# Naming, Binding, and Storage

CSCI 3136: Principles of Programming Languages

# Agenda

- Announcements
  - Assignment 6 is out and due July 5
  - Assignment 7 is out and due July 12 (Programming)
- Readings: Read Chapter 3
- Lecture Contents
  - Motivation
  - Naming and Binding
  - Reference Environments and Scope
  - Binding Times
  - Object and Binding Lifetimes
  - Storage Allocation

# Motivation

- Programs manipulate data through execution of code
- Both data and code needs to be identified when used in the program
- We use names (*symbols)* to identify the data or code
- How we link the symbols to the code or data affects what we can do!
- Question: Fundamentally, is there a difference between code and data?

# Naming and Binding

- A **Name** is a (mnemonic) string representing code or data:
  - x, sin, foo, prog1, null? are names
  - 42, 3.14, “test”, false, are literals, not names
  - +, !=, <<  . . . are operators and may be names if they are not built-in
- A **Binding** is an association between a name and code or data:
  - Name and value (for constants)
  - Name and memory location (for variables)
  - Name and function
  - Name and label (for gotos)
- Names and bindings are stored in a symbol table at compile time

# Symbol Table Implementation

- Use a hash table to map names (strings) to properties (tuples/structs/objects/etc)
- The insert() operation inserts (name, properties) into hash table.
- The lookup() operation gets name from hash table, which returns the properties of the symbol.

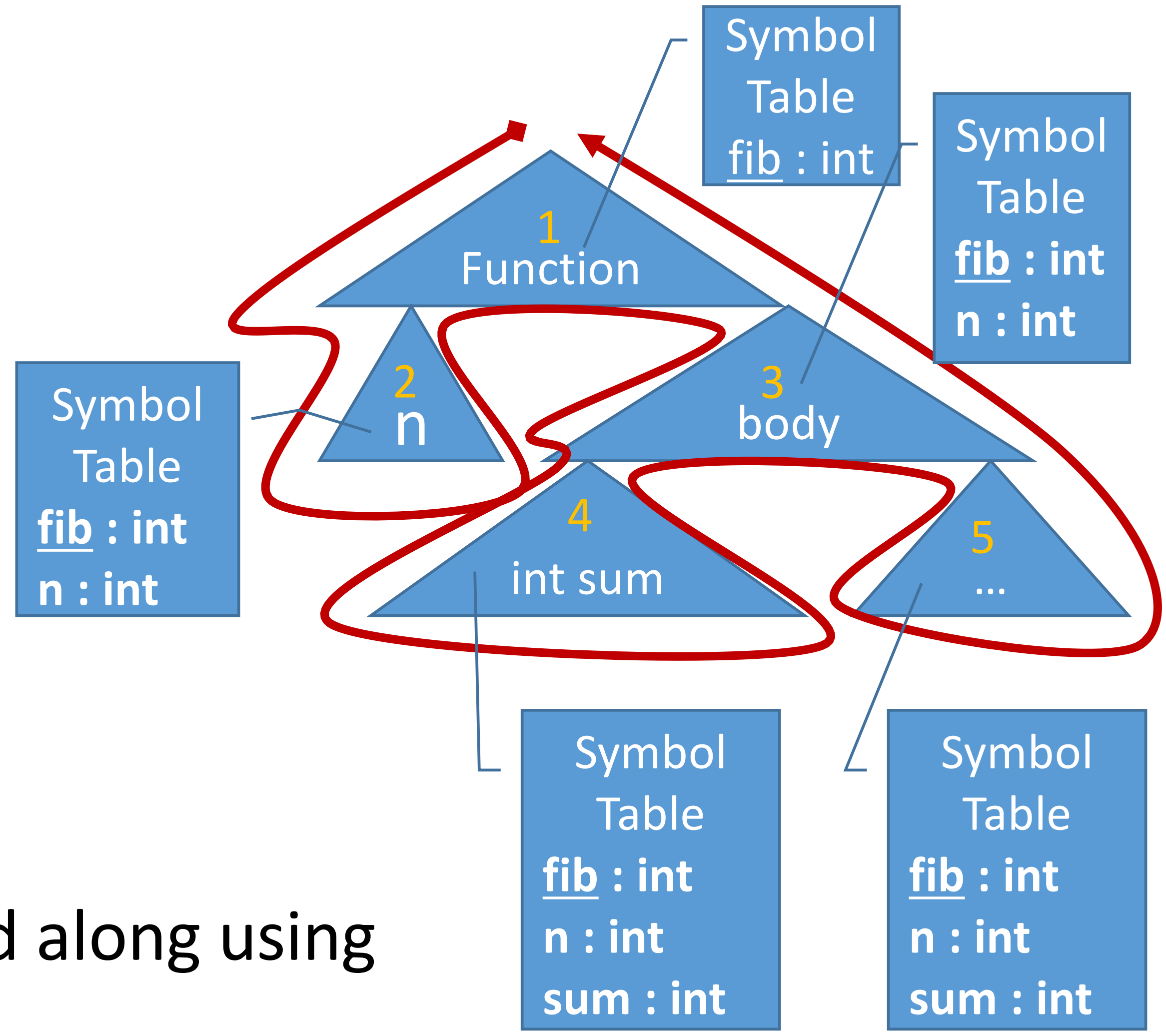
- How do we use the symbol table during semantic analysis?

# Example: a Function Definition

- Code

```
int fib(int n) {
   int sum
   …
}
```

- Order of evaluation:
  - Return type
  - Param List
  - Body
  - Variable Declaration
  - Rest of the body …
- Symbol table is passed along using inherited attributes



# Reference Environment and Scope

- A **Referencing Environment** is a complete set of active bindings at a point in a program
- The **Scope of a Binding** is
  - the region(s) of a program or
  - time interval(s) in the programs execution

  during which the binding is active
- A **Scope** is a maximal region of the program where no bindings are <u>destroyed</u>

  e.g., body of a procedure
- When do bindings occur?

```
int foo( int a, int b ) {
  // Inside the braces is
  // the scope of foo
}
```

# Binding Times

- Compile time
- Link time
- Load time
- Run time

# Compile-Time Binding

Bindings are encoded in machine code

- Literals are stored in the object file

```
final static String hello = "Hello, World!\n";
```

- Static data

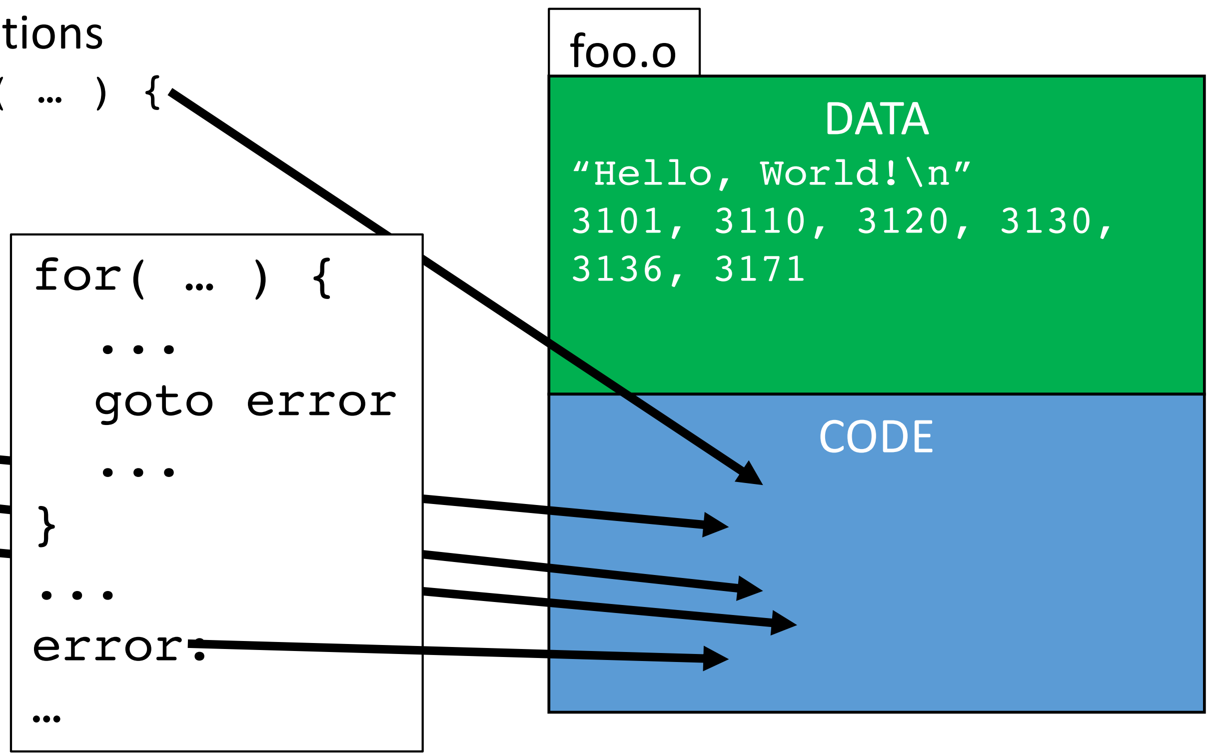
```
static int [] ThirdYearCore = {3101, 3110, 3120, 3130, 3136, 3171};
```

- Local (static/private) functions

```
private int helper( … ) {
    ...
}
```

- Gotos / labels
- Switch statements

```
switch( c ) {
case "YES": ...
case "NO": ...
case "MAYBE": ...
}
```



# Link-Time Binding

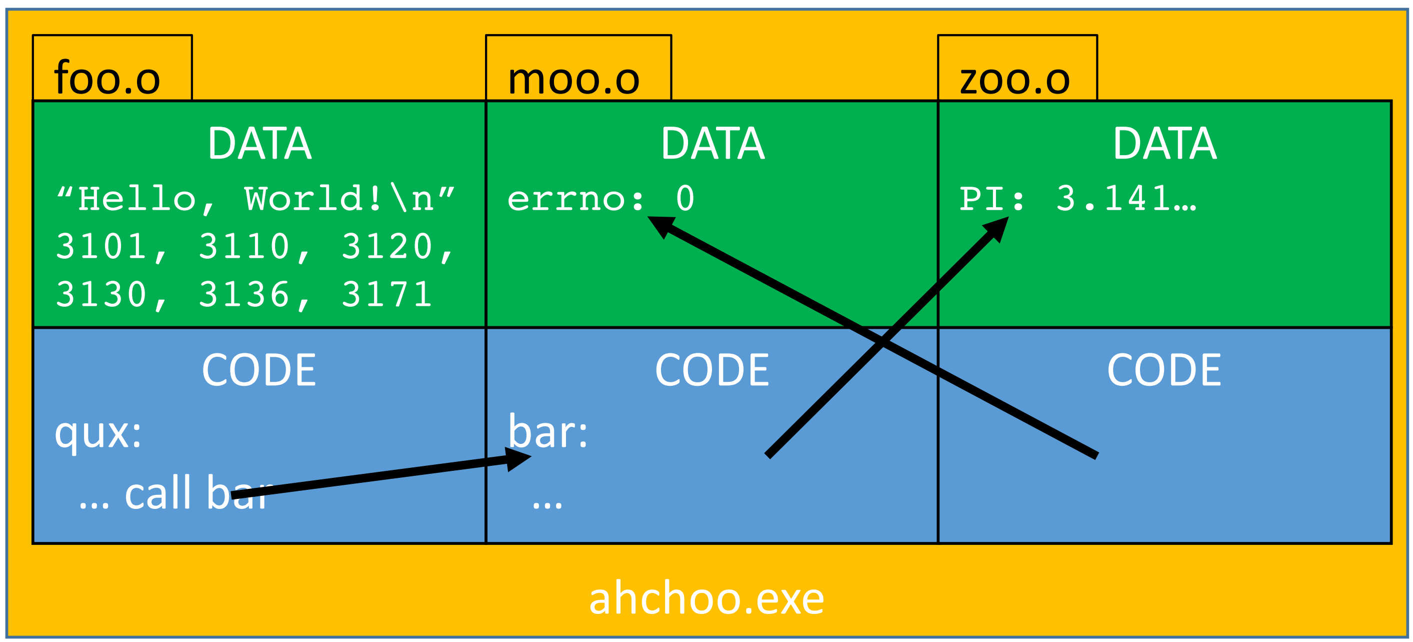
Bindings are initiated by compiler but finalized by linker

- Function calls between separately compiled modules

```
int qux( … ) {
   ...
   bar(...)
   ...
}
```

- Global variables

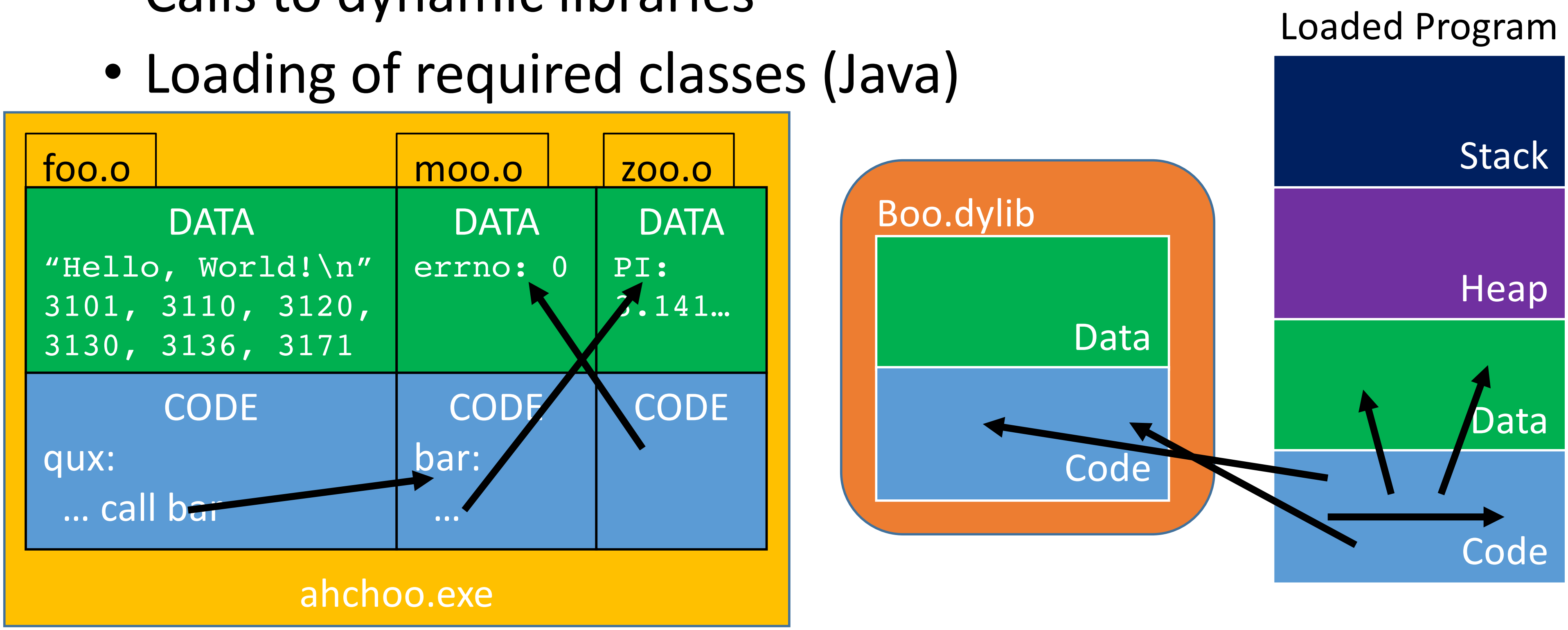
```
int errno
```



# Load-Time Binding

Bindings are finalized when program is loaded

- Static data locations are fixed (if not done by linker)
- Calls to dynamic libraries
- Loading of required classes (Java)



# Run-Time Binding

Bindings become active during execution

- Set variables

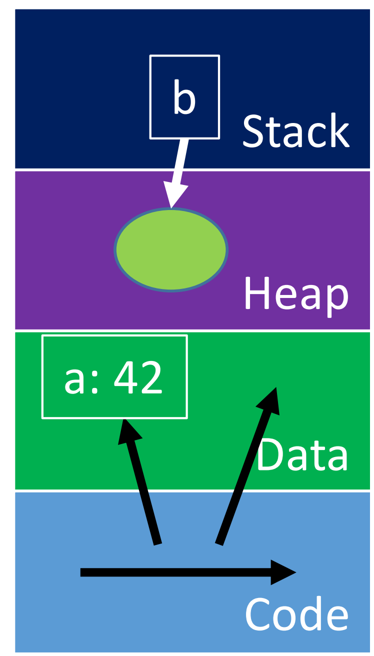
```
a = 42;
```

- Local and temporary variables created on the stack

```
int foo(…) {
  Object b;
  ...
}
```

- Dynamic memory allocation on the heap (assign references to variables)

```
b = new Object()
```



# Early vs Late Binding

- When a name is bound affects the program!
- Early binding : (compile/link/load time)
  - Faster code
  - Typical in compiled languages

  Example
  - Functions in C
- Late binding : (run time)
  - Greater flexibility
  - Typical in interpreted languages

  Example:
  - Lambda expressions in Scheme, Java, Python, etc

# Object and Binding Lifetimes

- **Object lifetime** : lifetime of code or data
- **Binding lifetime** lifetime of association between name and object

# Object Lifetimes

Lifetime of code or data

- Period between creation and destruction of object

Examples:

- Time between allocation (malloc) to deallocation (free) of a piece of memory

```
a = malloc( … );
...
free( a );
```

- Time between start/end of a function call (all local variables)

```
int foo( … ) {
  int a;
  int b;
  int c;
  ...
}
```

# Binding Lifetimes

Lifetime of association between name and object

- Period between the creation and destruction of name to object binding

Examples:

- Assignment to re-assignment of a pointer to memory

```
a = new Object();
…
a = null;
```

- Assignment to re-assignment of a function pointer

```
func = &func_a;
…
func = &func_b;
```

# Object Lifetimes vs Binding Lifetimes

- **Key Idea:** Lifetime of bindings typically match lifetime of objects except in the case of aliases
- Two common mistakes
  - Dangling reference : no object for a binding

    E.g., a pointer refers to an object that has already been deleted

```
m = malloc( … );
…
free( m );
// m no longer points to valid memory
```

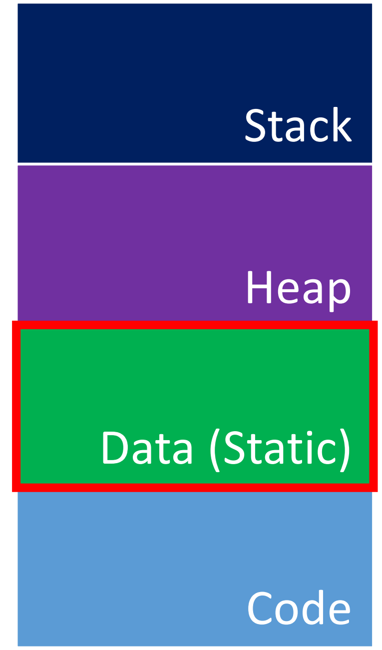
  - Memory leak : no binding for an object (preventing the object from being deallocated)

```
m = malloc( … );
m = NULL;
// memory allocated by malloc no longer accessible
```

# Storage Allocation

- Idea: An object's life-time depends on where it resides
- Options for Storage:
  - Static memory (Data)
  - Heap
  - Stack

# Static Objects



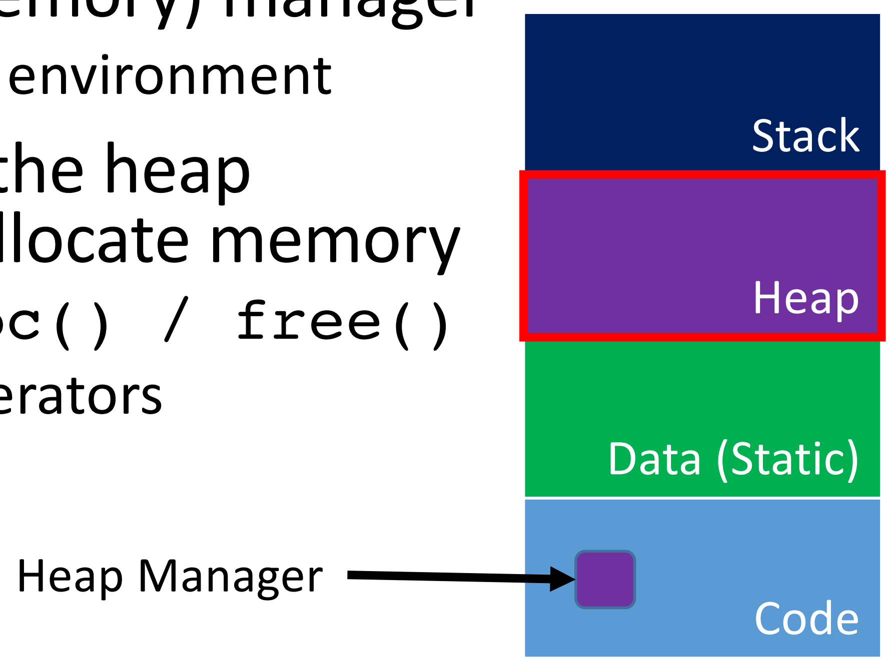

- Stored at a fixed absolute address
- Lifetime is the whole program execution
- Examples:
  - Global variables
  - Static variables local to subroutines that retain their value between invocations
  - Constant literals (strings)
  - Tables for run-time support: debugging, type checking, etc.
  - Space for subroutines, including local variables in languages that do not support recursion (e.g., early versions of FORTRAN)
- Allocated at compile/link/load time.

# Objects on the Heap

- Stored on heap
- Created and destroyed at arbitrary times
  - Explicitly by programmer
    E.g., malloc(), new, free()
  - Implicitly by garbage collector
- Examples:
  - Objects (Java)
  - Dynamically sized memory
  - Large objects

# Heap-Based Allocation

- The heap is a region of memory from which objects can be dynamically allocated
- The heap can grow as memory demands of a program grow.
- Each program has a heap (memory) manager
  - Part of the program's runtime environment
- Program's make requests to the heap manager to allocate and deallocate memory
  - In C: `malloc() / realloc() / free()`
  - In C++: `new / delete` operators
  - In Java: `new` operator



# Heap-Based De-allocation

Deallocation can take two forms:

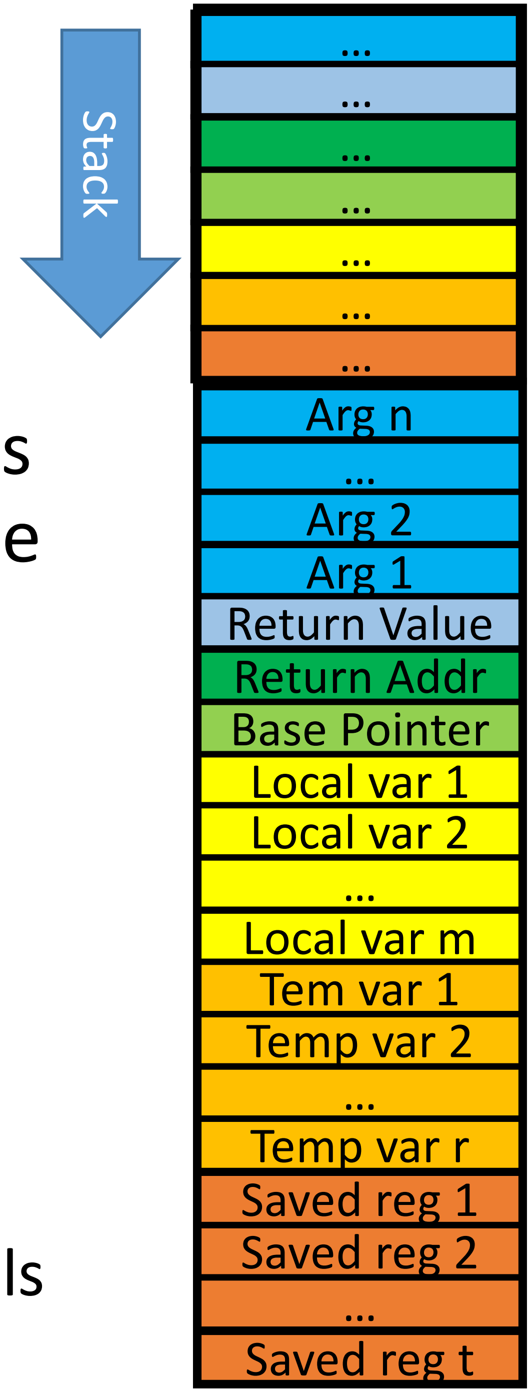
- **Explicit** deallocation by programmer
  - Used in Pascal, C, C++
  - Efficient
  - May lead to bugs that are difficult to find:
    - Dangling pointers/references from deallocating too soon
    - Memory leaks from not deallocating
- **Automatic** deallocation by garbage collector
  - Used in Java, functional, and logic programming languages
  - Can add significant runtime overhead
  - Safer

# Objects on the Stack

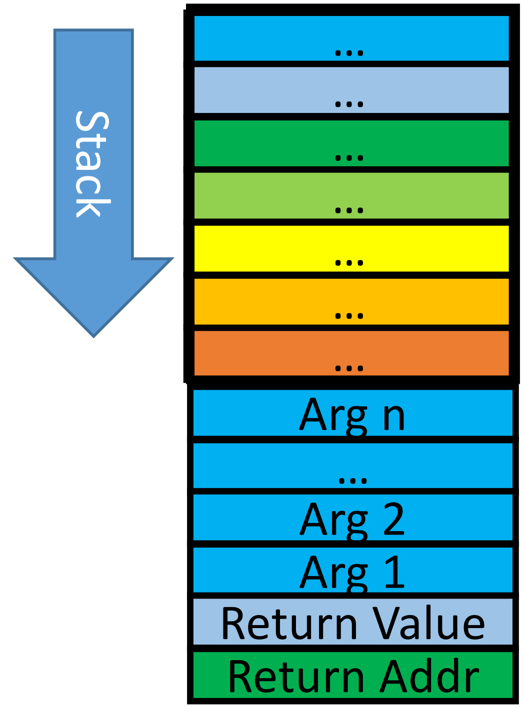
- Stored on stack in connection with a subroutine / method / function call
- Lifetime from invocation to return from subroutine
- Allocation is automatic with each call
- Examples:
  - All local variables
  - Arguments to a function
  - Return address
  - Temporary values

# Stack Based Allocation

- Idea: When a subroutine (function) is called, a *stack frame* is pushed on the stack
- Stack Frame contains:
  - Arguments passed to the subroutine
  - Space for the return value (optional)
  - Return address of caller
  - Local variables
  - Temporary (hidden) variables
  - Registers that must be saved across calls



# Before the Call




Caller

- Pushes arguments on the stack
- Pushes a dummy return value (optional)
- Executes call instruction

```
call foo
```

  - Pushes return address on stack
  - Jumps to subroutine (callee)

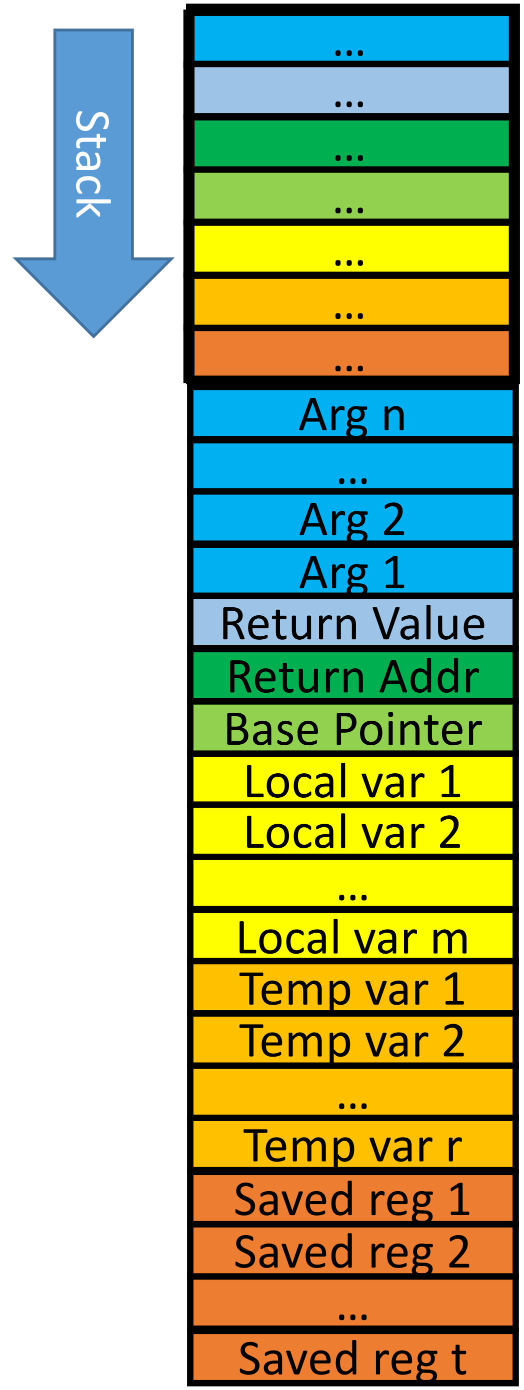
# During the Call

Callee

- Saves base pointer
- Allocates local variables
- Allocates temporary variables
- Saves registers
- Performs body of subroutine
- Restores registers
- Destroys local and temp variables
- Returns

  `ret`

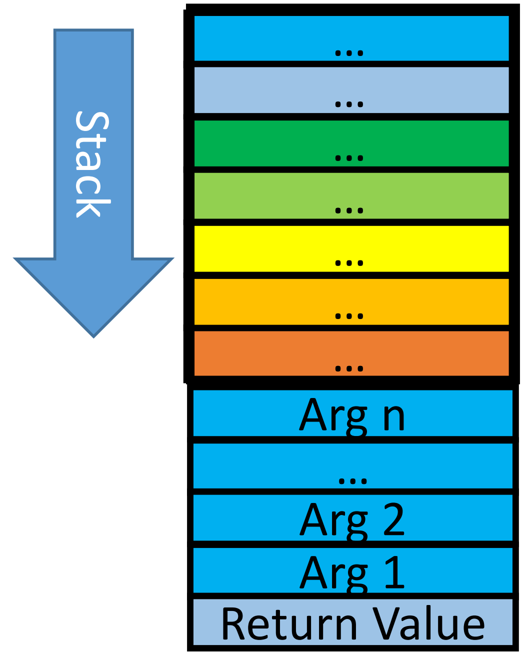
  - Pops return address off stack
  - Jumps to return address



# After the Call

Caller

- Pops arguments off the stack

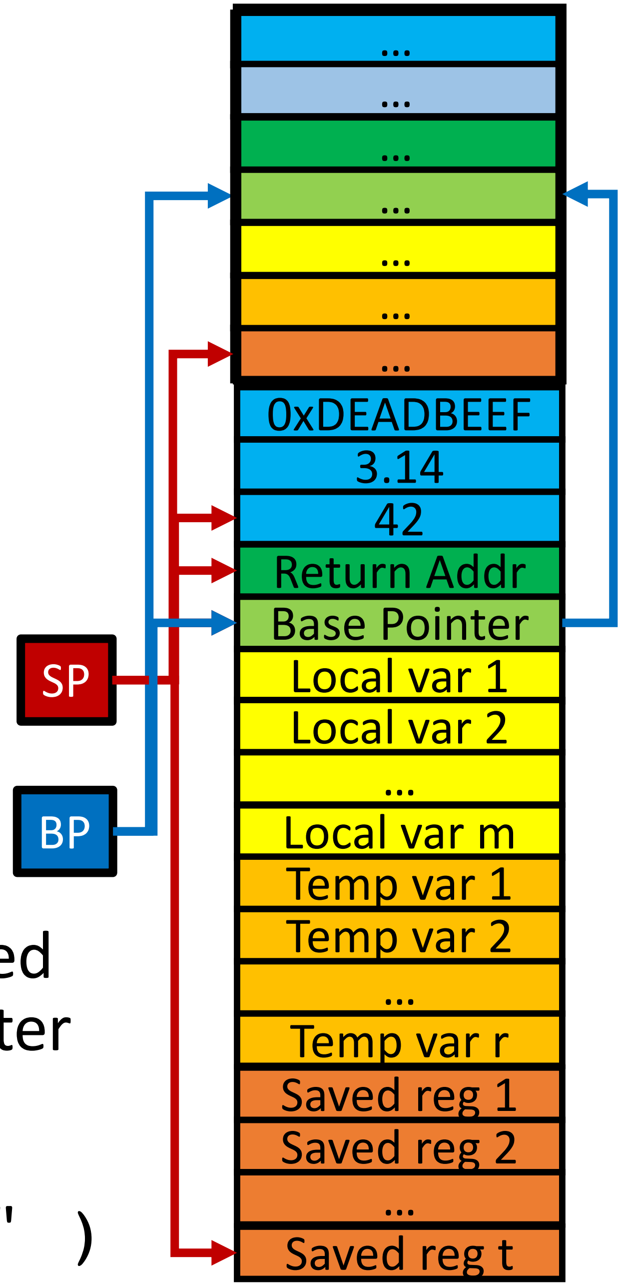


# Stack Management

- The stack is managed through two pointers (kept in registers)
  - **Stack pointer** points to the top of the stack
  - **Frame (base) pointer** points to the current stack frame

  Note: One of the registers typically saved on the stack is the previous frame pointer
- Example:

```
call foo( 42, 3.14, "Hello" )
```



# A Linked List of Stack Frames

- The stack frames are all linked, from the current function being executed all the way back to main()
- Stack frames contain
  - Return address of each function
  - Arguments passed
  - Local variables
- A debugger can easily access all this information and allow you to debug your program!