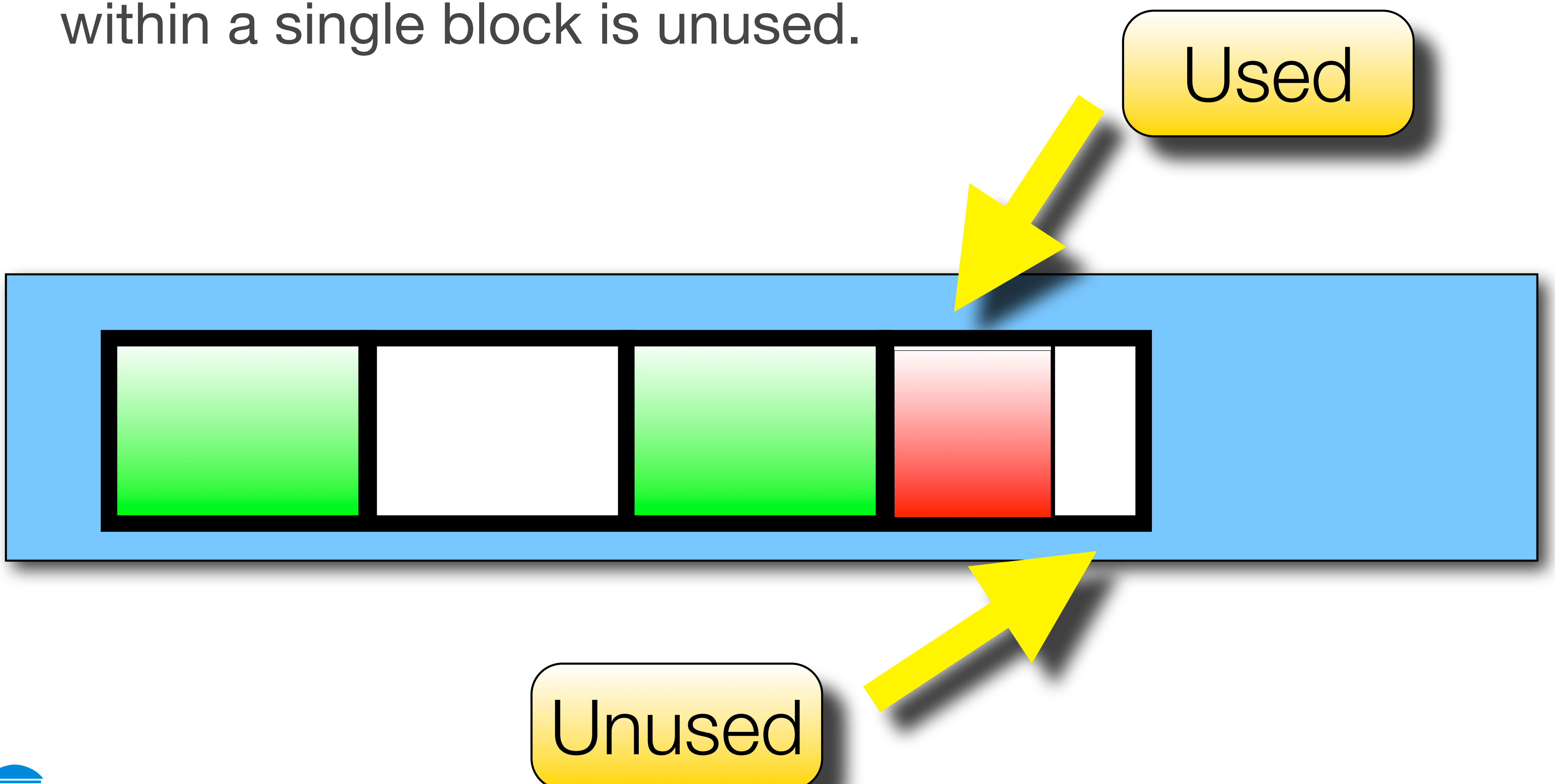


Heap Space Management

- In general, the heap is allocated sequentially.
- This creates fragmentation...

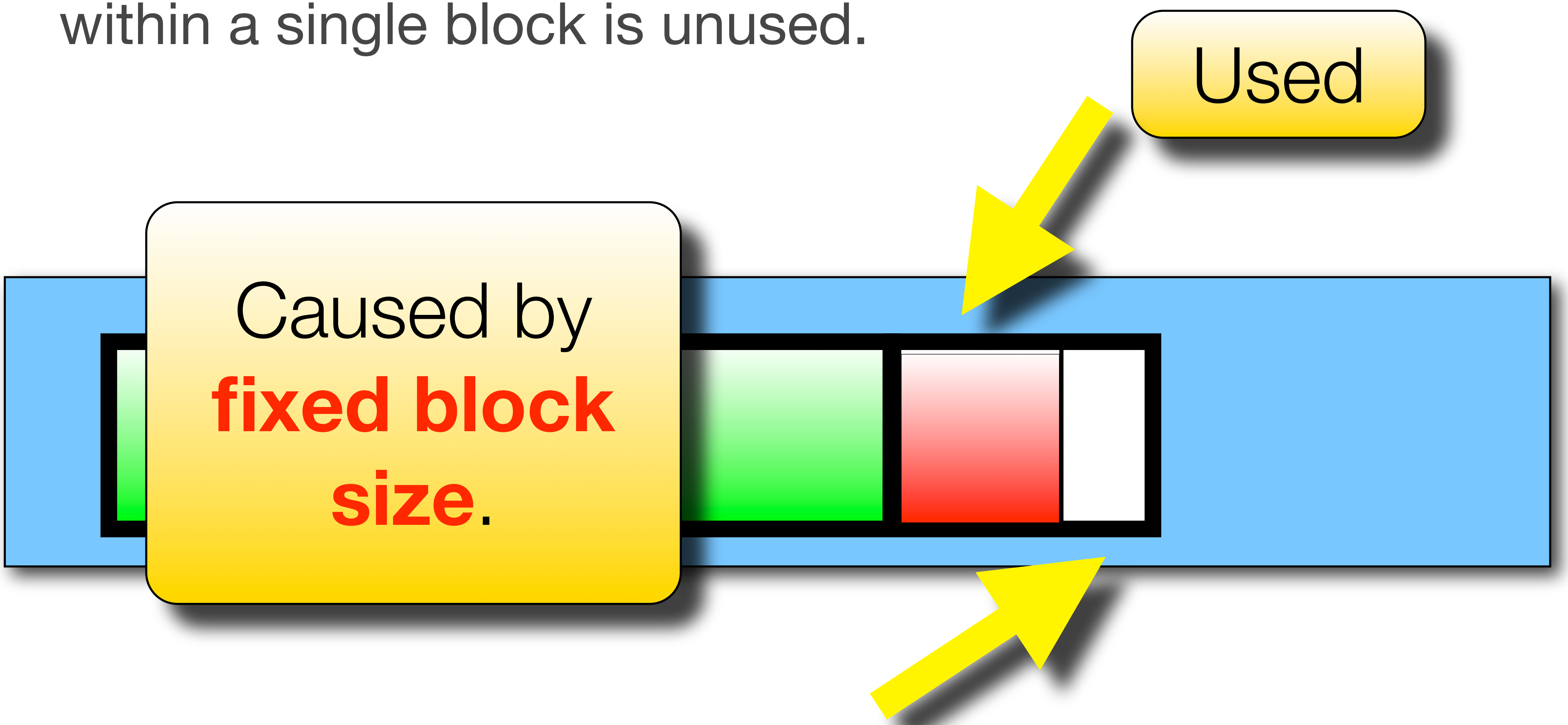
Internal fragmentation

- Internal fragmentation is caused when extra space within a single block is unused.



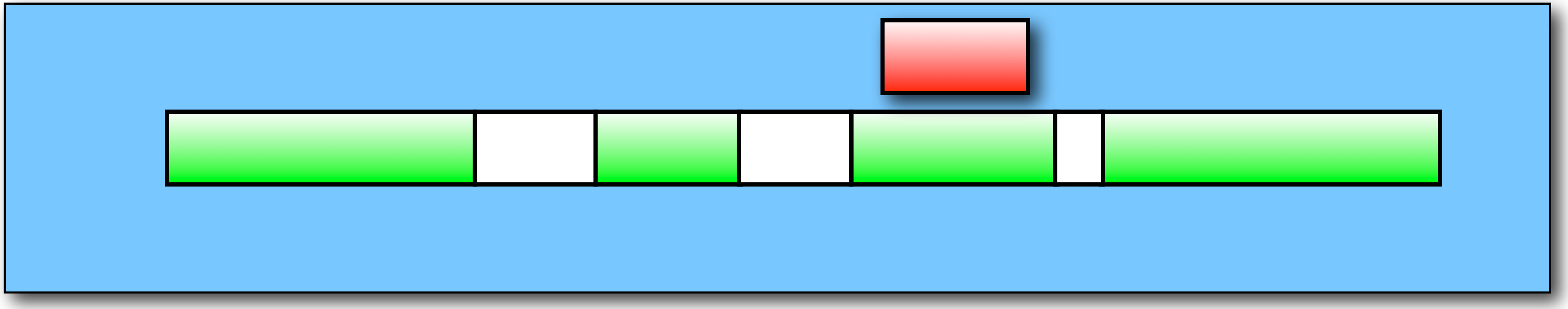
Internal fragmentation

- Internal fragmentation is caused when extra space within a single block is unused.



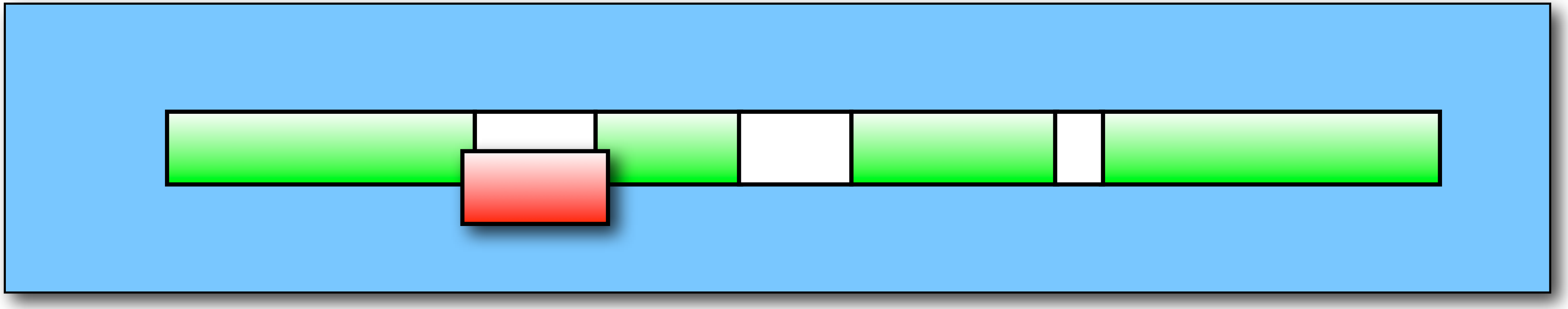
External Fragmentation

- **External fragmentation** occurs when there is **sufficient available space** for a new object, but there is **no single block of free space large enough**.



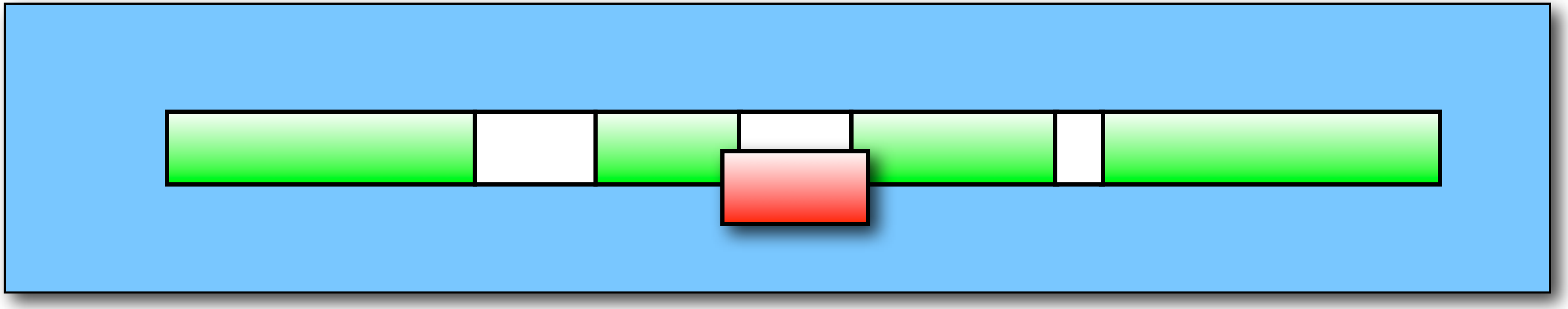
External Fragmentation

- **External fragmentation** occurs when there is **sufficient available space** for a new object, but there is **no single block of free space large enough**.



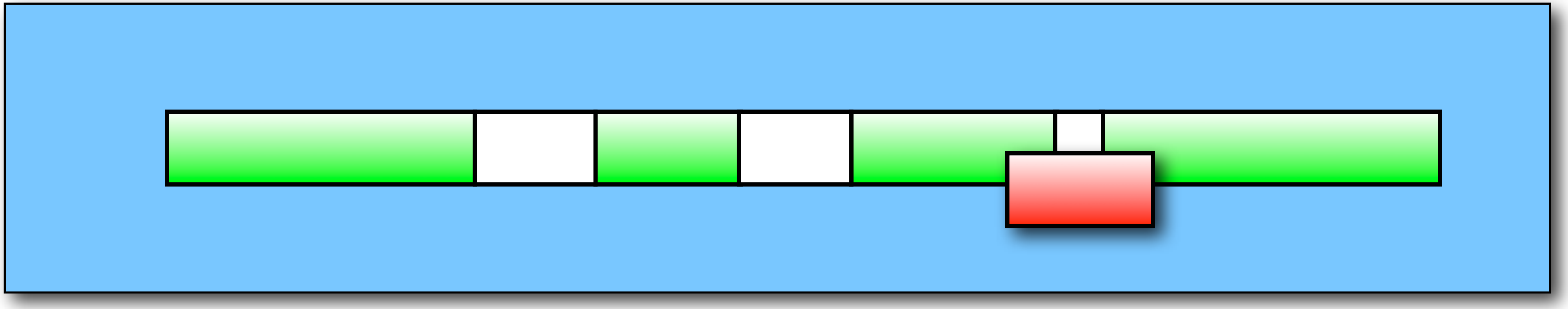
External Fragmentation

- **External fragmentation** occurs when there is **sufficient available space** for a new object, but there is **no single block of free space large enough**.



External Fragmentation

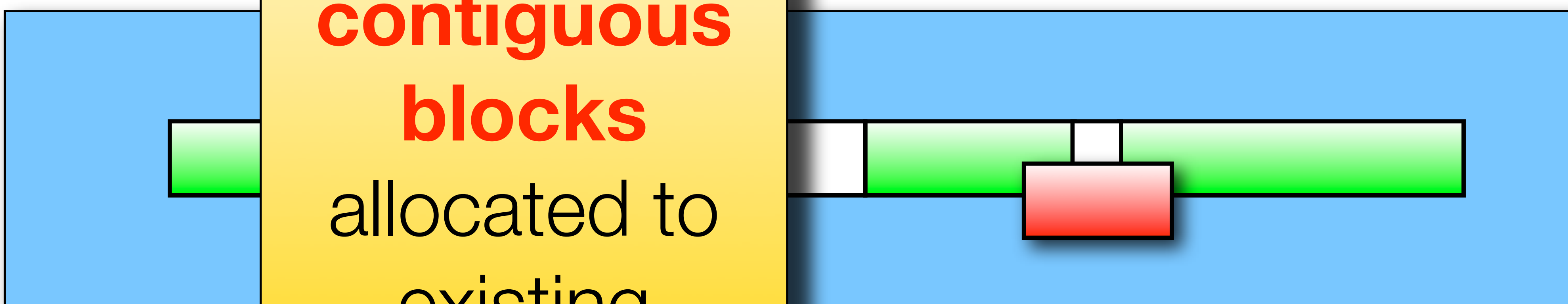
- **External fragmentation** occurs when there is **sufficient available space** for a new object, but there is **no single block of free space large enough**.



External Fragmentation

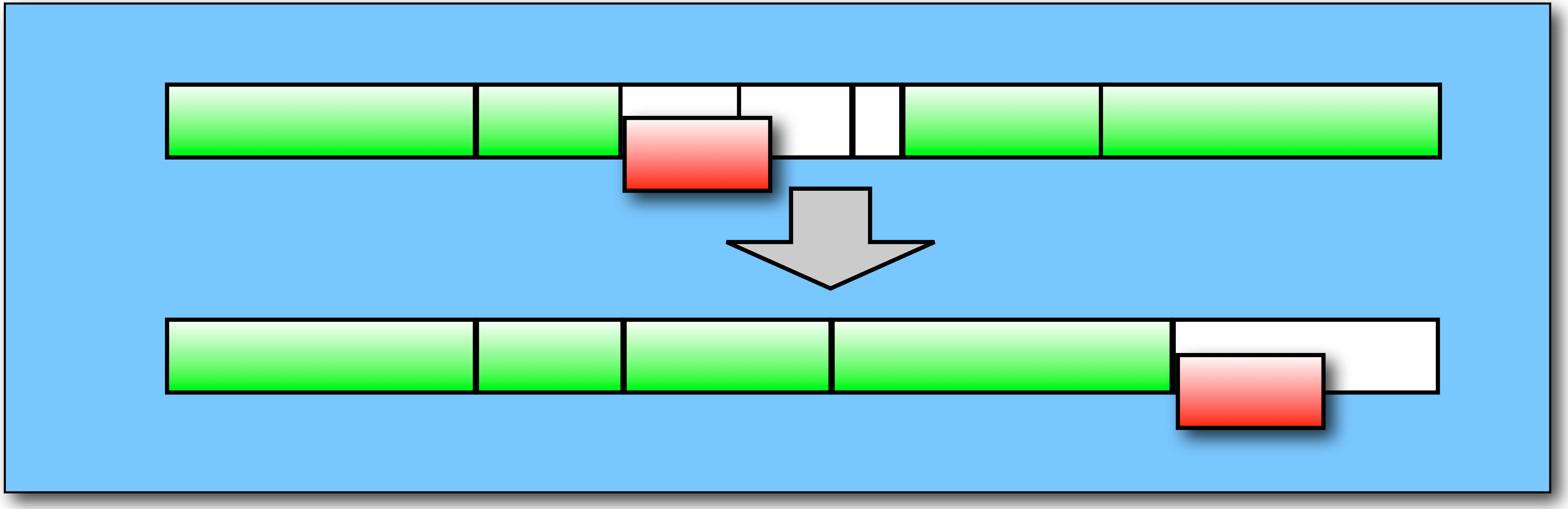
- **Externa** ccurs when there is **sufficient availab** bject, but there is **no single block o** **enough**.

Caused by **gaps between contiguous blocks** allocated to existing objects.



External Fragmentation

- **May require heap compaction**
  - Combine in the heap by moving existing objects (expensive)
  - Similar to defragmentation of a hard drive



Heap Management

- Some languages (C & C++) require **explicit heap** **management...**
  - In C, malloc and free
  - In C++, new and delete
- Easy to forget free...
  - Called a memory leak!

Heap Management

- Some languages (Java) **manage the heap** for you
  - new( ) object allocated on heap.
  - when done, object is reclaimed.
- Automatic de-allocation after an object has no binding/ references is called **garbage collection**.
  - Some runtime efficiency hit
  - No memory leaks.

Sample Memory Layout