Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

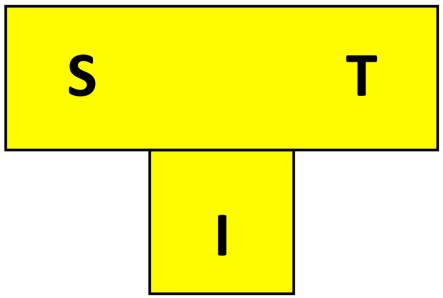
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 18

- Bootstrapping
- Names in programming languages
- Binding times
- Object allocation: static

Compilers, graphically

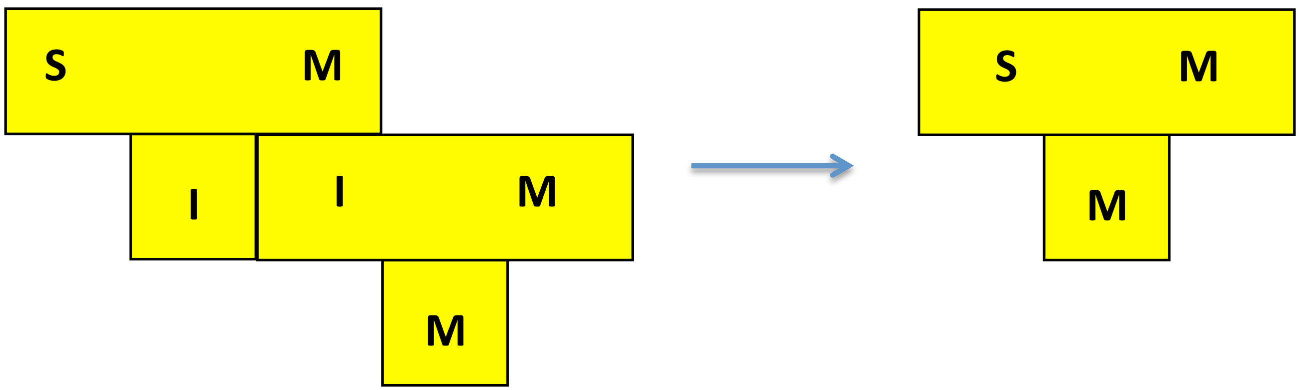
- Three languages involved in writing a compiler
  - Source Language (S)
  - Target Language (T)
  - Implementation Language (I)
- T-Diagram:



- If **I = T** we have a **Host Compiler**
- If **S**, **T**, and **I** are all different, we have a **Cross-Compiler**

Composing compilers

- Compiling a compiler we get a new one: the result is described by composing T-diagrams

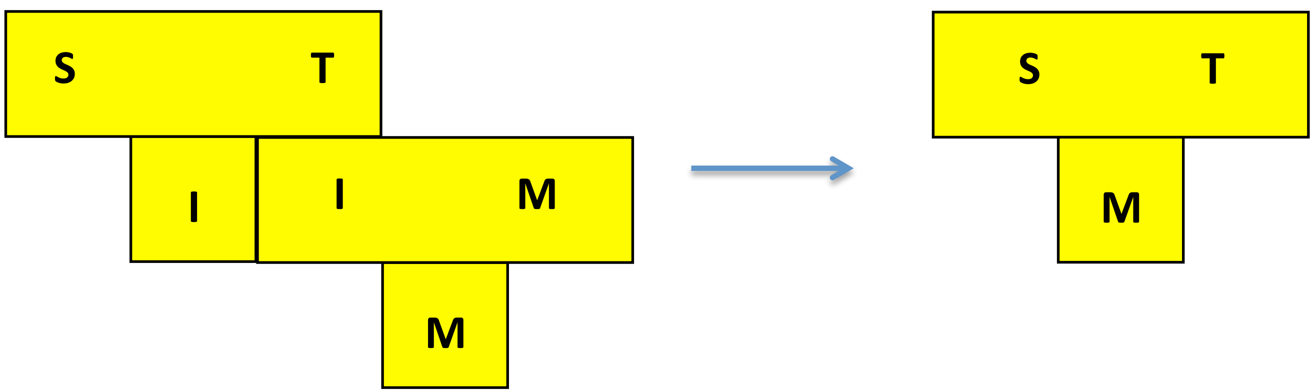


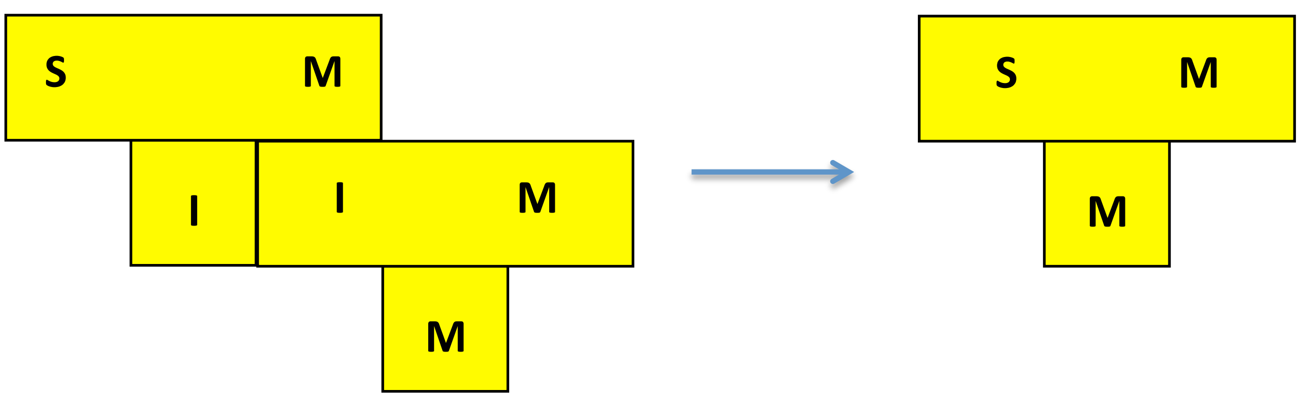
Example:
S **Pascal**
I **C**
M **68000**

- A compiler of **S** to **M** can be written in any language having a host compiler for **M**

Composing compilers

- Compiling a compiler we get a new one: the result is described by composing T-diagrams





Example:

S **Pascal**

I **C**

M **68000**

Bootstrapping

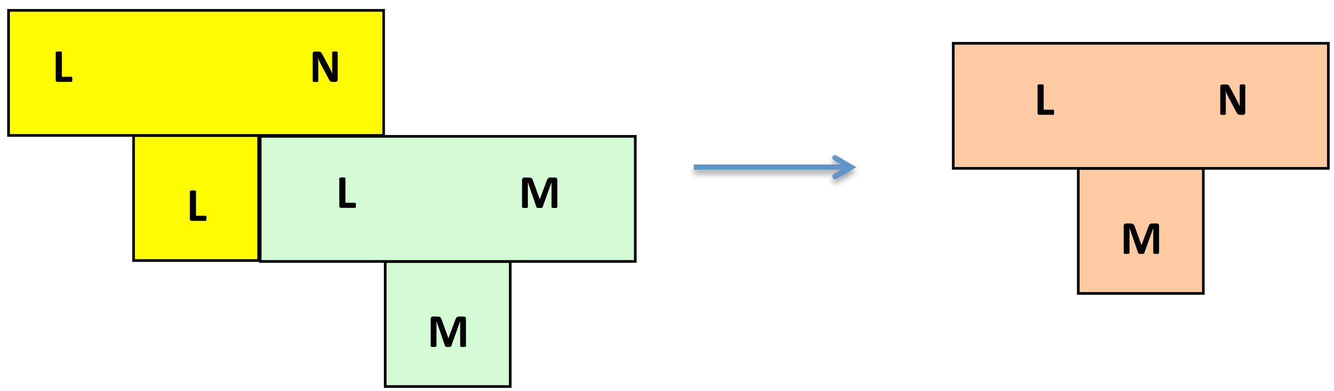
- **Bootstrapping**: techniques which use partial/inefficient compiler versions to generate complete/better ones
- Often compiling a translator programmed in its own language
- Why writing a compiler in its own language?
  - it is a non-trivial test of the language being compiled
  - compiler development can be done in the higher level language being compiled.
  - improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself
  - it is a comprehensive consistency check as it should be able to reproduce its own object code

Compilers: Portability Criteria

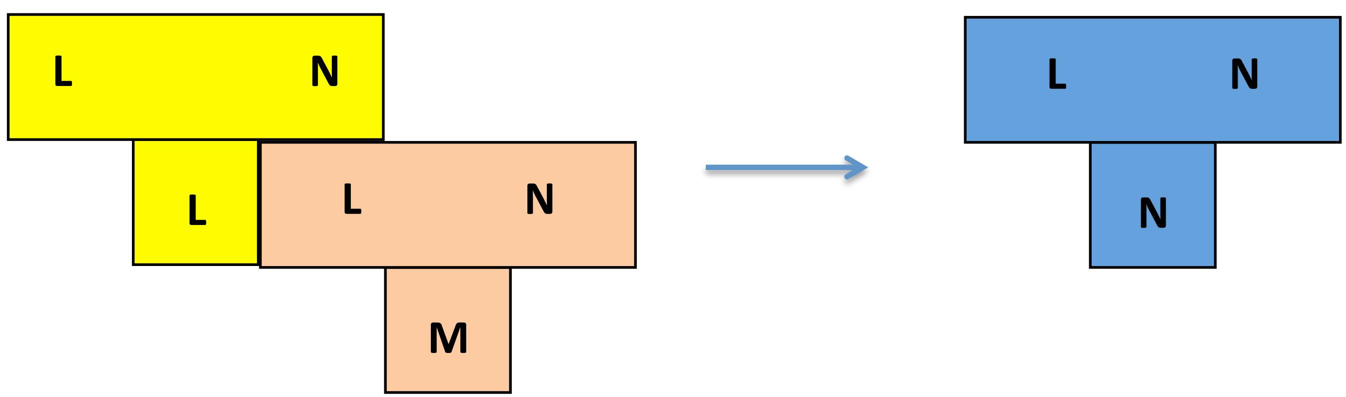
- Portability
  - Retargetability
  - Rehostability
- A **retargetable** compiler is one that can be modified easily to generate code for a new target language
- A **rehostable** compiler is one that can be moved easily to run on a new machine
- A portable compiler may not be as efficient as a compiler designed for a specific machine, because we cannot make any specific assumption about the target machine

Using Bootstrapping to port a compiler

- We have a host compiler/interpreter of **L** for **M**
- Write a compiler of **L** to **N** in language **L** itself



Example:
L **Pascal**
M **P-code**

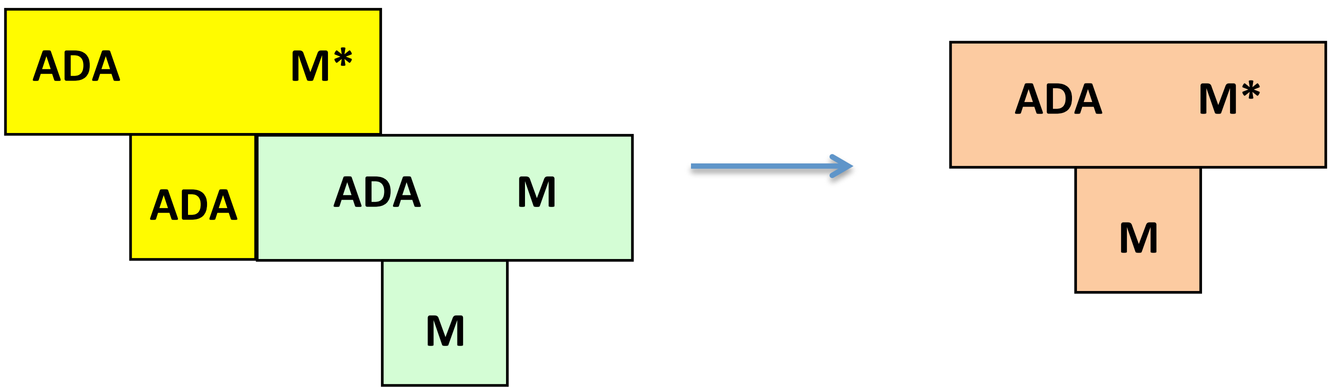


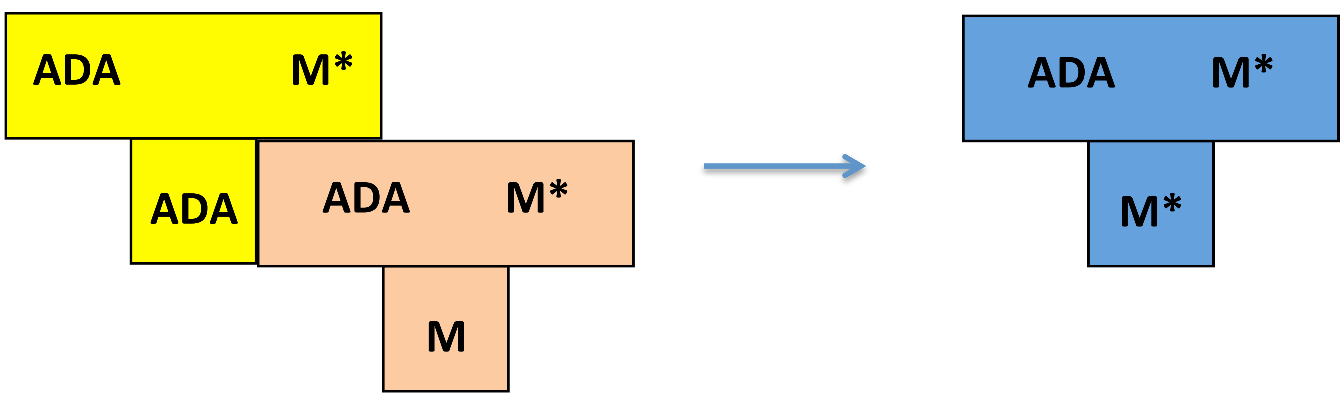
Bootstrapping to optimize a compiler

- The efficiency of programs and compilers:
  - Efficiency of programs:
    - memory usage
    - runtime
  - Efficiency of compilers:
    - Efficiency of the compiler itself
    - Efficiency of the emitted code
- Idea: Start from a simple compiler (generating inefficient code) and develop more sophisticated version of it. We can use bootstrapping to improve performance of the compiler.

Bootstrapping to optimize a compiler

- We have a host compiler of ADA to M
- Write an optimizing compiler of ADA to M in ADA





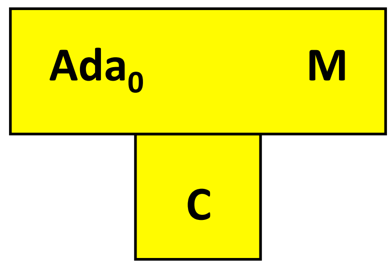
Full Bootstrapping

- A full bootstrap is necessary when building a new compiler from scratch.

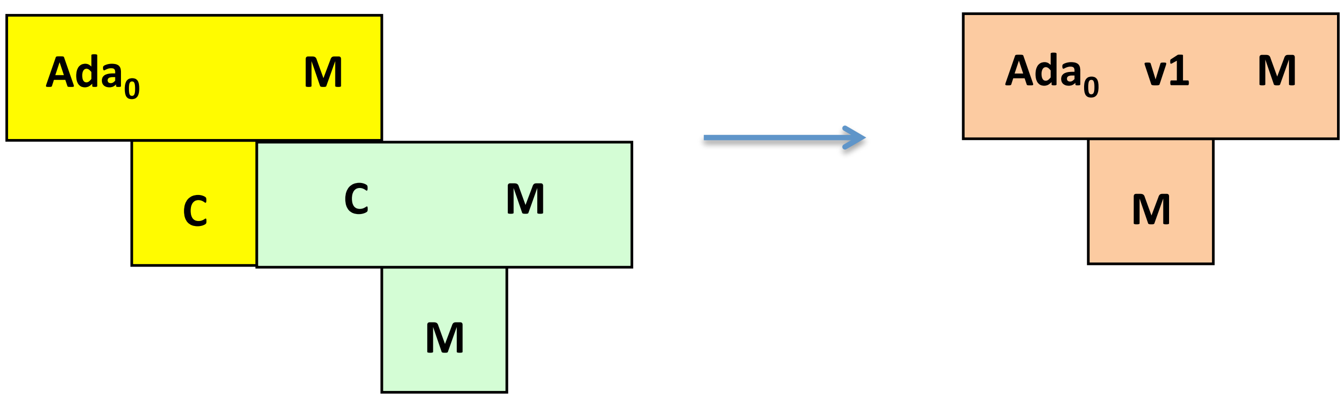
- **Example:**
- We want to implement an **Ada** compiler for machine **M**. We don’t have access to any **Ada** compiler
- Idea: **Ada** is very large, we will implement the compiler in a subset of **Ada** (call it $\mathbf{Ada_0}$) and bootstrap it from a subset of **Ada** compiler in another language (e.g. **C**)

Full Bootstrapping (2)

- **Step 1:** build a compiler of **Ada$_0$** to **M** in another language, say **C**



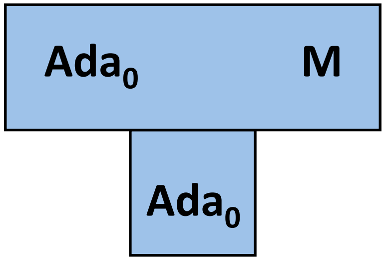
- **Step 2:** compile it using a host compiler of **C** for **M**



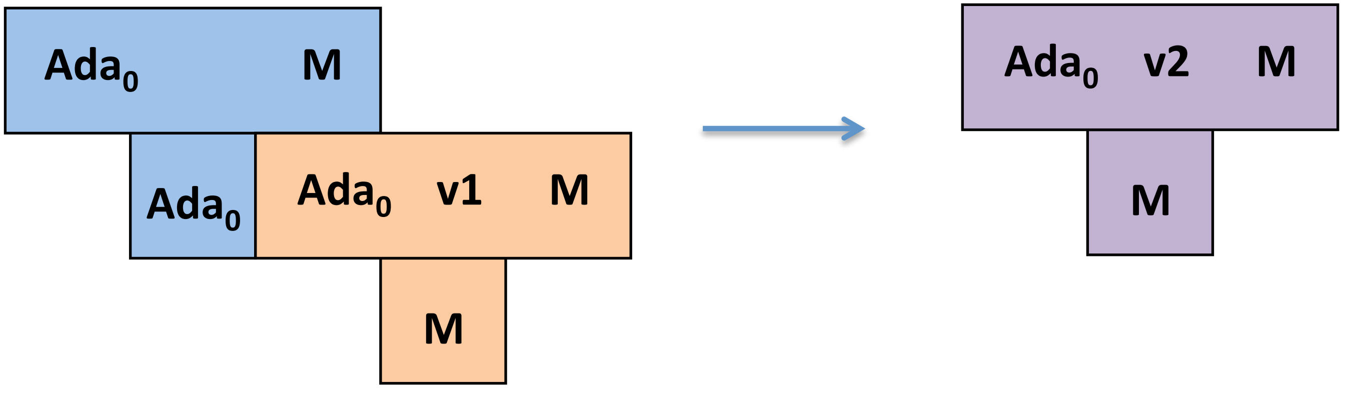
- **Note:** new versions would depend on the **C** compiled for **M**

Full Bootstrapping (3)

- **Step 3:** Build another compiler of **Ada_0** in **Ada_0**



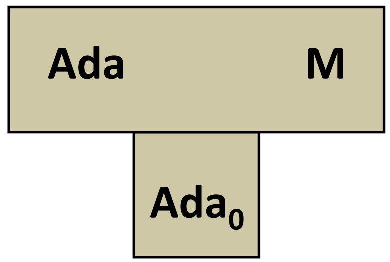
- **Step 4:** compile it using the **Ada_0** compiler for **M**



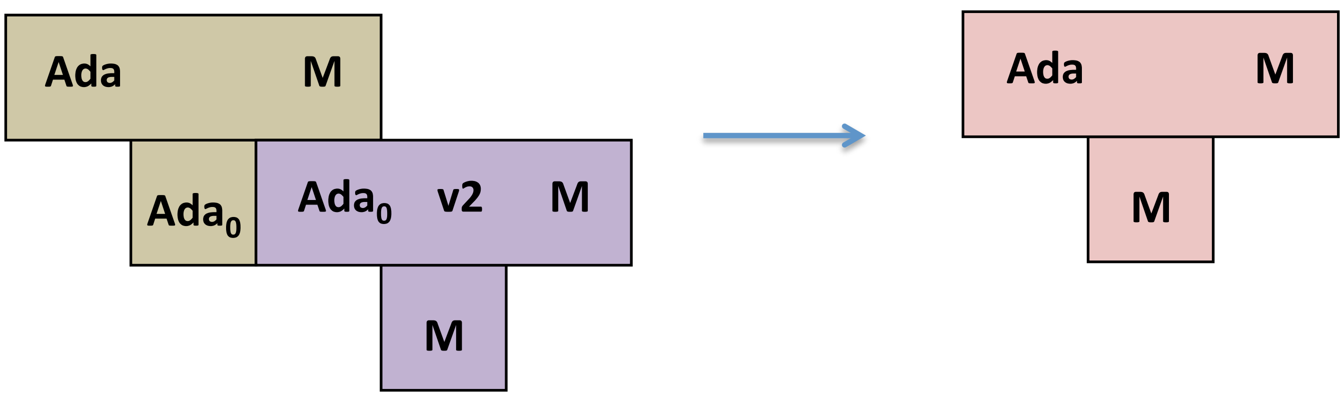
- **Note: C** compiler is no more necessary

Full Bootstrapping (4)

- **Step 5:** Build a full compiler of **Ada** in **Ada_0**



- **Step 4:** compile it using the second **Ada_0** compiler for **M**



- Future versions of the compiler can be written directly in Ada

Names, Binding and Scope: Summary

- Abstractions and names
- Binding time
- Object lifetime
- Object storage management
  - Static allocation
  - Stack allocation
  - Heap allocation

Name and abstraction

- Names used by programmers to refer to variables, constants, operations, types, …
- Names are fundamental for abstraction mechanisms
  - Control abstraction:
    - Subroutines (procedures and functions) allow programmers to focus on manageable subset of program text, hiding implementation details
    - Control flow constructs (if-then, while, for, return) hide low-level machine ops
  - Data abstraction:
    - Object-oriented classes hide data representation details behind a set of operations
- Abstraction in the context of high-level programming languages refers to the degree or level of working with code and data
  - Enhances the level of machine-independence

Binding Time

- A **binding** is an association between a **name** and an **entity**
- An entity that can have an associated name is called **denotable**
- **Binding time** is the time at which a *decision is made* to create a name ↔ entity binding (the actual binding can be created later):
  - **Language design time**: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  - **Language implementation time**: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc. (if not fixed by the language specification)

Binding Time (2)

- **Program writing time**: the programmer's choice of algorithms and data structures
- **Compile time**: the time of translation of high-level constructs to machine code and choice of memory layout for data objects
- **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable (e.g. external names are bound)
- **Load time**: when the operating system loads the executable in memory (e.g. physical addresses of static data)
- **Run time**: when a program executes

Binding Time Examples

- Language design:
  - Syntax (names ↔ grammar)
    - **`if (a>0) b:=a;`** (C syntax style)
    - **`if a>0 then b:=a end if`** (Ada syntax style)
  - Keywords (names ↔ builtins)
    - **`class`** (C++ and Java), **`endif`** or **`end if`** (Fortran, space insignificant)
  - Reserved words (names ↔ special constructs)
    - **`main`** (C), **`writeln`** (Pascal)
  - Meaning of operators (operator ↔ operation)
    - **`+`** (add), `%` (mod), **`**`** (power)
  - Built-in primitive types (type name ↔ type)
    - float, short, int, long, string

Binding Time Examples (cont'd)

- Language implementation
  - Internal representation of types and literals (type ↔ byte encoding, if not specified by language)
    - 3.1 (IEEE 754) and "foo bar" (\0 terminated or embedded string length)
  - Storage allocation method for variables (static/stack/heap)
- Compile time
  - The specific type of a variable in a declaration (name↔type)
  - Storage allocation mechanism for a global or local variable (name↔allocation mechanism)

Binding Time Examples (cont’d)

- Linker
  - Linking calls to static library routines (function↔address)
    - `printf` (in libc)
  - Merging and linking multiple object codes into one executable
- Loader
  - Loading executable in memory and adjusting absolute addresses
    - Mostly in older systems that do not have virtual memory
- Run time
  - Dynamic linking of libraries (library function↔library code)
    - DLL, dylib
  - Nonstatic allocation of space for variable (variable↔address)
    - Stack and heap

The Effect of Binding Time

- ***Early binding times*** (before run time) are associated with greater efficiency and clarity of program code
  - Compilers make implementation decisions at compile time (avoiding to generate code that makes the decision at run time)
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads
- ***Late binding times*** (at run time) are associated with greater flexibility (but may leave programmers sometimes guessing what's going on)
  - Interpreters allow programs to be extended at run time
  - Languages such as Smalltalk-80 with polymorphic types allow variable names to refer to objects of multiple types at run time
  - Method binding in object-oriented languages must be late to support **dynamic binding**
- Usually "**static**" means "before runtime", **dynamic** "at runtime"

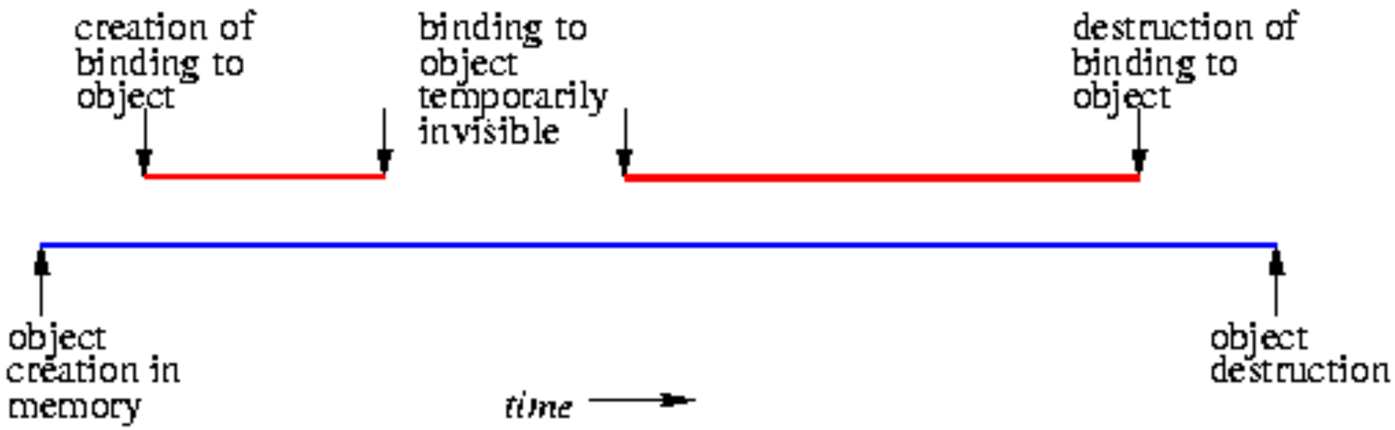
Binding Lifetime versus Object Lifetime

- Key events in object lifetime:
  - Object creation
  - Creation of bindings
  - The object is manipulated via its binding
  - Deactivation and reactivation of (temporarily invisible) bindings
  - Destruction of bindings
  - Destruction of objects

- **Binding lifetime**: time between creation and destruction of binding to object
  - Example: a pointer variable is set to the address of an object
  - Example: a formal argument is bound to an actual argument

- **Object lifetime:** time between creation and destruction of an object

Binding Lifetime versus Object Lifetime (cont'd)



- Bindings are temporarily invisible when code is executed where the binding (name ↔ object) is out of scope
- **Memory leak**: object never destroyed (binding to object may have been destroyed, rendering access impossible)
- **Dangling reference**: object destroyed before binding is destroyed
- **Garbage collection**: prevents these allocation/deallocation problems

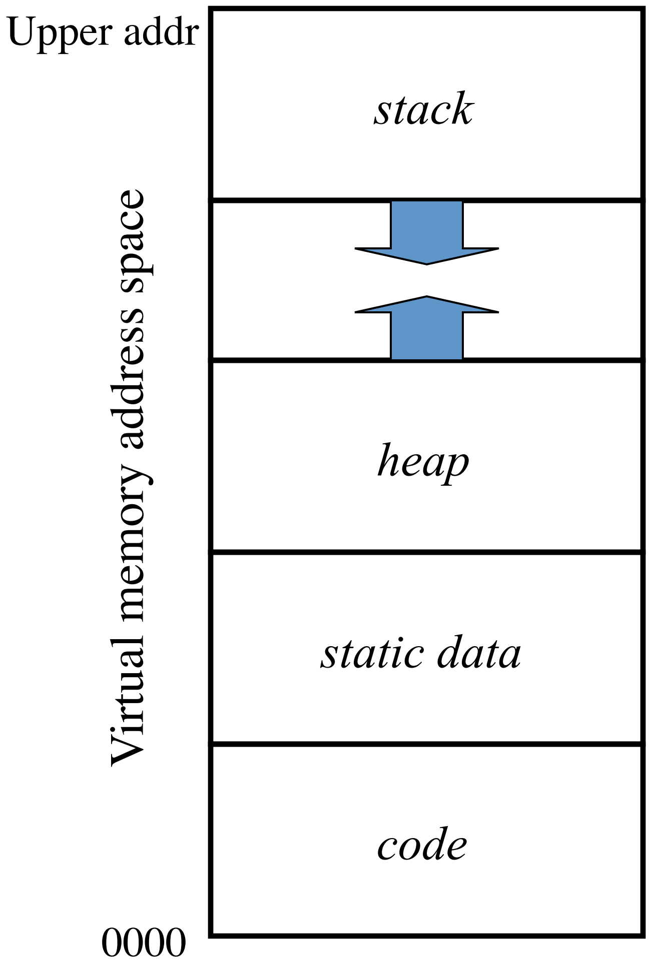
C++ Example

```
{
  SomeClass* myobject = new SomeClass;
  ...
  {
    OtherClass myobject;
    ... // the myobject name is bound to other object
    ...
  }
  ... // myobject binding is visible again
  ...
  myobject->action() // myobject in action():
                     // the name is not in scope
                     // but object is bound to 'this'
  delete myobject;
  ...
  ... // myobject is a dangling reference
}
```

Object Storage

- Objects (program data and code) have to be stored in memory during their lifetime
- **Static objects** have an absolute storage address that is retained throughout the execution of the program
  - Global variables and data
  - Subroutine code and class method code
- **Stack objects** are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
  - Actual arguments passed by value to a subroutine
  - Local variables of a subroutine
- **Heap objects** may be allocated and deallocated at arbitrary times, but require an expensive storage management algorithm
  - Example: Lisp lists
  - Example: Java class instances are always stored on the heap

Typical Program and Data Layout in Memory

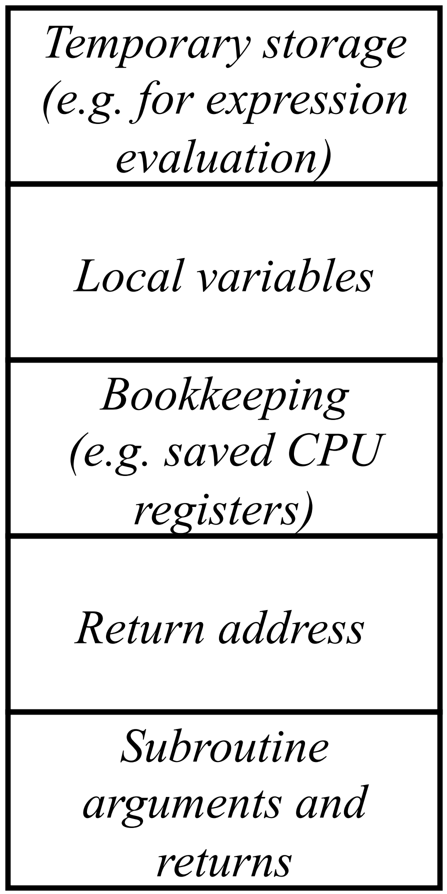


- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

Static Allocation

- Program code is statically allocated in most implementations of imperative languages
- Statically allocated variables are **history sensitive**
  - Global variables keep state during entire program lifetime
  - Static local variables in C functions keep state across function invocations
  - Static data members are “shared” by objects and keep state during program lifetime
- Advantage of statically allocated object is the fast access due to absolute addressing of the object
  - So why not allocate local variables statically?
  - Problem: static allocation of local variables cannot be used for recursive subroutines: each new function instantiation needs fresh locals

Static Allocation in Fortran 77



Typical static subroutine frame layout

- Fortran 77 has no recursion
- Global and local variables are statically allocated as decided by the compiler
- Global and local variables are referenced at absolute addresses
- Avoids overhead of creation and destruction of local objects for every subroutine call
- Each subroutine in the program has a ***subroutine frame*** that is statically allocated
- This subroutine frame stores all subroutine-relevant data that is needed to execute

Stack Allocation

- Each instance of a subroutine that is active has a *subroutine frame* (sometimes called *activation record*) on the run-time stack
  - Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards
  - Method invocation works the same way, but in addition methods are typically dynamically bound
- Subroutine frame layouts vary between languages, implementations, and machine platforms