

# Cool Type Checking

## Cool Run-Time Organization



### Run-Time Organization

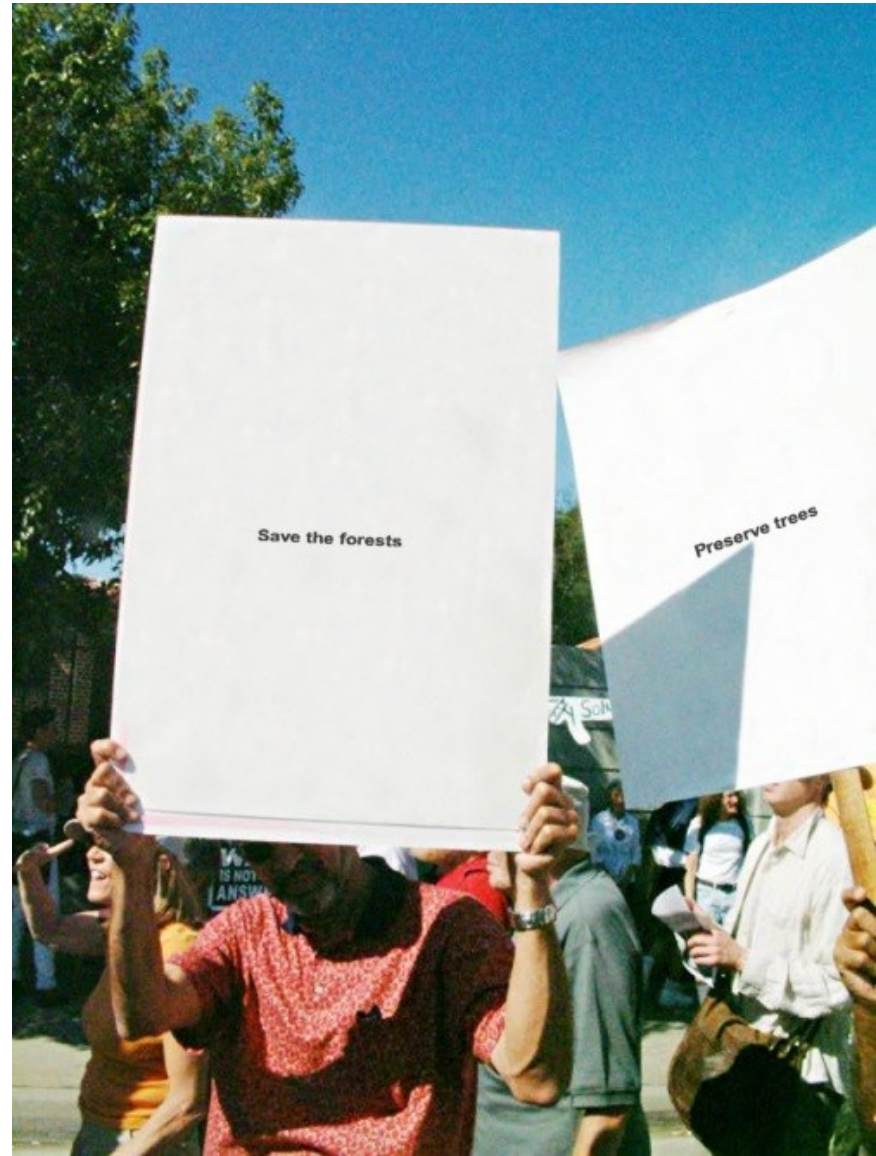
Gentlemen, tonight we are going after the big prize. The Keeblers are paying us handsomely, but some of us might not make it back from Pepperidge farm tonight...

# One-Slide Summary

- We will use **SELF\_TYPE<sub>C</sub>** for “C or any subtype of C”. It shows off the subtlety of our type system and allows us to check methods that return self objects.
- The **lifetime** of an activation of (i.e., a call to) procedure **P** is all the steps to execute **P** plus all the steps in procedures that **P** calls.
- Lifetime is a run-time (dynamic) notion; we can model it with trees or **stacks**.

# Lecture Outline

- SELF\_TYPE
- Object Lifetime
- Activation Records
- Stack Frames



# SELF\_TYPE Dynamic Dispatch

- If the return type of the method is **SELF\_TYPE** then the type of the dispatch is the type of the dispatch expression:

$$\mathbf{O, M, C \vdash e_0 : T_0} \quad \left. \vphantom{\mathbf{O, M, C \vdash e_0 : T_0}} \right\} \mathbf{A}$$

$$\begin{array}{c} \dots \\ \mathbf{O, M, C \vdash e_n : T_n} \end{array} \quad \left. \vphantom{\mathbf{O, M, C \vdash e_n : T_n}} \right\} \mathbf{B}$$

$$\mathbf{M(T_0, f) = (T_1', \dots, T_n', SELF\_TYPE)} \quad \left. \vphantom{\mathbf{M(T_0, f) = (T_1', \dots, T_n', SELF\_TYPE)}} \right\} \mathbf{C}$$

$$\frac{\mathbf{T_i \leq T_i' \qquad 1 \leq i \leq n} \quad \left. \vphantom{\mathbf{T_i \leq T_i' \qquad 1 \leq i \leq n}} \right\} \mathbf{D}}{\mathbf{O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_0}}$$

# Where is SELF\_TYPE Illegal in COOL?

$m(x : T) : T' \{ \dots \}$

- Only  $T'$  can be SELF\_TYPE! *Not T.*

What could go **wrong** if  $T$  were SELF\_TYPE?

```
class A { comp(x : SELF_TYPE) : Bool {...}; };  
class B inherits A {  
    b() : int { ... };  
    comp(y : SELF_TYPE) : Bool { ... y.b() ...}; };  
...  
let x : A ← new B in ... x.comp(new A); ...  
...
```

Diagram illustrating the code structure with annotations:

- Star 1 points to the variable `x` in the `let` statement.
- Star 2 points to the variable `y` in the `comp` method of class `B`.
- Star 3 points to the expression `new A` in the `comp` method of class `B`.
- Star 4 points to the `y.b()` expression in the `comp` method of class `B`.

# Summary of SELF\_TYPE

- The extended  $\leq$  and `lub` operations do a lot of the work. Implement them to handle `SELF_TYPE`
- `SELF_TYPE` can be used only in a few places. Be sure it isn't used anywhere else.
- A use of `SELF_TYPE` always refers to any subtype in the current class
  - The exception is the type checking of dispatch, where `SELF_TYPE` *as the return type* of an invoked method might have nothing to do with the current enclosing class

# Why Cover SELF\_TYPE ?

- SELF\_TYPE is a research idea
  - It adds more expressiveness to the type system
- SELF\_TYPE is itself not so important
  - except for the project
- Rather, SELF\_TYPE is meant to illustrate that type checking can be quite subtle
- In practice, there should be a balance between the complexity of the type system and its expressiveness

# Type Systems

- The rules in these lecture were Cool-specific
  - Other languages have very different rules
  - We'll survey a few more type systems later
- General themes
  - Type rules are defined on the **structure of expressions**
  - Types of variables are **modeled by an environment**
- Type systems tradeoff **flexibility** and **safety**



# Course Goals

- At the end of this course, you will be acquainted with the fundamental concepts in the **design and implementation** of high-level programming **languages**. In particular, you will understand the **theory and practice** of **lexing, parsing, semantic** analysis, and **code** interpretation. You will also have gained practical experience programming in multiple **different** languages.

# Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Optimization (optional)
  - Code execution (or code generation)
- We'll do **code execution** first . . .

# Run-time environments

- Before discussing code execution, we need to understand **what we are trying to execute**
- There are a number of standard techniques that are widely used for structuring executable code
- Standard Way:
  - Code
  - Stack
  - Heap



# Run-Time Organization Outline

- Management of run-time resources
- Correspondence between **static** (compile-time) and **dynamic** (run-time) structures
  - “Compile-time” == “Interpret-time”
- Storage organization

# Run-time Resources

- Execution of a program is initially under the control of the operating system
- 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”)
- How does “space” work?



# Space

A photograph of an astronaut in a white spacesuit floating in space. The astronaut is holding a long, thin object, possibly a tool or a piece of equipment. The background is a deep blue and black space with some light clouds visible at the bottom.

Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist's, but that's just peanuts to space. -- Douglas Adams

Space is as infinite as we can imagine, and expanding this perspective is what adjusts humankind's focus on conquering our true enemies, the formidable foes: ignorance and limitation. -- Vanna Bonta

# (Digression) Virtual Memory

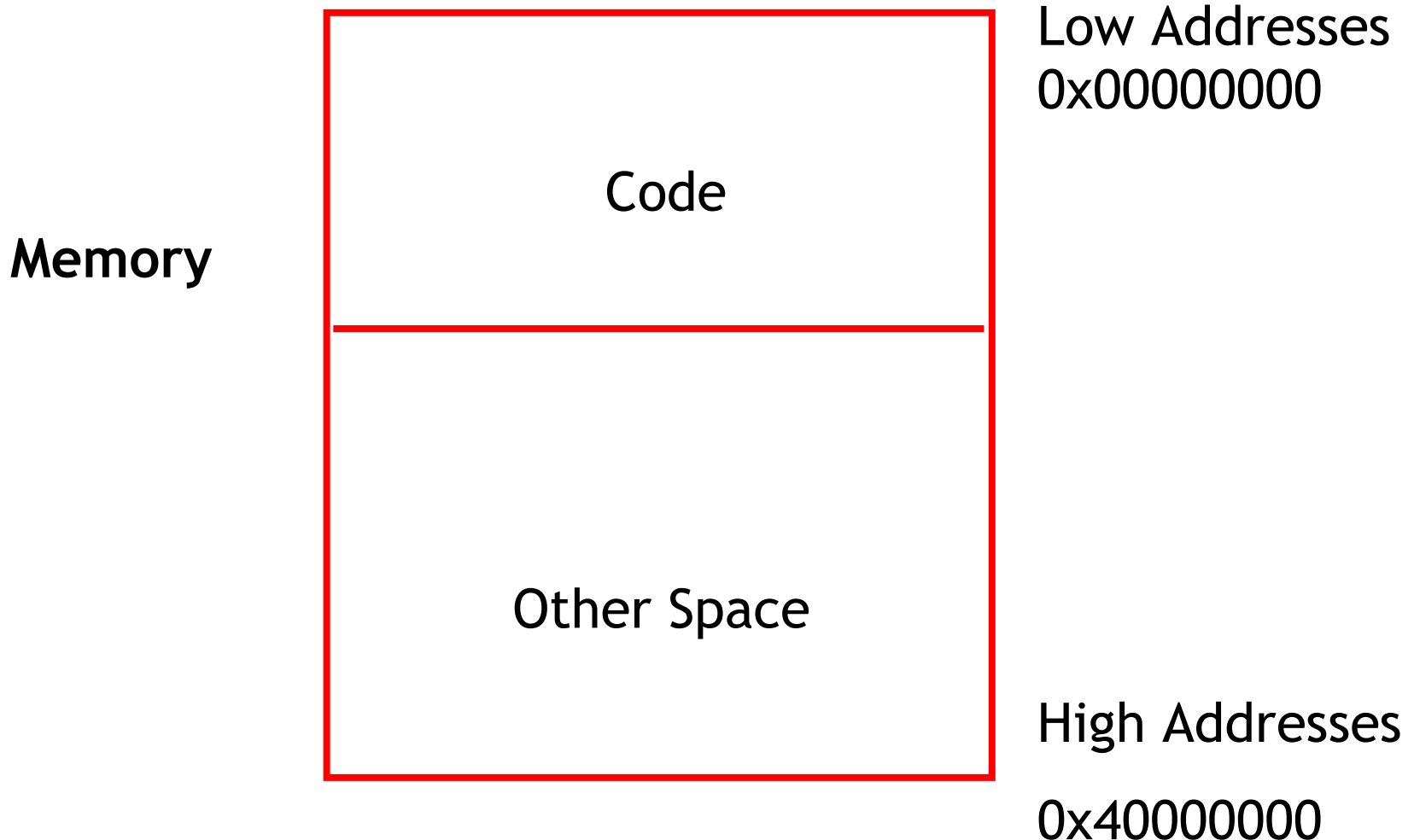
- An **address space** is a partial mapping from addresses to values. Like a big array: the value at memory address 0x12340000 might be 87. *Partial* means some addresses may be invalid.
- There is an address space associated with the **physical memory** in your computer. If you have 1GB of RAM, addresses 0 to 0x40000000 are valid.
- If I want to store some information on MachineX and you want to store some information on MachineX, we would have to collude to use *different* physical addresses (= different parts of the address space).

# (Digression) Virtual Memory 2

- **Virtual memory** is an abstraction in which **each process** gets its own *virtual address space*. The OS and hardware work together to provide this abstraction. All modern computers use it.
- Each virtual address space is then mapped separately into a different part of physical memory.  
(simplification)
- So **Process1** can store information at its virtual address **0x4444** and **Process2** can *also* store information at its virtual address **0x4444** and there will be *no overlap* in physical memory.
  - e.g., **P1 0x4444** virtual -> 0x1000 physical
  - and **P2 0x4444** virtual -> 0x8000 physical



# Program Memory Layout

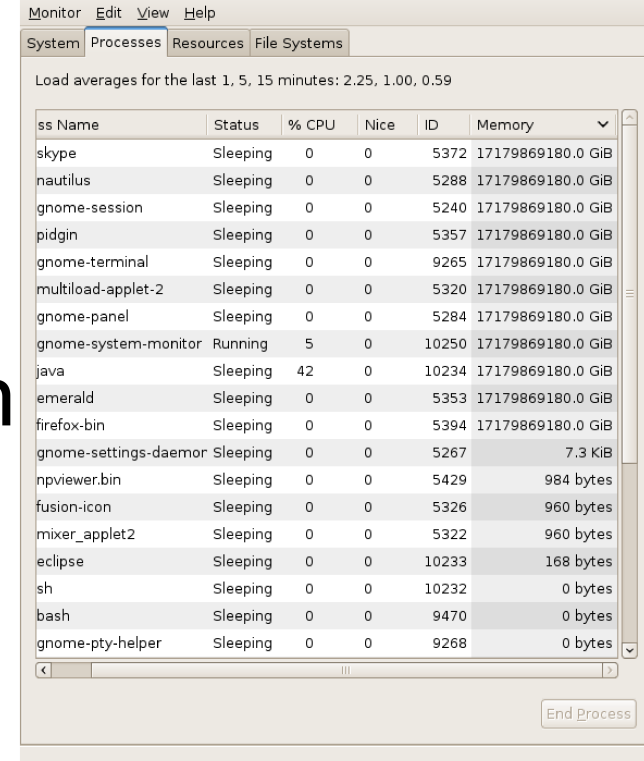


# Notes

- Our 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
- In some textbooks lower addresses are at bottom (doesn't matter)

# What is Other Space?

- Holds all data for the program
- Other Space = Data Space
- A **compiler** is responsible for:
  - Generating code (that is run later)
  - Orchestrating use of the data area
- An **interpreter** is responsible for:
  - Executing the code directly (now)
  - Orchestrating use of the (run-time) data



Monitor Edit View Help

System Processes Resources File Systems

Load averages for the last 1, 5, 15 minutes: 2.25, 1.00, 0.59

ss Name	Status	% CPU	Nice	ID	Memory
skype	Sleeping	0	0	5372	17179869180.0 GiB
nautilus	Sleeping	0	0	5288	17179869180.0 GiB
gnome-session	Sleeping	0	0	5240	17179869180.0 GiB
pidgin	Sleeping	0	0	5357	17179869180.0 GiB
gnome-terminal	Sleeping	0	0	9265	17179869180.0 GiB
multiloader-applet-2	Sleeping	0	0	5320	17179869180.0 GiB
gnome-panel	Sleeping	0	0	5284	17179869180.0 GiB
gnome-system-monitor	Running	5	0	10250	17179869180.0 GiB
java	Sleeping	42	0	10234	17179869180.0 GiB
emerald	Sleeping	0	0	5353	17179869180.0 GiB
firefox-bin	Sleeping	0	0	5394	17179869180.0 GiB
gnome-settings-daemon	Sleeping	0	0	5267	7.3 KiB
npviewer.bin	Sleeping	0	0	5429	984 bytes
fusion-icon	Sleeping	0	0	5326	960 bytes
mixer_applet2	Sleeping	0	0	5322	960 bytes
eclipse	Sleeping	0	0	10233	168 bytes
sh	Sleeping	0	0	10232	0 bytes
bash	Sleeping	0	0	9470	0 bytes
gnome-pty-helper	Sleeping	0	0	9268	0 bytes

End Process

# Code Execution Goals

- Two goals:
  - **Correctness**
  - **Speed**
- Most complications at this stage come from trying to be fast as well as correct



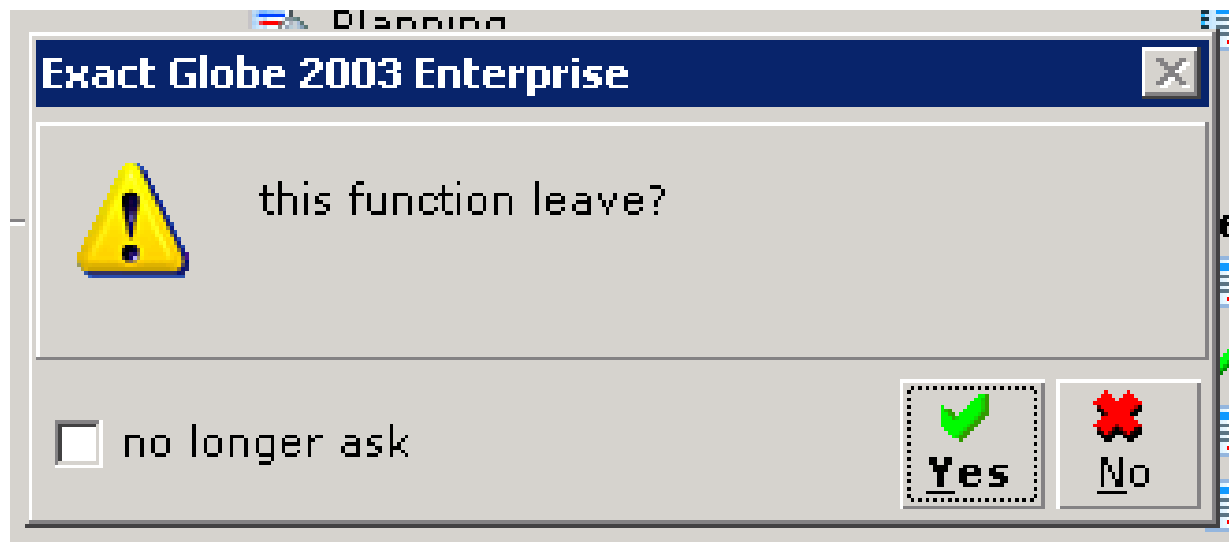
# Assumptions about Execution

- (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 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 during which  $x$  is defined
- Note that
  - Scope is a static concept
  - Lifetime is a **dynamic** (run-time) concept



# Activation Trees

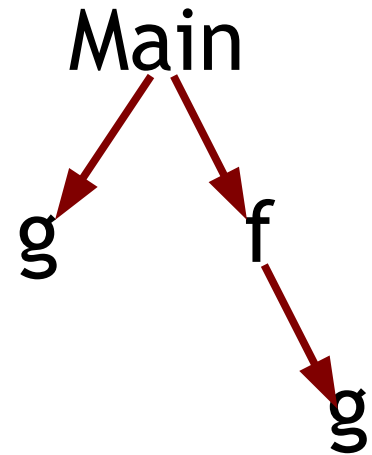
- Assumption (2) requires that when **P** calls **Q**, then **Q** returns before **P** does
- Lifetimes of procedure activations are **properly nested**
- Activation lifetimes can be depicted as a **tree**





# Example

```
Class Main {  
    g() : Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```



# Example 2

```
Class Main {  
    g() : Int { 1 };  
    f(x:Int): Int {  
        if x = 0 then g() else f(x - 1) fi  
    };  
    main(): Int {{ f(3); }};  
}
```

What is the activation tree for this example?

# Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a **stack** can track currently active procedures
  - This is the **call stack**

# Example

```
Class Main {  
  g() : Int { 1 };  
  f(): Int { g() };  
  main(): Int {{ g(); f(); }};  
}
```

Main

Stack

*Main*

# Example

```
Class Main {  
  g() : Int { 1 };  
  f(): Int { g() };  
  main(): Int {{ g(); f(); }};  
}
```

Main  
←  
*g*

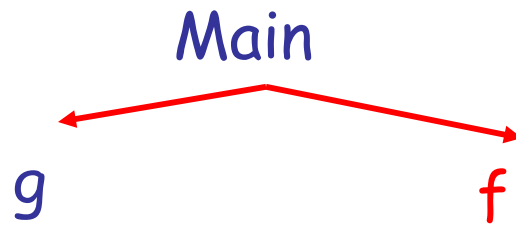
**Stack**

*Main*

*g*

# Example

```
Class Main {  
  g() : Int { 1 };  
  f(): Int { g() };  
  main(): Int {{ g(); f(); }};  
}
```



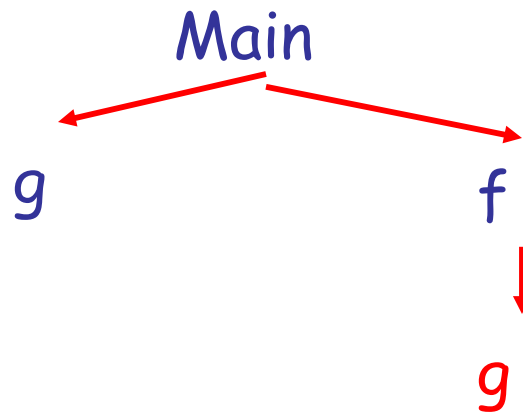
**Stack**

*Main*

*f*

# Example

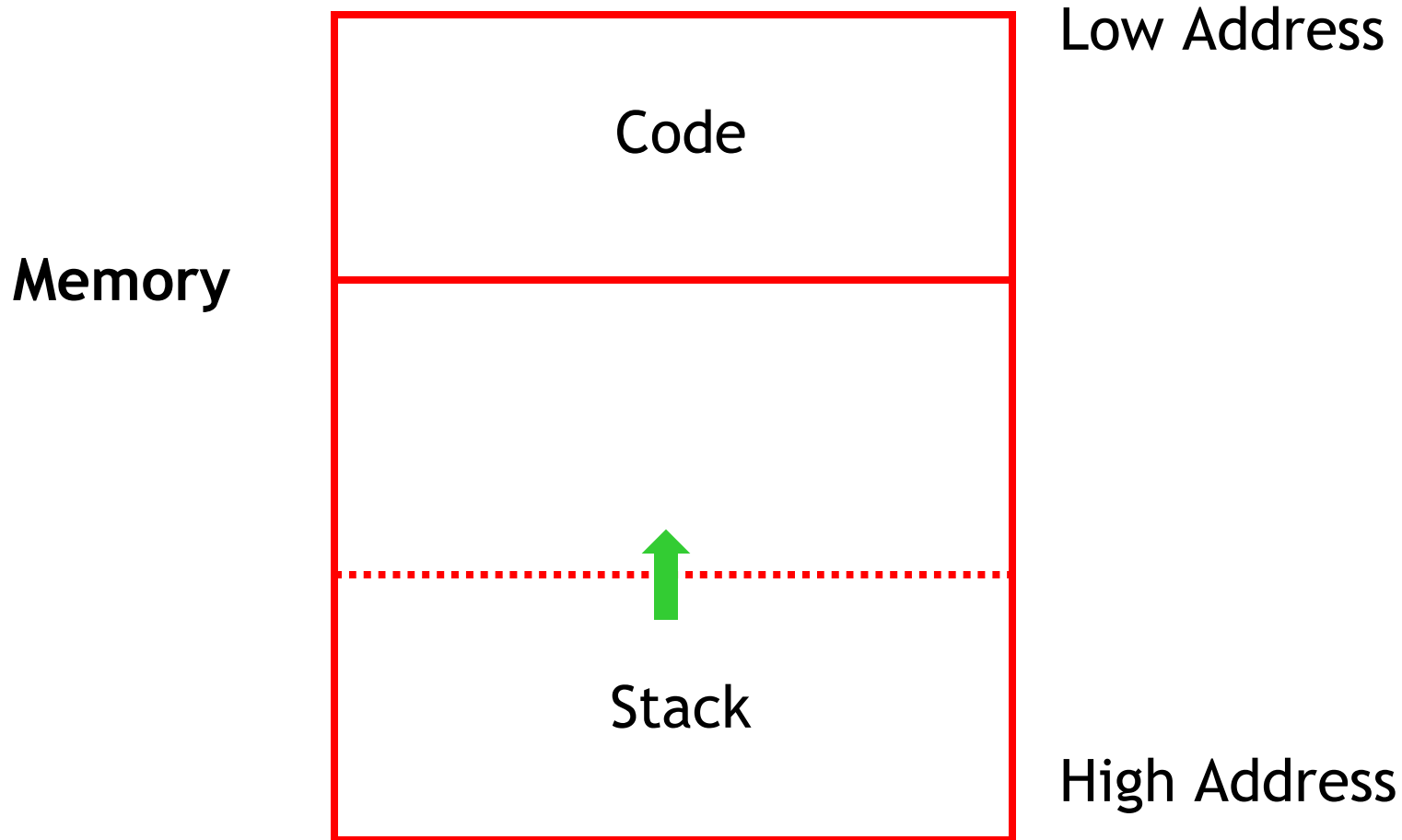
```
Class Main {  
  g() : Int { 1 };  
  f(): Int { g() };  
  main(): Int {{ g(); f(); }};  
}
```



**Stack**

*Main*  
*f*  
*g*

# Revised Memory Layout





Q: TV (110 / 842)

- Name the series and either of the characters involved in the first interracial kiss on US television. The kiss took place in the 1968 episode "Plato's Stepchildren".

# Real-World Languages

- This Asian language features a relatively small vocabulary of sounds, a focus on the relative status of the speaker and listener in the conversation, three written scripts, and S-O-V ordering. Ex: 日本に行きたい。

# Real-World Languages

- This Southern Athabaskan language is the most commonly-spoken Native American language north of Mexico. It has four basic vowels, two tones, inflected verbs, and was used as encryption to relay tactical secret messages in World War II.

# Activation Records

- On many machines the stack starts at high-addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an **activation record** (AR) or **frame**
- 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:
  - Actual parameters to **G** (supplied by **F**)
  - **G**'s return value (needed by **F**)
  - Space for **G**'s local variables

# The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The **control link** points to AR of F (caller of G)
  - (possibly also called the **frame pointer**)
- Machine status prior to calling G
  - Local variables
  - (Compiler: register & program counter contents)
- Other temporary values

# Example 2, Revisited

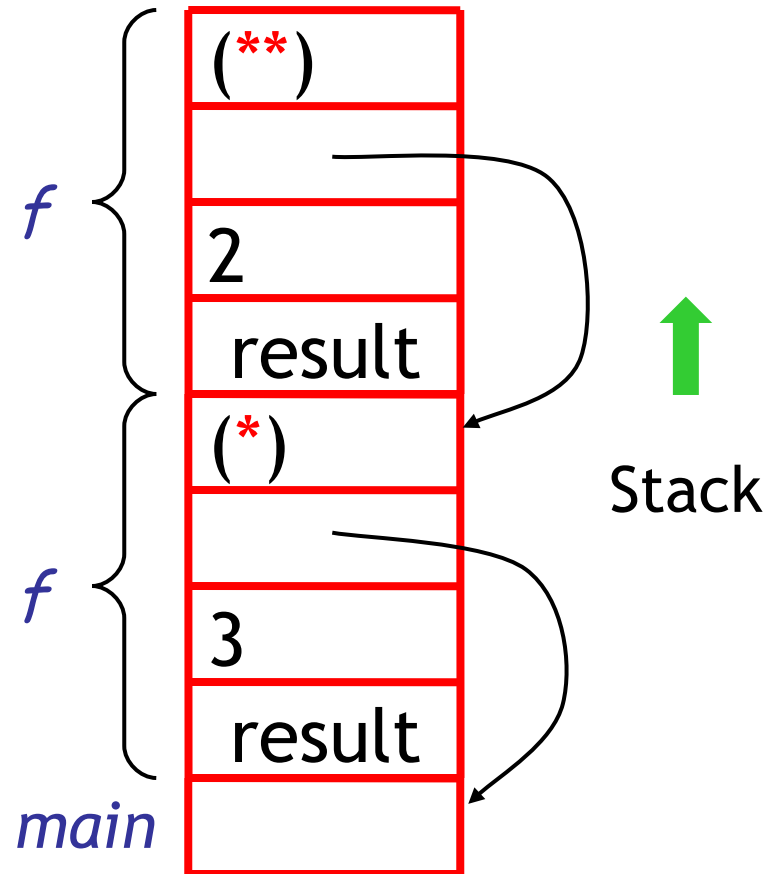
```
Class Main {  
  g() : Int { 1 };  
  f(x:Int):Int {  
    if x=0 then g() else f(x - 1) (**) fi  
  };  
  main(): Int {{f(3); (*) }};}
```

AR for f:

<i>return address</i>
<i>control link</i>
<i>argument</i>
<i>space for result</i>

# Stack After Two Calls to *f*

```
Class Main {  
  g() : Int { 1 };  
  f(x:Int):Int {  
    if x=0 then g()  
    else f(x - 1) (**) fi  
  };  
  main(): Int {{f(3); (*) }};  
}
```





# 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.

# The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that, when executed at run-time, correctly accesses locations in those activation records.

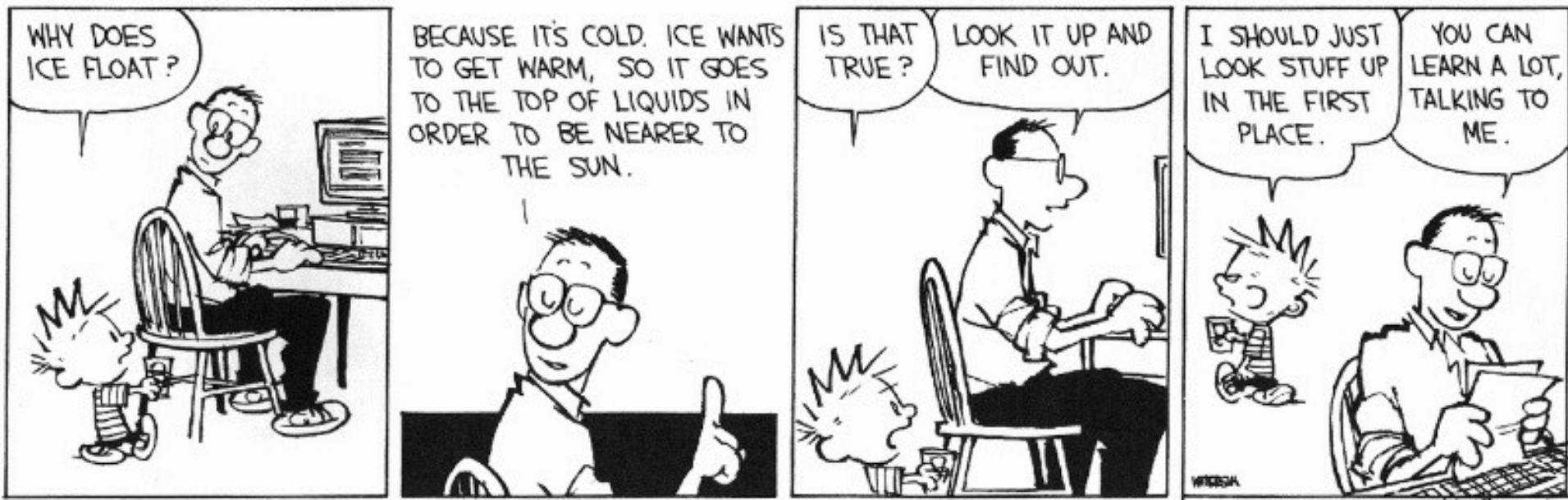
*Thus, the AR layout and the compiler must be designed together!*

# Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame
  - The caller must write the return address there
- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
    - Ask me about what embedded devices do.

# Discussion (Cont.)

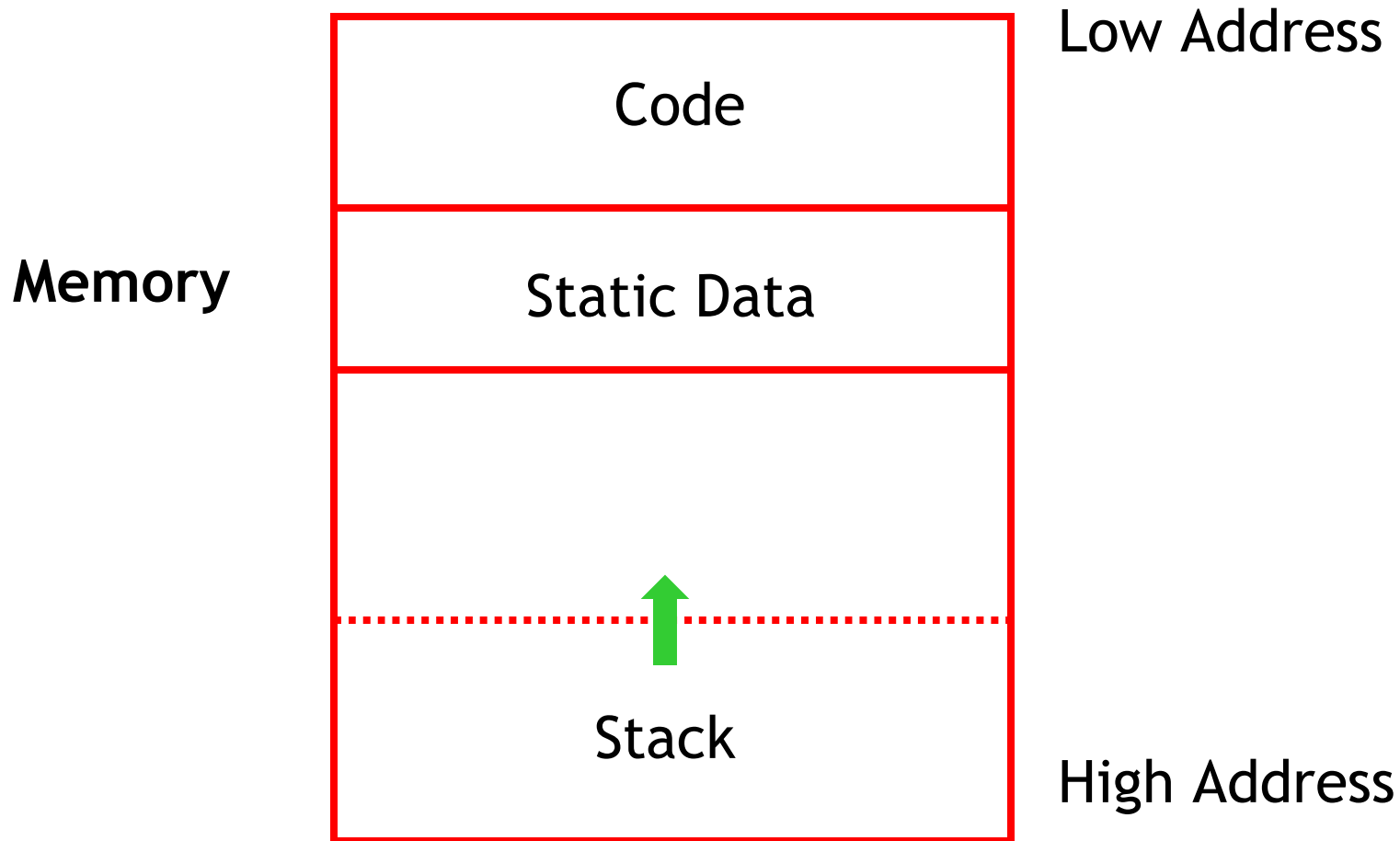
- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments
- Why?



# Globals

- All references to a global variable point to the same object
  - Can't store a global in an activation record
    - Is this true?
- 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



# Heap Storage

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

`method foo() { new Bar }`

The `Bar` value must survive deallocation of `foo`'s AR

- Languages with dynamically allocated data use a **heap** to store dynamic data

# Notes

- 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
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*



# Notes (Cont.)

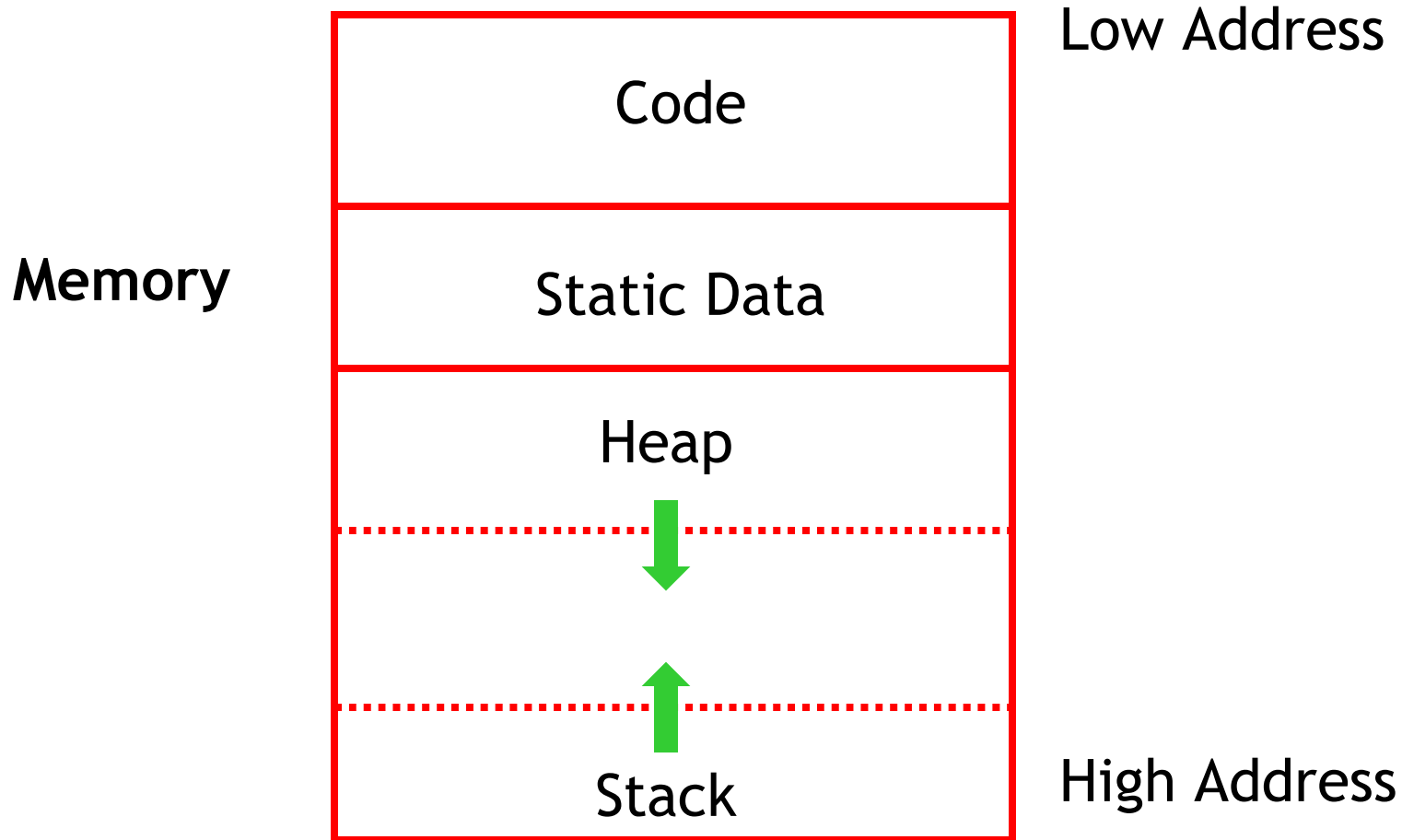
- Both the heap and the stack grow
- Compilers must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other



EXPERIENCE

It's what lets you recognize a mistake when you make it again.

# Memory Layout with Heap



# Your Own Heap

- PA6 must emit assembly code for things like:  
let **x** = new Counter(5) in  
let **y** = **x** in {  
    **x**.increment(1);  
    print( **y**.getCount() ); // what does this print?  
}
- You'll need to use and manage **explicit heap** (as described today and also next week). A heap maps addresses (integers) to values.

# Homework

- WA4 due
- PA4 due on Wednesday