Run-time organization
and
General Principles of Code Generation

Lecture 12

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Intermediate Code Generation
  - Optimization
  - Code generation
- We'll do code generation first . . .

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate

- There are a number of standard techniques for structuring executable code that are widely used

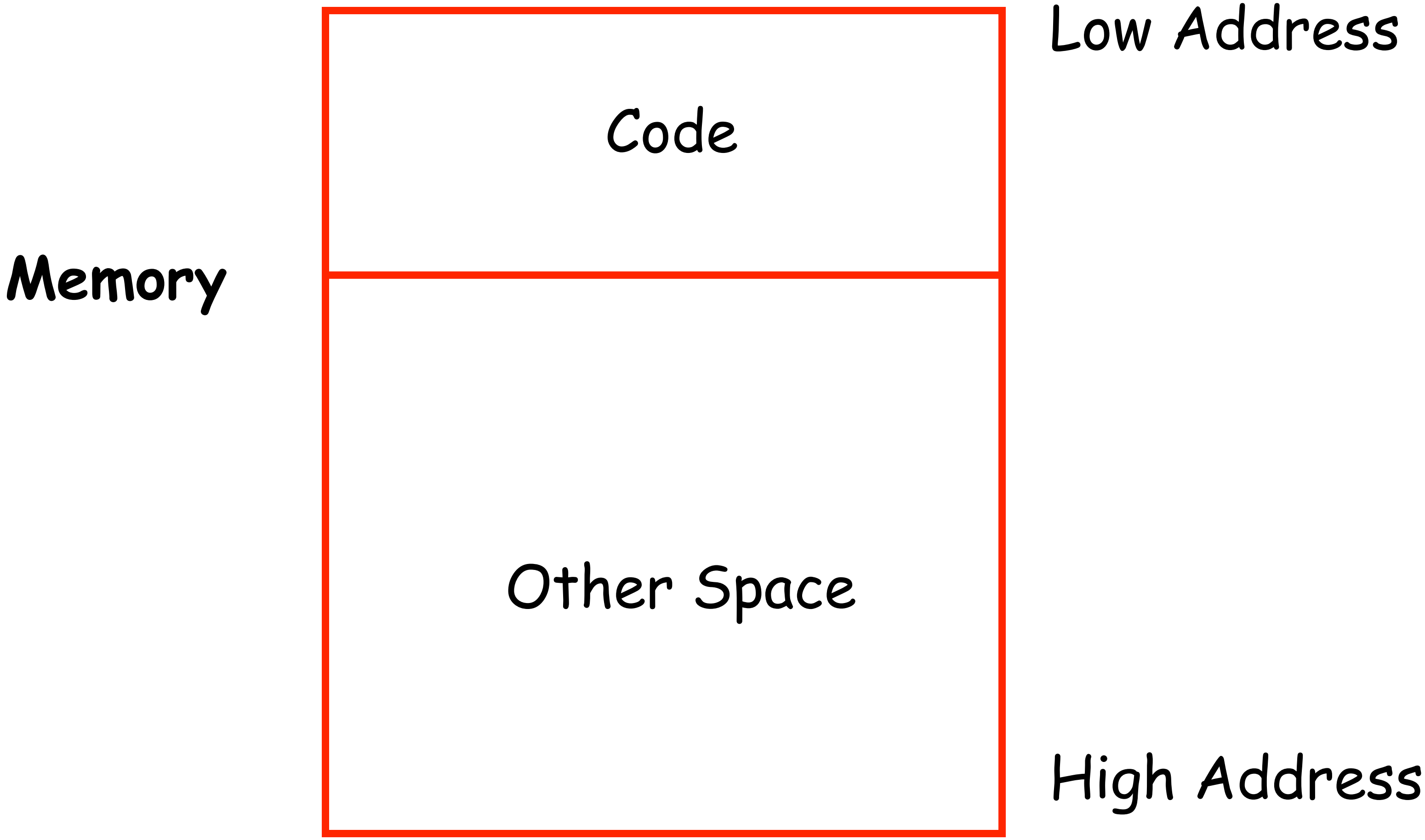
Outline

- Management of run-time resources
- Correspondence between static (compile-time) and dynamic (run-time) structures
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system
- Run-Time Enviroment: Where the program is being executed
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



Notes

- By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

- These pictures are simplifications
  - E.g., not all memory need be contiguous

What is Other Space?

- Holds all data for the program
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed

- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order.

2. When a procedure is called, control eventually returns to the point immediately after the call.

Do these assumptions always hold?

Activations

- An invocation of procedure P is an *activation* of P

- The *lifetime* of an activation of P is
  - All the steps (instructions sequence) to execute P
  - Including all the steps in procedures that P calls

Lifetimes of Variables

- The *lifetime* of a variable x is the portion of execution in which x is defined

- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Activation Trees

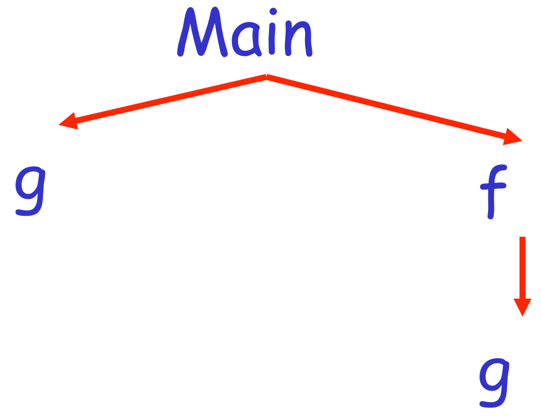
- Assumption (2) requires that when P calls Q, then Q returns before P does

- Lifetimes of procedure activations are properly nested

- Activation lifetimes can be depicted as a tree

Example

```
class Main {
   int g() { return 1; }
   int f() {return g(); }
   void main() { g(); f(); }
}
```



Example 2

```
class Main {
    int g() { return 1; }
    int f(int x) {
        if (x == 0) { return g(); }
        else { return f(x - 1); }
    }
    void main() { f(3); }
}
```

What is the activation tree for this example?

Notes

- The activation tree depends on run-time behavior, in particular:

- The activation tree may be different for a different input

- Since activations are properly nested, a stack can track currently active procedures

Example

```
class Main {
   int g() { return 1; }
   int f() { return g(); }
   void main() { g(); f(); }
}
```

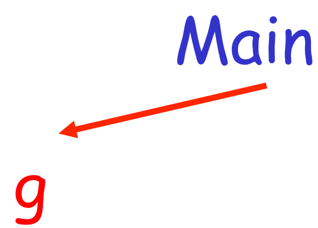
Main

Stack

Main

Example

```
class Main {
   int g() { return 1; }
   int f() { return g(); }
   void main() { g(); f(); }
}
```



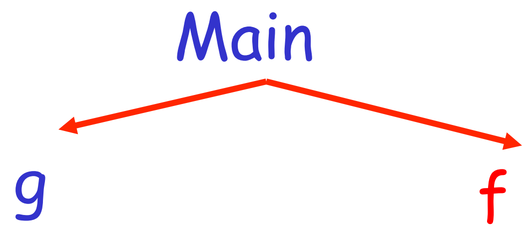
Stack

Main

g

Example

```
class Main {
    int g() { return 1; }
    int f() { return g(); }
    void main() { g(); f(); }
}
```



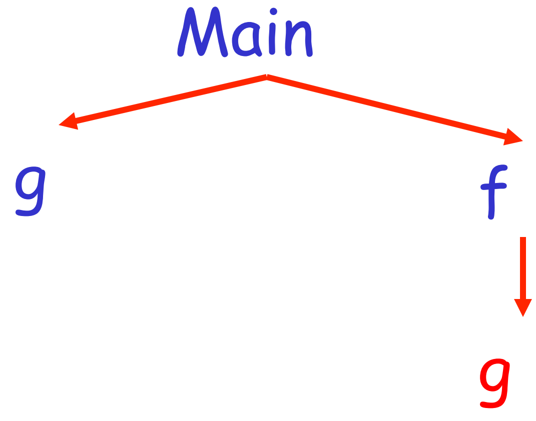
Stack

Main

f

Example

```
class Main {
    int g() { return 1; }
    int f() { return g(); }
    void main() { g(); f(); }
}
```



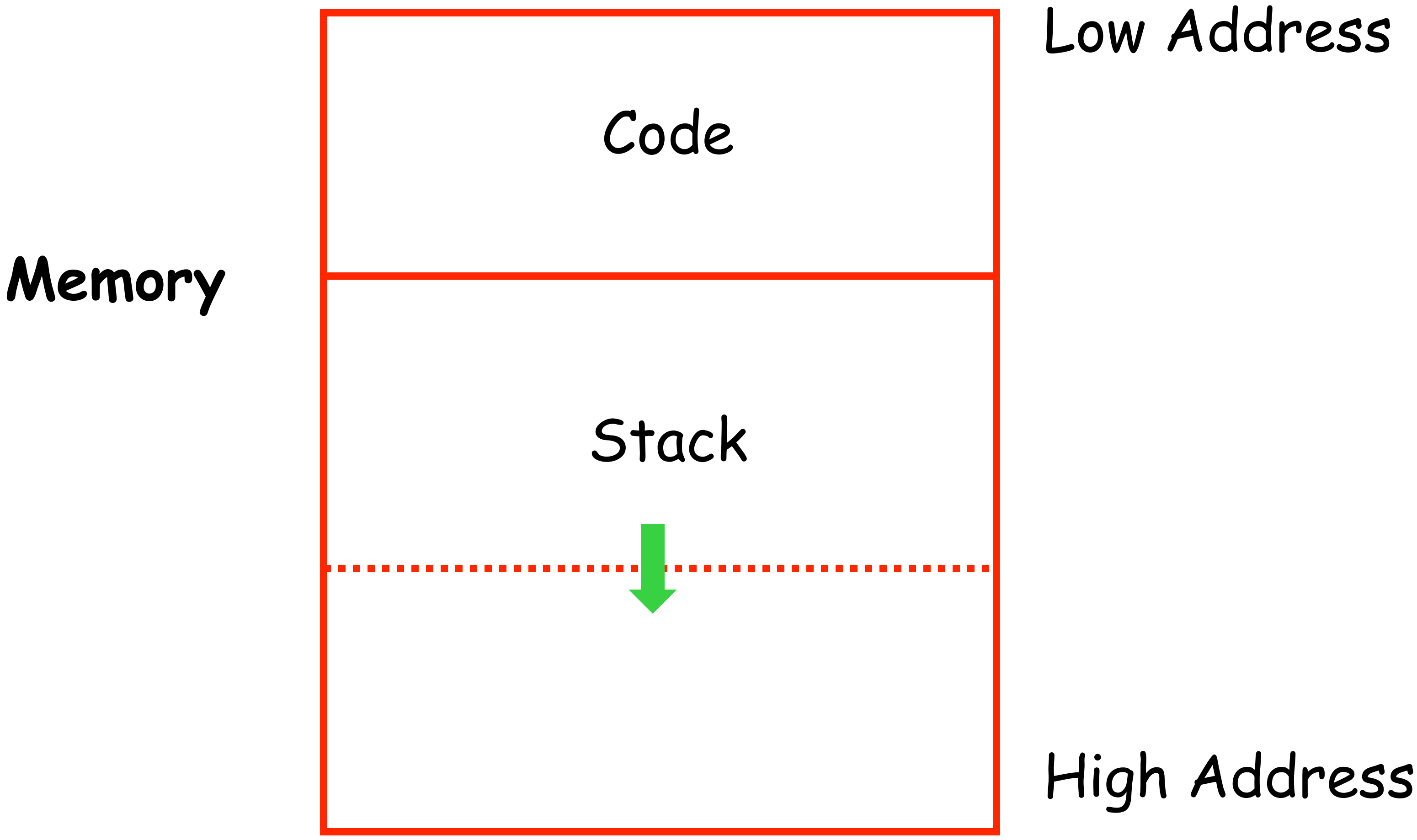
Stack

Main

f

g

Revised Memory Layout



Activation Records (Stack Allocation)

- The information needed to manage one procedure activation is called an *activation record* *(AR)* or *frame*

- Each live activation has its own AR pushed in the stack which is popped when it terminates

- If procedure F calls *G*, then *G*' s activation record contains a mix of info about F and *G*.

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes. G's AR contains information needed to resume execution of F.

- G's AR may also contain:
  - G's return value (to resume F)
  - Actual parameters to G (supplied by F)
  - Space for G's local variables

The Contents of a Typical AR for *G*

1. Space for *G*'s return value
2. Actual parameters
3. Pointer to the previous activation record: The optional Control Link
4. Pointer to previous activation records
   - The (optional) *control link* points to AR of the immediate super-nested procedure, needed to access non-local data stored in other ARs due to scope nexting of variables
5. Machine status prior to calling *G*
   - Contents of registers & program counter
6. Local and Temporary variables

Discussion

- The advantage of placing the return value 1st in a AR is that the caller can find it at a fixed offset from the end of its own AR without knowing the layout of the AR for the callee.

- Similar considerations apply for the local parameters.

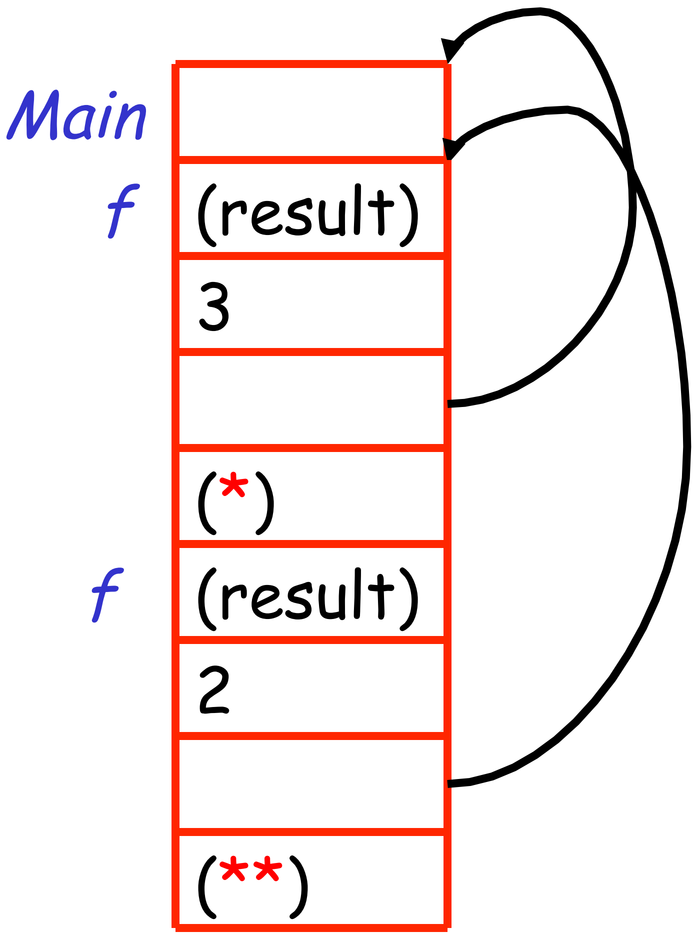
Example 2, Revisited

```
class Main {
    int g() { return 1; }
    int f(int x) {
        if (x == 0) { return g(); }
        else { return f(x - 1); (**) }
    }
    void main() { f(3); (*) }
}
```

AR for f:

return result
parameters
control link
return address and Registers
Local+Temporary

Stack After Two Calls to *f*



Notes

- Main has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

Local Variables

- Locals are stored in different AR for each different procedure execution:
  - Locals are bound to different storage in each activation (think of recursive calls)
- Storage is lost (free) when the activation ends
- The position of an AR is decided at run-time and stored in the SP (Stack Pointer) Register: a pointer to the beginning of the AR .
- Addresses for locals are determined at run-time as an offset from the SP Register and computed starting from the offset stored in the Symbol Table

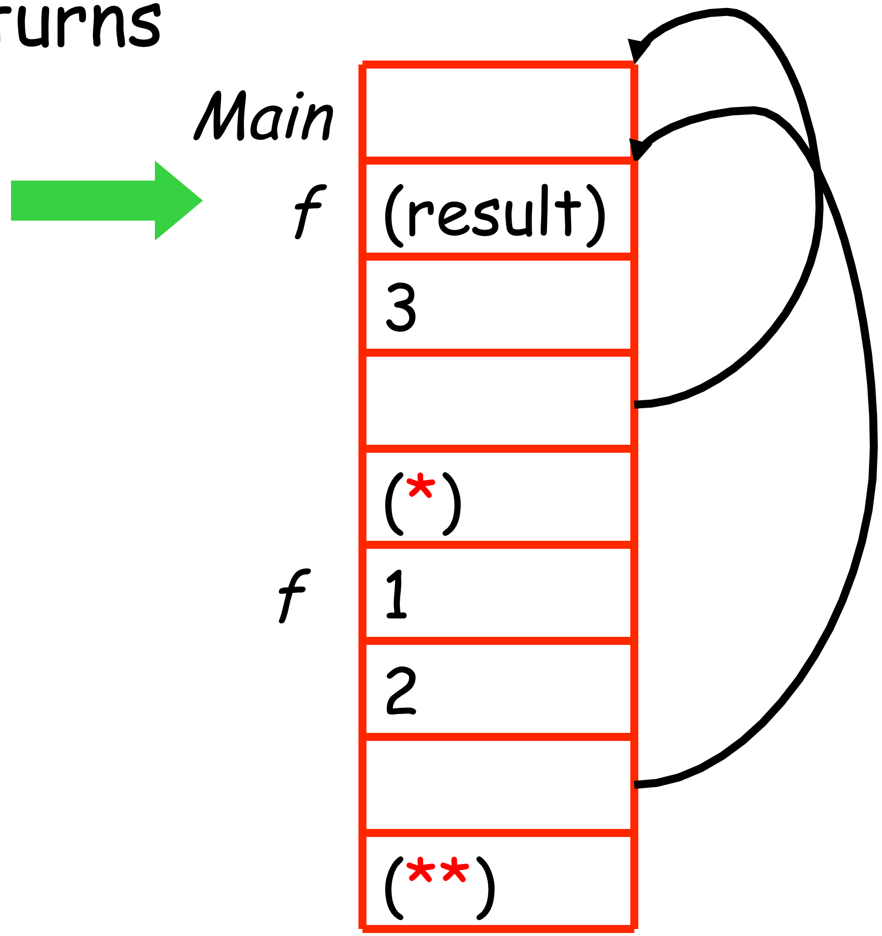
The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record

Thus, the AR layout and the code generator must be designed together!

Example

The picture shows the state after the call to 2nd invocation of f returns



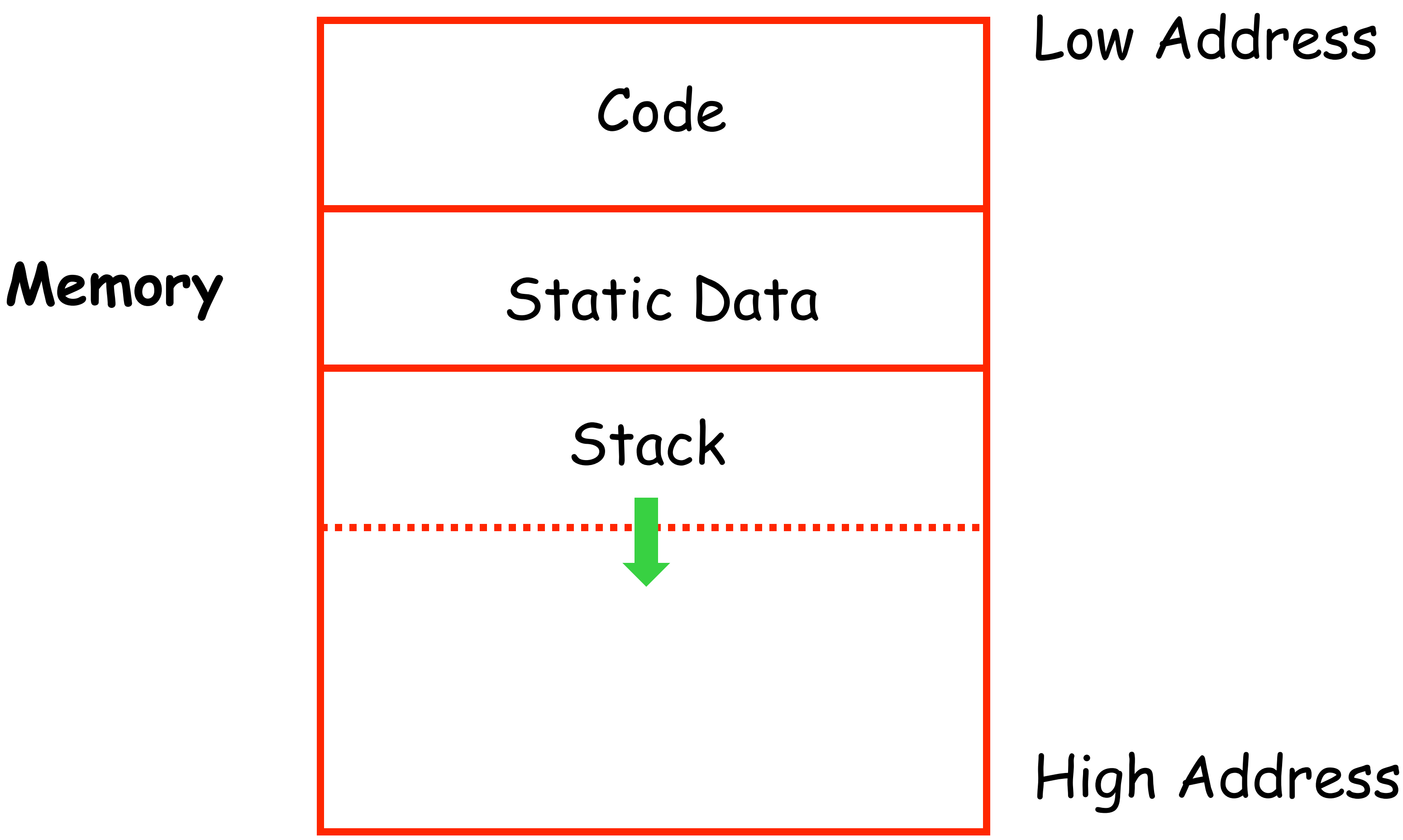
Discussion

- There is nothing magic about this organization
  - Can rearrange order of AR elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

Globals

- All references to a global variable point to the same object
  - Can't store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data



The Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

```
Bar
foo() { return new Bar }
```

  The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use the *heap* to store dynamic data
- Memory requests are satisfied by allocating portions from a large pool of memory called the heap or **free store**.

Memory Layout

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- The Heap contains all other data
  - Dynamic Data Structures
  - In C, the heap is managed by *malloc* and *free*

Memory Layout (Cont.)

- Both the heap and the stack grow
- Must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let the grow towards each other

Code for AR Allocation/Deallocation

- In the following we introduce a simplified Stack-Allocation code for AR

- We assume a simplified AR containing just the return address and Local+Temporary variables.

return address
Local+Temporary

Procedure Call

- The code for the first procedure initializes the stack by setting SP to the start of the of the Stack Area in Memory:

```
MOV #stackstart, SP  /* #stackstart given by OS */
```

- A Procedure Call Sequence must:
  - Increment the SP to point to the next AR
  - Transfer control to the called procedure

```
ADD #caller.recordsize, SP
MOV *PC+16, *SP    /* save return address */
GOTO calle.code_area
```

Note: The constant caller.recordsize is determined at complile time for each procedure using info in the Symbol Table.

Return Sequence

- The called procedure transfers control to the caller procedure using the return address stored at the beginning of its AR:

```
GOTO *0(SP)  /* return to caller */
```

- While 0(SP) denotes the address of first word in AR, *0(SP) is the return address saved there.

- In the caller procedure we need to decrement SP by restoring SP to point to the beginning of the caller AR:

```
SUB #caller.recordsize, SP
```

General Principles of Code Generation

- The target language depends on the particular architecture, e.g., RISC, CISC, Stack Machine,⋯

- 3 MAIN TASKS:
  1. Instruction Selection
  2. Register Allocation
  3. Instruction Ordering

General Principles of Code Generation

Instruction Selection

- Select the most appropriate instruction-set based on the set of instructions available in the target language (e.g., INC x must be preferred to x := x+1)

General Principles of Code Generation

Register Allocation

- Register are fast but limited in space: decide what variables to hold in Registers and what to hold in main memory;
- Good performing algorithms for Registers allocation makes a huge difference in performance;
- Avoid redundant LOAD and STORE operations;
- Minimize register usage for intermediate results.

General Principles of Code Generation

Instruction Ordering

- Involves deciding in what order to schedule the execution of instructions;
- Important in modern multi-processors machine that can execute several operations in a single clock cycle;
- The compiler is responsible for deciding what part of the generated code can be executed in parallel.