Principles of Programming Languages

Names, Binding and Scope

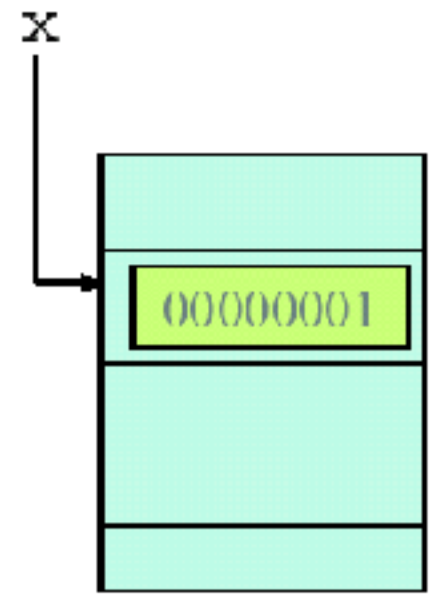
Overview

Study fundamental semantic issues of variables:

- Attributes of variables (Sec. 5.1, 5.2, 5.3)
  - Name, value, type, address, lifetime, scope
- Binding of variables (Sec. 5.4)
- Scope and lifetime (Sec. 5.5, 5.6)
- Referencing environments (Sec. 5.7)
- Named constants (Sec. 5.8)

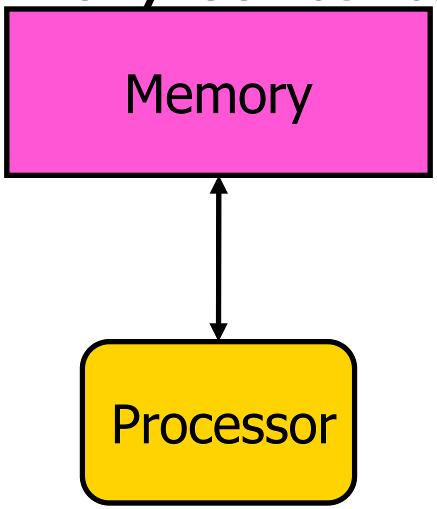
The Concept of Variables

- What do x = 1 ; and x = x + 1; mean?
  - “=“ is not the equal sign of mathematics!
  - “x=1” does not mean “x is one” !
- “x” is the name of a variable; refer to a variable
  - A variable is an abstraction of a memory cell
  - “x” is an identifier (name) refers to a location where certain values can be stored.



Imperative Lang. and von Neumann

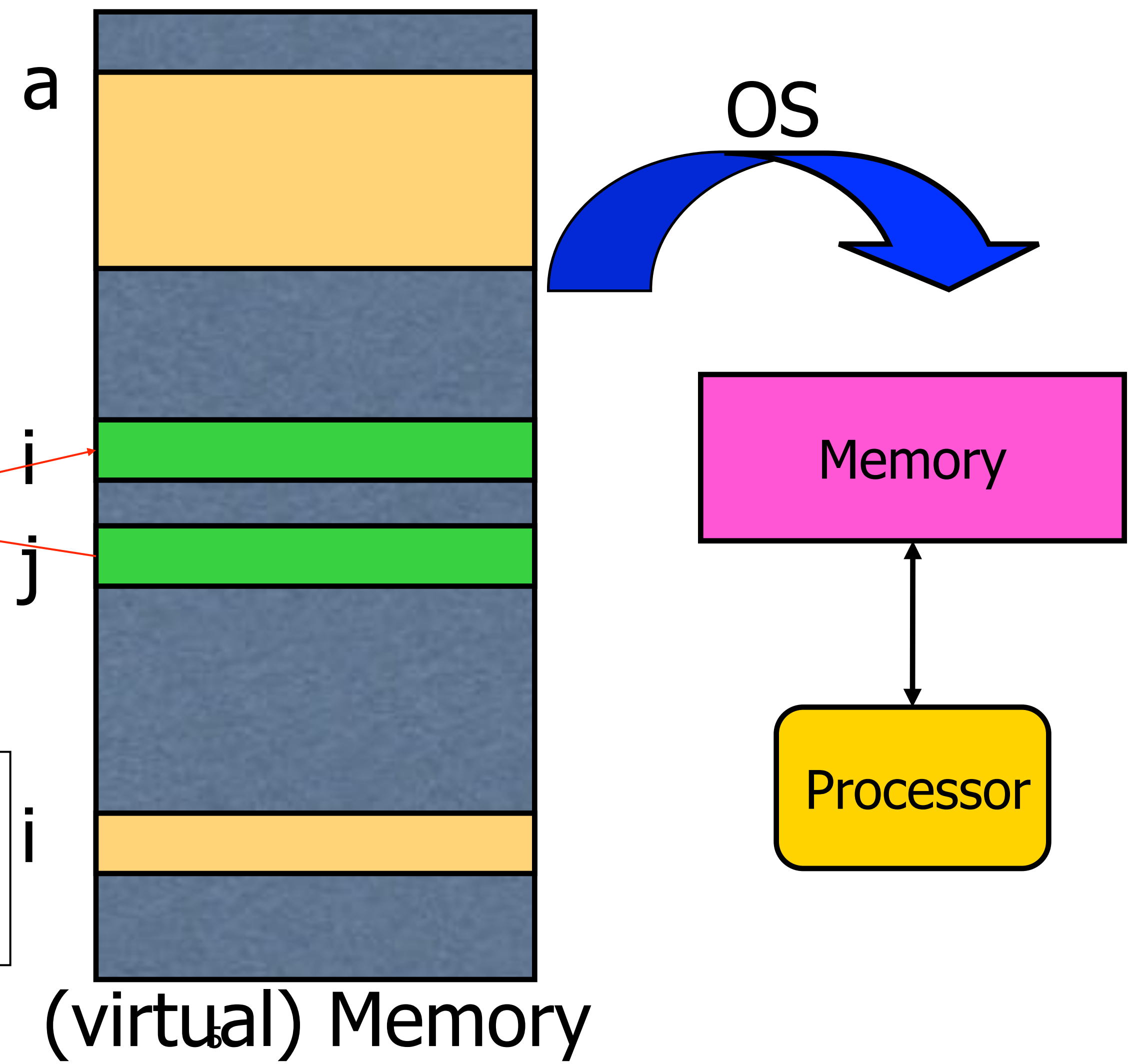
- Imperative languages are abstractions of von Neumann architecture
- Key components of von Neumann architecture:
  - Memory: store data and instructions
  - Processor: do operations to modify memory contents
- Imperative language abstractions:
  - Variables ←→ memory cells
  - Expressions ←→ CPU executions



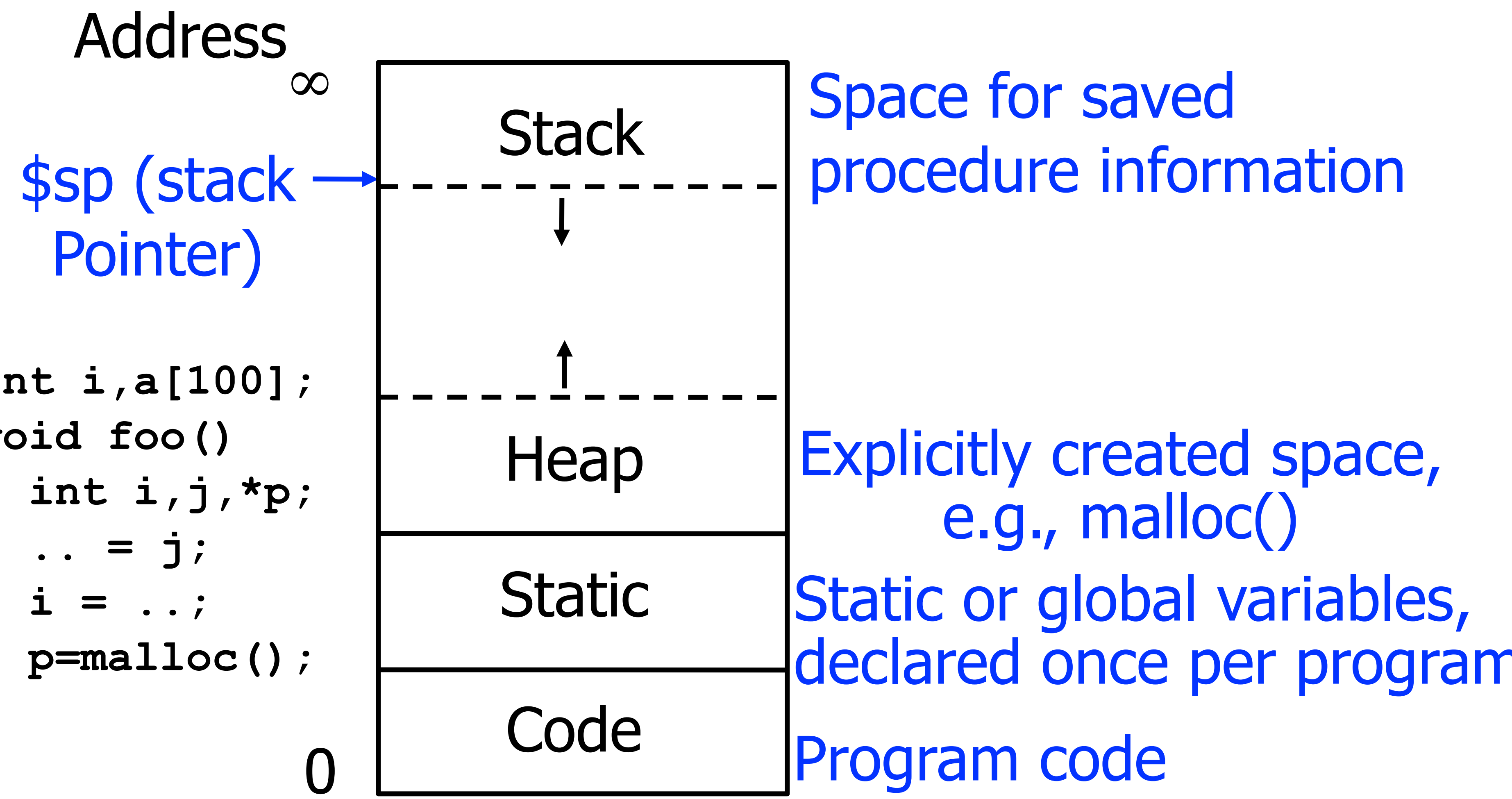
Programmers' View of Memory

```
int i,a[100];


void foo()

{ int i,j;

   .. = j;

   i = ..;

}
```

- How variables are defined?
- How are they associated with storage?



General Memory Layout




```
int i,a[100];
void foo()
{ int i,j,*p;
  .. = j;
  i = ..;
  p=malloc();
}
```

Variable Attributes: Name, Value

- Name: also called identifier
  - Length:
    - Most modern PL do not impose a limit
  - Connectors: e.g., underscore "_"
    - Model PL prefer camel notation, e.g., **`MyStack`**
  - Case sensitivity:
    - C, C++, and Java names are case sensitive
    - Names that look alike are different: `rose`, `ROSE`
- Value:
  - The contents of the location with which the variable is associated

Variable Attributes: Type

- Type:
  - Determines (1) the range of values that the variable can store; and (2) the set of operations that can be performed for values of that type
  - Uses of type system: error detection through type checking, program modularization, documentation
  - Can a variable have different types at different times?
  - Are two given types equivalent?
  - What happen when two variables of different types operate together?
  - Type will be discussed in length in next chapter

Variable Attributes: Address

- Which address is variable **`i`** bound?
  - What if foo() is in recursion?
- Which binding is visible in which part of the code?
- For how long does a binding last?

→ Binding and lifetime

```
int
  i,a[100];
void foo()
{ int i,j;
   .. = j;
   i = ..;
}
void bar()
{ int j;
   .. = j;
   i = ..;
}
```

The Concept of Binding

- Binding: an association, e.g. between a variable and its storage or its type
- Possible binding times
  - Language design time: bind operator symbols to operations
  - Language implementation time: bind floating point type to a representation
  - Compile time: bind a variable to a type in C or Java
  - Load time: bind a FORTRAN 77 variable to a memory cell (or a C `static` variable)
  - Runtime: bind a local variable to a memory in stack

Static and Dynamic Binding

- A binding is static if it first occurs before run time and remains unchanged throughout program execution
- A binding is dynamic if it first occurs during execution or can change during execution

```
int i,a[100];

void foo()
{ int i,j;
  .. = j;
  i = ..;
}
```

Storage Bindings and Lifetime

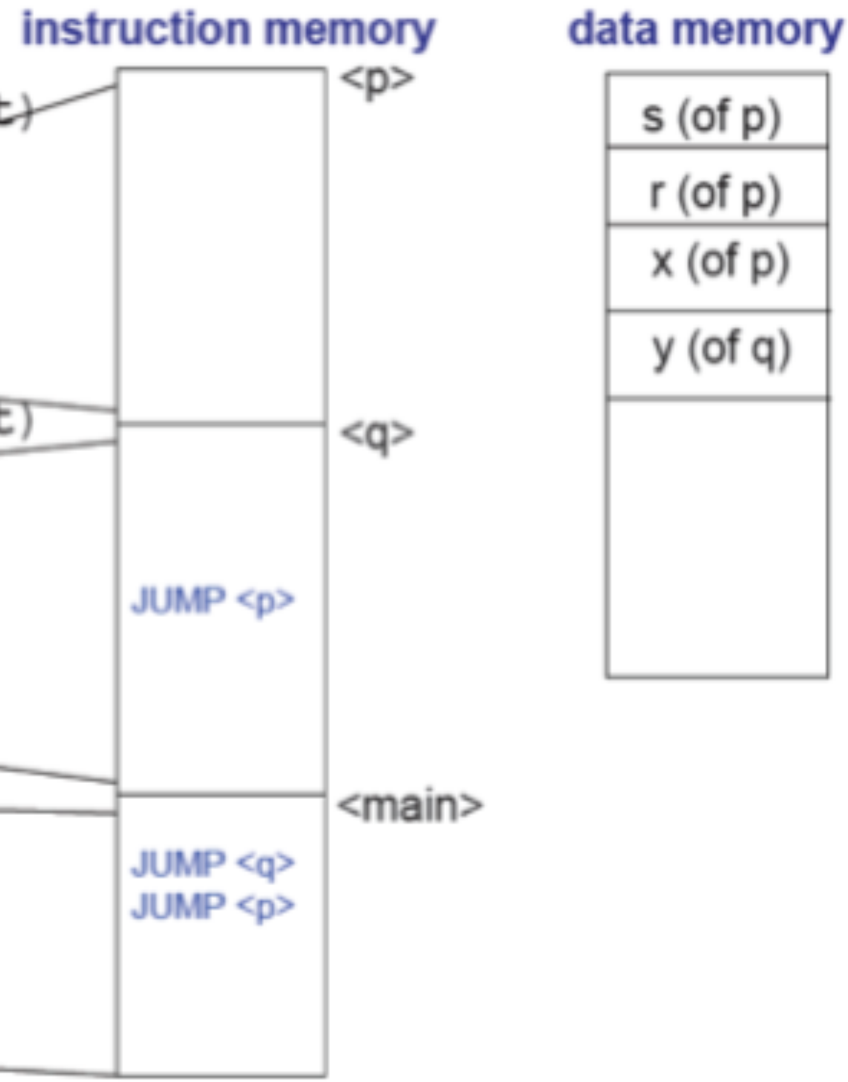
- Storage binding:
  - Allocation: get a cell from some pool of available cells
  - Deallocation: put a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell
  - Starts when a variable is bound to a specific cell and ends when it is unbound
  - Four categories for scalar variables according to lifetimes: static, stack-dynamic, explicit-heap-dynamic, implicit heap-dynamic

Static Variables

- Bound to memory cells before execution begins and remains throughout execution, e.g., all FORTRAN 77, C static and global variables
- Advantages:
  - Efficiency (direct addressing, no runtime allocation overhead), for globally accessible variables, history-sensitive subprogram support
- Disadvantage:
  - Lack of flexibility (no recursion), no sharing of storage among variables
  - Size of data objects must be known at compile time
  - Data structures cannot be created dynamically

Static Memory Model Cannot Support Recursion

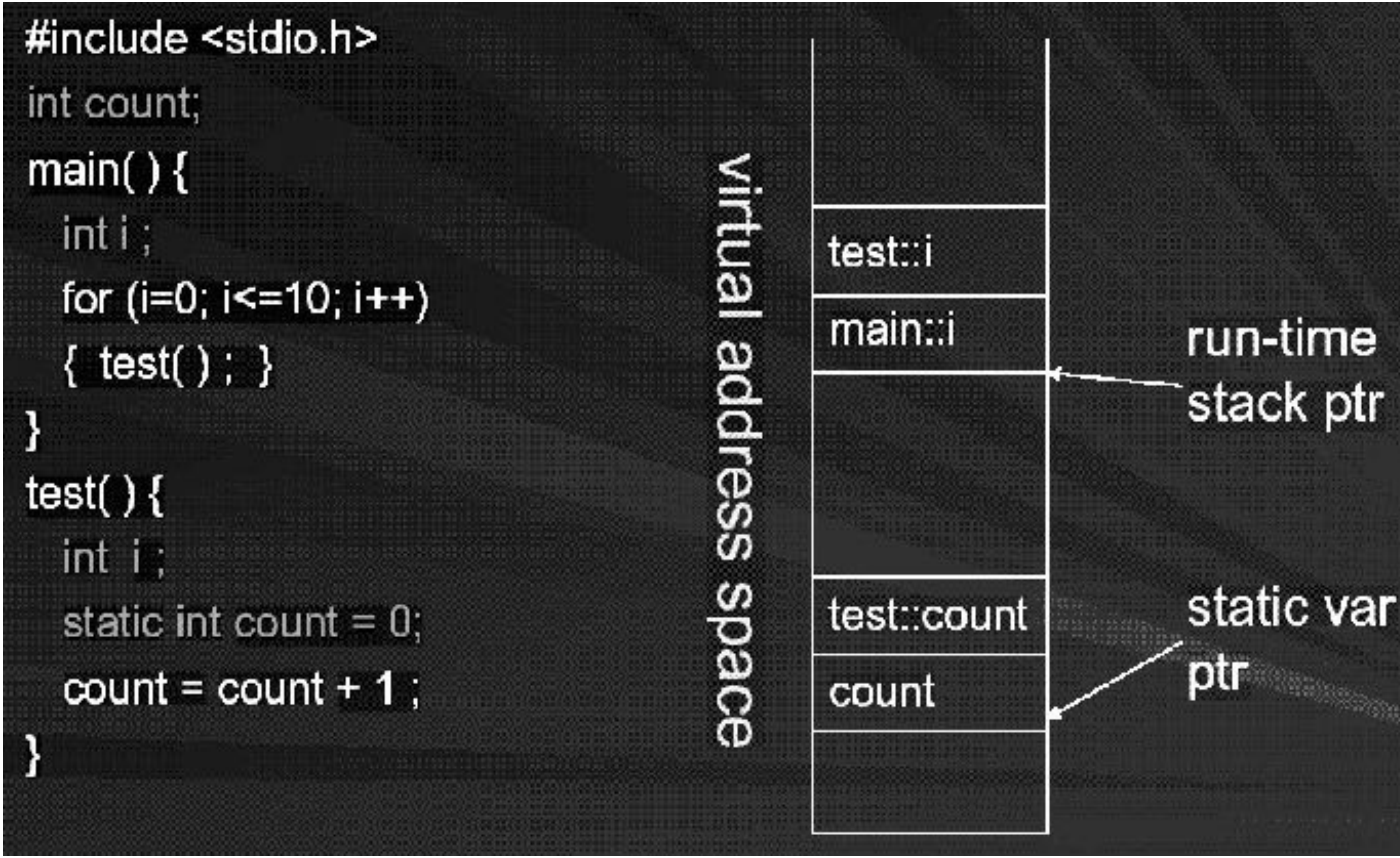
```
program test
  procedure p(x : short)
    var s, r: short;
    begin
     ...P(0);
    end
  procedure q(y : short)
    begin
      ...
      p(4);
      ...
    end
  begin
    q(2);
    p(3);
    ...
  end
```



Stack-dynamic Variables

- Storage bindings are created when declaration statements are elaborated at runtime
  - If scalar, all attributes except address are statically bound, e.g., local variables in C subprograms and Java methods, allocated from the runtime stack
- Advantage:
  - Allows recursion; conserves storage by all subprog.
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Stack-dynamic Variables



Explicit Heap-dynamic Variables

- Heap: a section of virtual address space reserved for dynamic memory allocation
- Allocated and deallocated (in heap) by explicit directives or operators, specified by the programmer, which take effect during execution
  - Referenced through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`)
  - Static type binding, dynamic storage binding
  - Explicit or implicit deallocation (garbage collection)

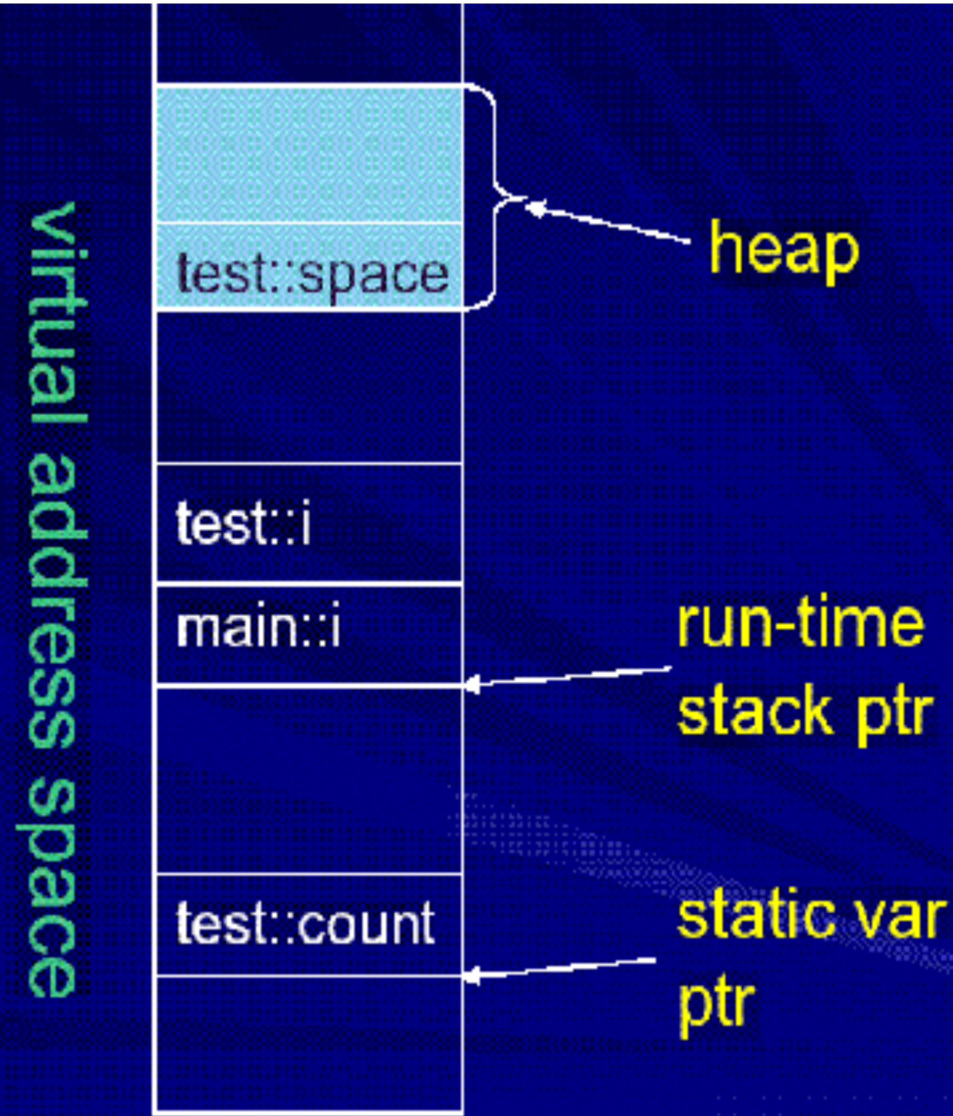
Explicit Heap-dynamic Variables

```
Person *p;
p=(Person *) malloc(sizeof Person);
p->name = "Mike";p->age = 40;
free(p);
```

- Java objects are explicit heap-dynamic variable
  - implicit garbage collection (no free or delete)
- Advantage: can construct dynamic structures
- Disadvantage: inefficient, unreliable, heap management cost

Explicit Heap-dynamic Variables

```
#include <stdio.h>
main( ) {
  int  i ;
  test( ) ;
}
test( ) {
   int  i ;
   char  *space;
   static int count = 0;
   space = (char *) malloc(64);
   // memory leak
}
```



Implicit Heap-dynamic Variables

- Allocation and deallocation caused by assignment statements, regardless of what the variable was previously used for
  - All variables in APL; all strings and arrays in Perl and JavaScript
- Advantage: flexibility
- Disadvantages:
  - Runtime overhead for maintaining dynamic attributes
  - Loss of error detection by compiler

Why Scope?

- Two bindings for "x"
  - One of type `int`
  - Another `float`

An occurrence of "x"

```
int x;
void foo (int y)
{ float x;
  ... x ...
}
int main() {
    ...
    foo(x);
  // which binding of x?
    ...
}
```

Scope

- The scope of a variable is the range of statements over which it is visible
  - A variable is visible in a statement if it can be referenced in that statement
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how a reference to a name is associated with a variable and its declaration

Scope

- "**int X;**" has a global scope
- "float X;" has a local scope.

```
int X - 0;
void foo (int X)
{
  float X;
   X = …    // which X?
}
int main() {
  int y;
  foo(X); // which X?
   ...
}
```

Static Scope

- Scope of a variable can be statically determined
  - Based on program text, a spatial concept
- To connect a name reference to a variable, you (or the compiler) must find the declaration
  - First search locally, then in increasingly larger enclosing scopes, until one is found for the given name, or an undeclared variable error

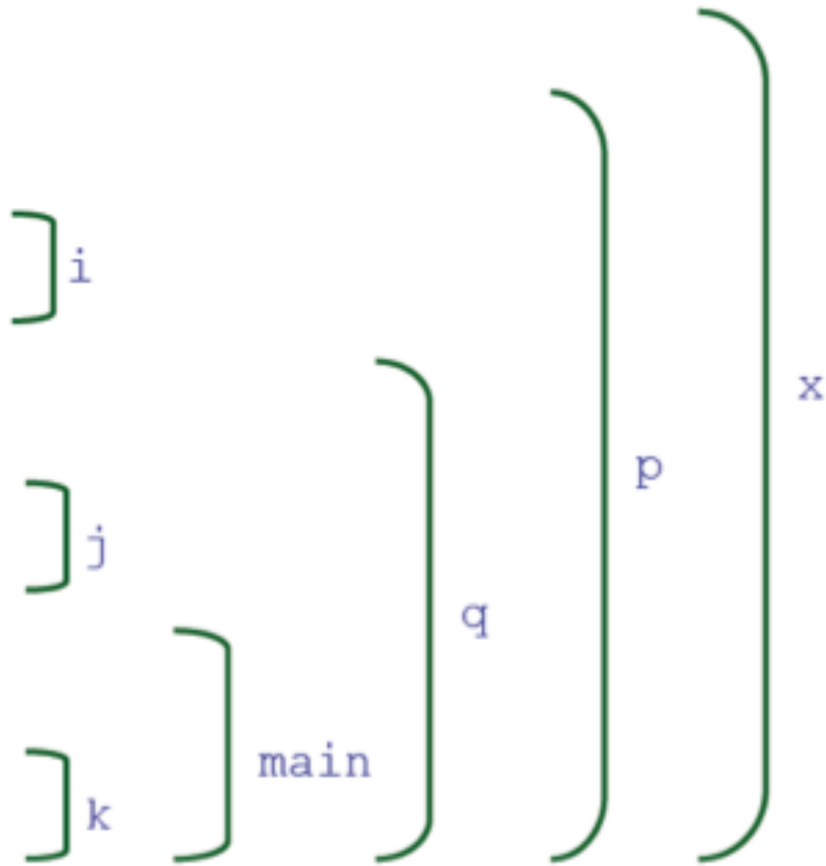
Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name
  - C++ and Ada allow access to "hidden" variables:
    `unit.name` (in Ada)
    `class_name::name` (in C++)
- Block: a method of creating new static scopes inside program units (from ALGOL 60)
  - e.g.: C and C++ in any compound statement

```
 for (...) {
int index;
          ...
  }
```

An Example of Block Scope in C

```c
int x;
void p(void)
{
    int i;  ...
}
void q(void)
{
    int j;  ...
}
main()
{
    int  k;  ...
}
```



Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
  - Can only be determined at run time
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
MAIN
    - declaration of x
        SUB1
         - declaration of x
         ...
         call SUB2
         ...


       SUB2
        ...
        - reference to x
        ...

    ...
    call SUB1
    ...
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Static scoping:
Reference to x in SUB2 is to MAIN's x

Dynamic scoping:
Reference to x in SUB2 is to SUB1's x

Static vs. Dynamic Scoping

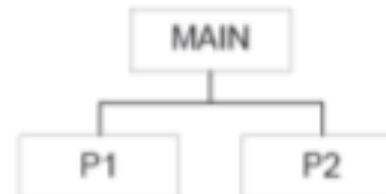
```
program MAIN;
  var a : integer;

  procedure P1;
  begin
    print a;
  end; {of P1}

  procedure P2;
  var a : integer;
  begin
      a := 0;
      P1;
  end; {of P2}

  begin
      a := 7;
      P2;
  end. {of MAIN}
```

static (lexical)

non-local variables are bound based on program structure

if not local, go "out" a level

→ example prints 7

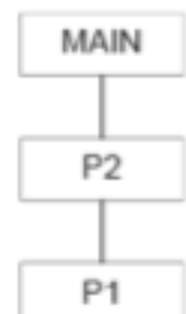


dynamic

non-local variables are bound based on calling sequence

if not local, go to calling point

→ example prints 0



Evaluation of Dynamic Scope

- Advantage:
  - Convenience (no need to pass parameters from caller to callee)
- Disadvantage:
  - Local variables of an active subprogram are visible to any other active subprograms → reliability
  - Cannot statically type check references to nonlocals
  - Poor readability
  - Longer accesses to nonlocal variables

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Ex.: a `static` variable in a C or C++ function
  - Scope is static and local to the function
  - Lifetime extends over entire execution of program
- Ex.: subprogram calls

```
void printheader() { … }
void compute() {
  int sum; …        //scope vs lifetime of sum
  printheader();
}
```

Referencing Environments

- Referencing environment of a statement is the collection of all names visible to the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
  - A subprogram is active if its execution has begun but has not yet terminated

Referencing Environments

```
procedure Example is
  A, B : Integer;
  ...
  procedure Sub1 is
    X, Y : Integer;
    begin  -- of Sub1
    ...  <---------------- 1
    end;  -- of Sub1
  procedure Sub2 is
    X : Integer;
    ...
    procedure Sub3 is
      X : Integer;
      begin  -- of Sub3
      ...  <------------- 2
      end;  -- of Sub3
    begin  -- of Sub2
    ...  <---------------- 3
    end;  -- of Sub2
  begin  -- of Example
  ...  <------------------ 4
  end.  -- of Example
```

Referencing Environment

X and Y of Sub1, A and B of Example

X of Sub3, (X of Sub2 is hidden), A and B of Example

X of Sub2, A and B of Example

Named Constants

- A named constant is a variable that is bound to a value only once, when it is bound to storage
  - Advantages: readability, modifiability, can be used to parameterize programs
- Binding of values to named constants can be static (called manifest constants) or dynamic
  - FORTRAN 90: only constant-valued expressions
  - Ada, C++, Java: expressions of any kind
- The binding of a variable to a value at the time it is bound to storage is called initialization
  - Often done on declaration statement

Summary

- Variables are abstractions for memory cells of the computer and are characterized by name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities: type and storage binding
- Scope determines the visibility of variables in a statement