Chapter 5 Names, Bindings, and Scopes

This chapter introduces the fundamental semantic issues of variables. The attributes of variables, including **type**, **address**, and **value**, are then discussed.

5.1 Introduction

- What are variables?
  - The abstractions in a language for the memory cells of the machine
- A variable can be characterized by a collection of properties, or attributes
  - Type (the most important)
  - Scope
  - Lifetime

5.2 Names

- Names are also associated with subprograms, formal parameters, and other program constructs.
- Identifier ≡ Name

5.2.1 Design Issues

- Are names case sensitive?
- Are the special words of the language reserved words or keywords?

5.2.2 Name Forms

- A **name** is a string of characters used to identify some entity in its names
  - Length limitations are different for different languages
    - C99, Java, C#, Ada, C++
  - Naming convention
    - Underscore characters
    - Camel notation
    - Other: PHP, Perl, Ruby

5.2.2 Name Forms

- Case sensitive
  - To some people, this is a serious detriment to readability
  - Not everyone agrees that case sensitivity is bad for names

5.2.3 Special Words

- Special works in programming languages are used to make programs more readable by naming actions to be performed.
  - They are used to separate the syntactic parts of statements and programs.
  - Keyword and reserved word

5.2.3 Special Words

- A **keyword** is a word of programming language that is special only in certain contexts.
- In Fortran,

```
Integer Apple
Integer = 4
Integer Real
Real Integer
```

5.2.3 Special Words

- A **reserved word** is a special word of a programming language that cannot be used as a name
- In C, Java, and C++

```
int i; /*a legal statement*/
float int; /*an illegal statement*/
```

- COBOL has 300 reserved words,
  - LENGTH, BOTTOM, DESTINATION, COUNT

5.3 Variables

- Definition of variable
  - A program variable is an abstraction of a computer memory cell or collection of cells.
- A variable can be characterized as a sextuple of attributes:
  - (Name, address, type, lifetime, and scope)

5.3.1 Name

- Identifier
- Most variables have names
  - Variables without names
    - Temporary variables
      - E.g. x=y*z+3
        - » The result of y*z may be stored in a temporary variable
    - Variables stored in heap
      - Section 5.4.3.3

5.3.2 Address

- Definition of address
  - The address of a variable is the machine memory address with which it is associated.
- In many language, it is possible for the same variable to be associated with different addresses at different times in the program
  - E.g., local variables in subroutine

5.3.2 Address (Cont'd)

- Address ≡ l-value
- When more than one variable name can be used to access the same memory location, the variables are called aliases.
  - A hindrance to readability because it allows a variable to have its value changes by an assignment to a different variable
    - UNION, pointer, subroutine parameter

5.3.3 Type

- The type of a variable determines the same of values the variable can store and the set of operations that are defined for values of the type.

5.3.4 Value

- The value of a variable is the contents of the memory cell or cells associated with the variable
  - Abstract cells > physical cells
- Value ≡ r-value

5.4 The Concept of Binding

- Definition of binding
  - A binding is an association between an attribute and an entity
    - A variable and its type or value
    - An operation and symbol
- Binding time
  - The time at which a binding takes place

5.4 The Concept of Binding (Cont'd)

- When can binding take place?
  - Language design time
  - Language implementation time
  - Compile time
  - Load time
  - Link time
  - Run time
- Check the example in the first para. of Section 5.4 and make sure you understand it.

5.4 The Concept of Binding (Cont'd)

- Consider the Java statement:

```
count = count + 5;
```

  - The type of `count`
  - The set of possible values of `count`
  - The meaning of operator "+"
  - The internal representation of literal "`5`"
  - The value of `count`

5.4.1 Binding of Attributes toVariables

- Static binding
  - Occurs before run time begins and remains unchanged throughout program execution
- Dynamic binding
  - Occurs during run time or can change in the course of program execution

5.4.2 Type Bindings

- Before a variable can be referenced in a program, it must be bound to a data type

5.4.2.1 Static Type Binding

- Static type binding ≃ Variable declaration
  - Explicit declaration
    - A declaration statement that lists variable names and the specified type
  - Implicit declaration
    - Associate variables with types through default conventions
      - Naming conventions of FORTRAN

5.4.2.1 Static Type Binding (Cont'd)

- Although they are a minor convenience to programmers, implicit declarations can be detrimental to reliability
  - Prevent the compilation process from detecting some typographical and programmer errors
  - Solution:
    - FORTRAN: declaration `Implicit none`
    - Specific types to begin with particular special characters
      - Perl: $, @, %
    - Type inference in C#

5.4.2.2 Dynamic Type Binding

- The type of a variable is not specified by a declaration statement
- The variable is bound to a type when it is assigned a value in an assignment statement
- Advantage:
  – It provides more programming flexibility
    - Generic program to deal with data for any numeric type

5.4.2.2 Dynamic Type Binding (Cont’d)

- Before the mid-1990s, the most commonly used programming languages used static type binding
- However, since then there has been a significant shift languages that use dynamic type bindign
  - Python, Ruby, JavaScript, PHP, …

5.4.2.2 Dynamic Type Binding (Cont'd)

- JavaScript

```
List = [10.2, 3.5];
...
List = 47;
```

- C# 2010
  - "`any`" can be assigned a value of any type. It is useful when data of unknown type come into a program from an external source

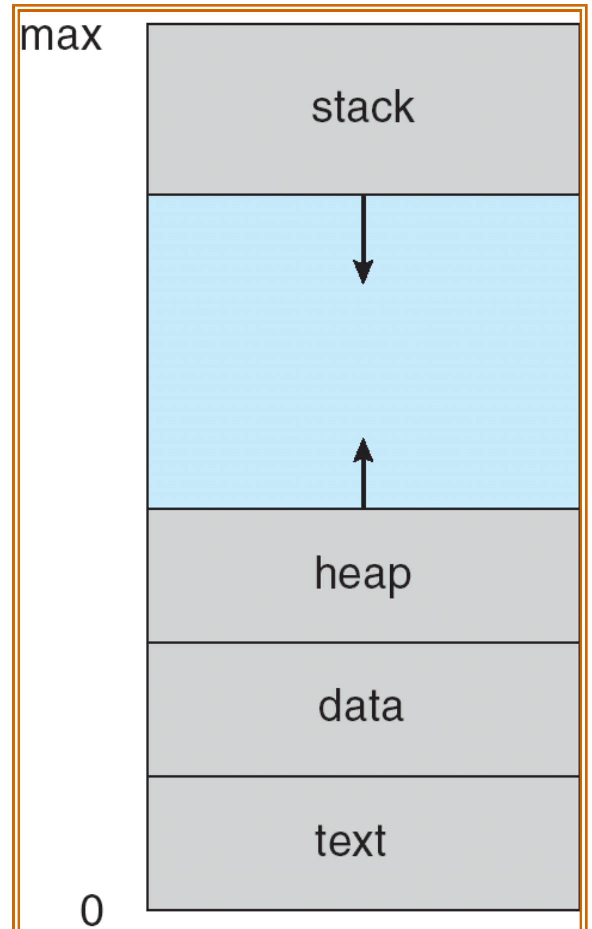
```
dynamic any;
```

5.4.2.2 Dynamic Type Binding (Cont'd)

- Disadvantages:
  - It causes programs to be less reliable
    - Error-detection capability of the compiler is diminished
      - Incorrect types of right sides of assignments are not detected as errors
        - » E.g., *keying err*or of "i=x;" and "i=y;".
  - Cost
    - Type checking must be done at run time
      - Run-time descriptor
      - Storage of a variable must be of varying size
  - Usually implemented using pure interpreters

5.4.3 Storage Bindings and Lifetime

- Process in Memory

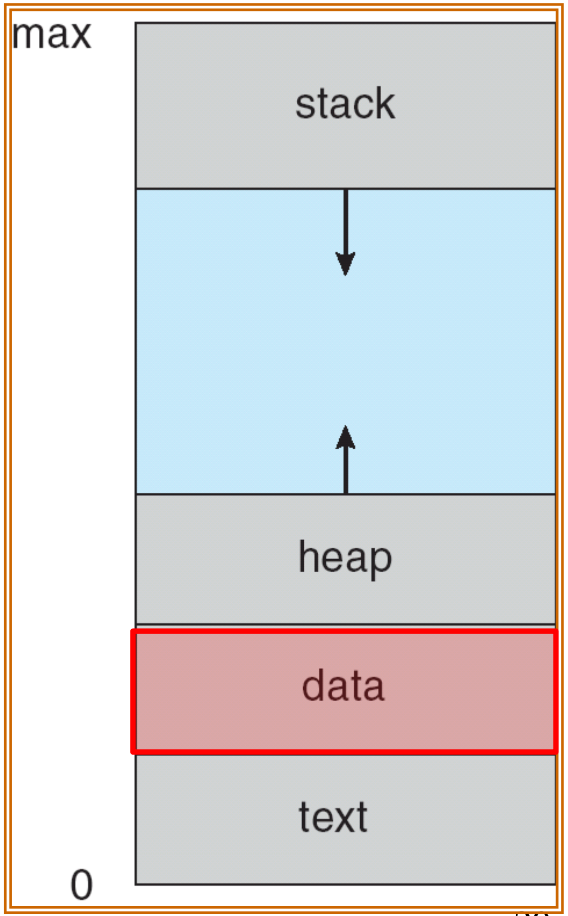


5.4.3 Storage Bindings and Lifetime (Cont’d)

- Allocation
  - The memory cell to which a variable is bound somehow must be taken from a pool of available memory
- Deallocation
  - Placing a memory cell that has been unbound from a variable back into the pool of available memory
- Lifetime
  - The time during which the variable is bound to a specific memory location

5.4.3.1 Static Variables

- Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates
  - Globally accessible variables
  - History sensitive



5.4.3.1 Static Variables (Cont'd)

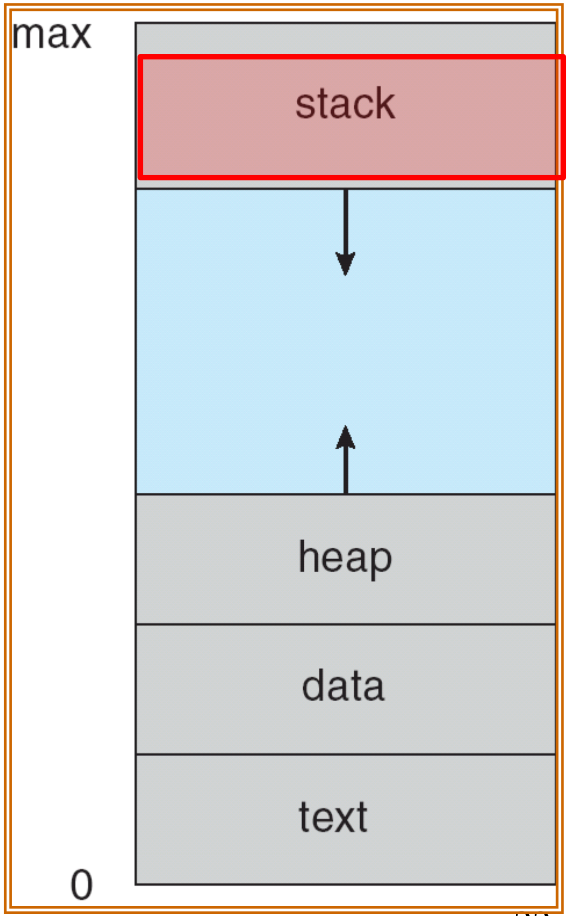
- Advantage:
  - Efficiency
    - Direct addressing
    - No run-time overhead for allocation and deallocation
- Disadvantage:
  - Cannot support recursive
  - Storage cannot be shared among variable
- C and C++
  - "`static`" specifier on a variable definition in a function

5.4.3.2 Stack-Dynamic Variables

- Storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.
  - **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
    - Occurs during run time

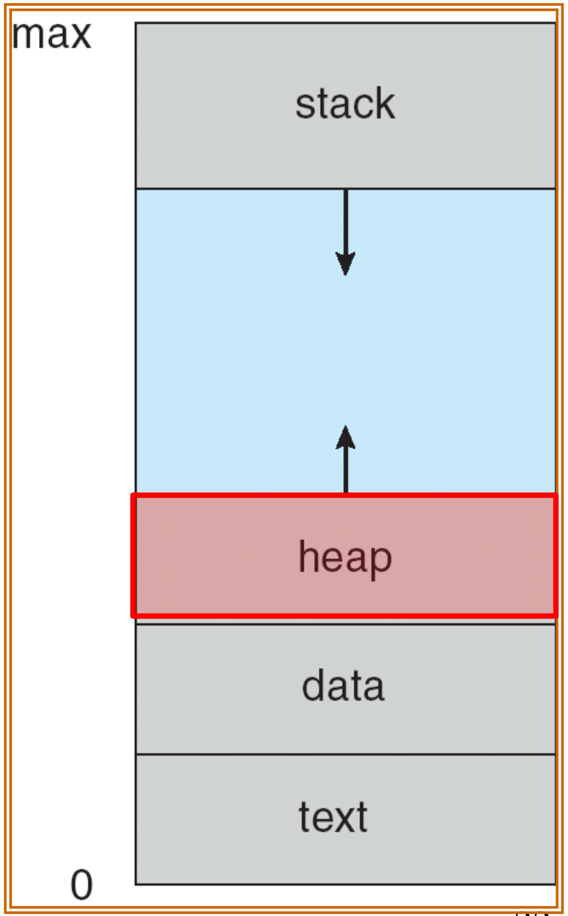
5.4.3.2 Stack-Dynamic Variables

- Stack-dynamic variables are allocated from the run-time stack
- Advantages
  - Recursive subprograms support
  - Storage sharing
- Disadvantages
  - Indirect addressing
  - Overhead for allocation and deallocation



5.4.3.3 Explicit Heap-Dynamic Variables

- Explicit Heap-Dynamic variables are nameless memory cells that are allocated and deallocated by explicit run-time instructions
  – Allocated from and deallocated to the heap, can only be referenced through pointer or reference variables



5.4.3.3 Explicit Heap-Dynamic Variables (Cont'd)

- C++

```
int *intnode;
intnode = new int;
…
delete intnode;
```

- C

```
 int *ptr = malloc(sizeof(int));
*ptr = 200;
…
free(ptr);

int *arr = malloc(1000 * sizeof(int));
…
free(ptr);
```

5.4.3.3 Explicit Heap-Dynamic Variables (Cont'd)

- Java
  - Java objects are explicit heap dynamic and are accessed through reference variables
- Usage:
  - Explicit heap-dynamic variables are often used to construct dynamic structures,
    - Linked lists and trees, that need to grow and/or shrink during execution

5.4.3.3 Explicit Heap-Dynamic Variables (Cont’d)

- Disadvantage
  - Difficulty of using pointer and reference variable correctly
  - Cost of references to the variables
  - Complexity of the required storage management implementation

5.4.3.4 Implicit Heap-Dynamic Variables

- Variables bound to heap storage only when they are assigned values
- All attributes are bound every time they are assigned
- E.g., JavaScript

```
highs=[74, 84, 86, 90, 71];
```

5.4.3.4 Implicit Heap-Dynamic Variables (Cont'd)

- Advantages:
  - High degree of flexibility
  - Allowing highly generic code to be written
- Disadvantages:
  - Run-time overhead of maintaining all the dynamic attributes
    - Array subscript types and ranges
    - Loss of some error detection by the compiler

5.5 Scope

- The scope of a variable is the range of statements in which the variable is visible.
  - A variable is visible in a statement if it can be referenced in that statement.
- Local & non local variables

5.5.1 Static Scope

- ALGOL 60 introduced the method of binding names to nonlocal variables call **static scoping**
  - The scope of a variable can be statically determined
    - Prior to execution

5.5.1 Static Scope (Cont’d)

- Two categories of static scoped languages
  - Subroutine can be nested
    - Nested static scopes
    - E.g., Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python
  - Subroutine cannot be nested
    - E.g. , C-based language

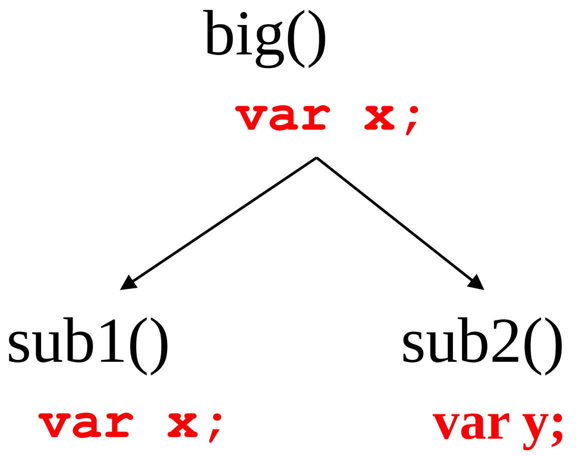
5.5.1 Static Scope (Cont'd)

- How to find a reference to a variable in static-scoped language?
  - Suppose a reference is made to a variable `x` in subprogram `sub1`.
  - The correct declaration is found by first searching the declarations of subprogram `sub1`.
  - If no declaration is found for the variable there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is call its **static parent**.

5.5.1 Static Scope (Cont'd)

- A JavaScript function

```
function big() {
 function sub1(){
   var x=7;
   sub2(); }
 function sub2() {
   var y=x; }
 var x=3;
 sub1();
}
```



5.5.1 Static Scope (Cont'd)

- Static ancestor
- Hidden
  - The outer `x` is hidden from `sub1`.
- Hidden variables can be accessed in some languages
  - E.g., Ada

    `big.x`

5.5.2 Blocks

- Many languages allow new static scopes to be defined in the midst of executable code
  - Originated from ALGOL 60
  - Allows a section of code to have its own local variables whose scope is minimized
    - Defined variables are typically static dynamic
  - Called a **block**
    - Origin of the phrase **block-structured language**

5.5.2 Blocks (Cont'd)

- Many languages allow new static scopes to be defined in the midst of executable code
  - Originated from ALGOL 60
  - Allows a section of code to have its own local variables whose scope is minimized
    - Defined variables are typically static dynamic
  - Called a **block**
    - Origin of the phrase **block-structured language**

5.5.2 Blocks (Cont'd)

- The scopes created by blocks, which could nested in larger blocks, are treated exactly like those created by subprograms
  - legal in C and C++, but not in Java and C# - too error-prone

```
void sub() {
        int count;
        while (...) {
          int count;
          count++;
          ...
        }
        …
      }
```

5.5.3 Declaration order

- In C89, all data declarations in a function except those in nested blocks must appear at the beginning of the function
- However, C99, C++, Java, JavaScript, C##, allow variable declarations to appear anywhere
  - Scoping rules are different

5.5.4 Global Scope

- In C, C++, PHP, JavaScript, and Python, variable definitions can appear outside all the functions
  - Create global variables, which potentially can be visible to those functions

5.5.4 Global Scope (Cont'd)

- C, C++ have both declarations and definitions of global data.
  - *Declarations* specify types and other attributes but do not cause allocation of storage.
  - *Definitions* specify attributes and cause storage allocation
  - For a specific global name, a C program can have any number of compatible declaration, but only a single definition

5.5.4 Global Scope (Cont’d)

- A declaration of variable outside function definitions specifies that the variable is defined in a different file.

```
extern int sum;
```

5.5.4 Global Scope (Cont'd)

- The idea of declaration and definition carries over to the functions of C and C++.

```
main(){
    int foo(int);
    …
}
int foo(int x;)
{
   …
}
```

A prototype, declaration

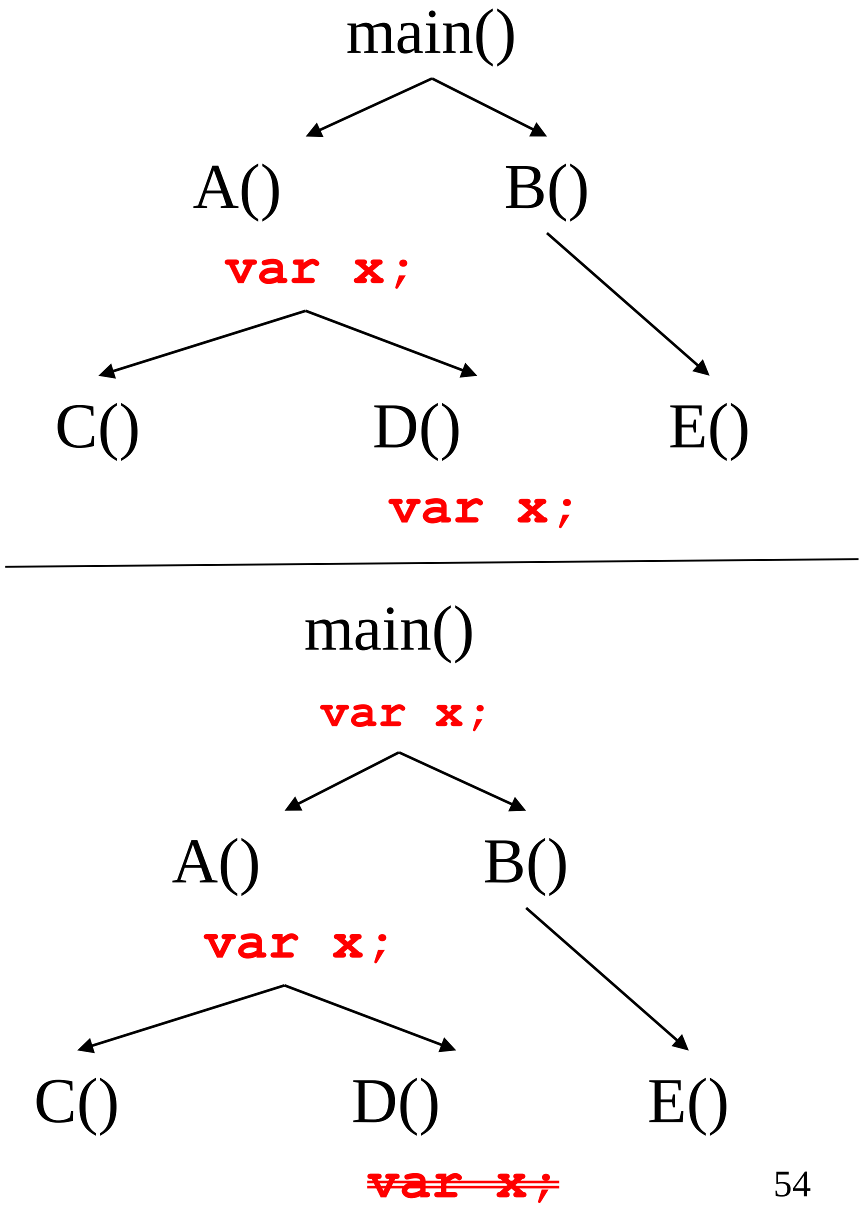
A function definition

5.5.4 Global Scope (Cont'd)

- Check the global scope rule of
  - C++
  - PHP
  - JavaScript
  - Python

5.5.5 Evaluation of Static Scope

- Problems of static scoping
  - In most cases it allows more access to both variables and subprograms than is necessary
  - Software is highly dynamic – programs that are used regularly continually change.
    - E.g., `E()` wants to access `x` in `D()`



5.5.6 Dynamic Scope

- **Dynamic scoping** is based on the calling sequence of subprogram, not on their spatial relationship to each other.
  - The scope can be determined only at run time.

5.5.6 Dynamic Scope (Cont'd)

- Consider the following two calling sequences:
  - `big` calls `sub1`, `sub1` calls `sub2`
  - `big` calls `sub2`

```
function big() {
 function sub1(){
   var x=7;
   sub2(); }
 function sub2() {
   var y=x;
   var z=3; }
 var x=3;
 sub1();
 sub2();
}
```

5.5.7 Evaluation of Dynamic Scoping

- Problems follow directly from dynamic scoping:
  - No way to protect local variables from this accessibility
  - In ability to type check references to nonlocals directly
  - Make programs much more difficult to read
  - Slow in referencing nonlocal variables

5.5.7 Evaluation of Dynamic Scoping (Cont'd)

- Merit:
  - The parameters passed from one subprogram to another are variables that are defined in the caller.
  - None of these needs to be passed
- Dynamic scoping is not widely used
  - LISP replaced dynamic scope with static scope

5.6 Scope and Lifetime (Cont'd)

- The apparent relationship between scope and lifetime does not hold in other situation
  - Second para.
  - E.g., The lifetime of sum extends over the time during which `printheader` executes.

```
void printheader() {
… }
void compute() {
  int sum;
  …
  printheader(); }
```

5.7 Referencing Environments

- The referencing environment of a statement is the collection of all variables that are visible in the statement
  - In a static scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes
  -

5.7 Referencing Environments (Cont'd)

- For dynamic scoped language:
  - A subprogram is **active** if its execution has begun but has not yet terminated
  - The reference environment in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active.

5.8 Named Constants

- A name constant is variable that is bound to a value only once.
  - Useful as aids to readability and program reliability
- E.g.
  - In Java,
    - `final int len=100;`
  - C++ allow dynamic binding of values to named constants, in C++:
    - `const int result =  2* width +1 ;`