Lecture 4
Memory Management

Dr. Wilson Rivera

ICOM 4036: Programming Languages

Electrical and Computer Engineering Department
University of Puerto Rico

Some slides adapted from Sebesta's textbook

Lecture Topics

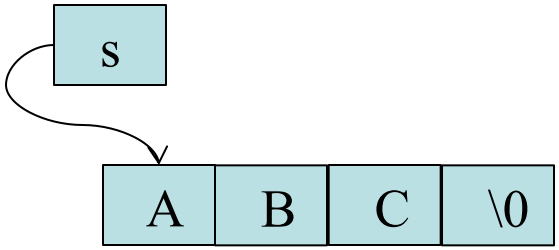
- Pointers
- Data types
- Stack vs. Heap
- Activation records
- Passing parameters
- Scope
  - Static vs. Dynamic

Arrays and Pointers

```
#include <string.h>
#define max 100

Char c='a', *p, s[max];

p=&c;
strcpy(s,"ABC");
```



*p	a
*p+1	b
*p+2	c
s	ABC
*s	A
*s+6	G
*s+7	H
s+1	BC
p=s	A
s=p	Syntax error

$$s[i] \equiv *(s+i)$$

$$\&\, s[i] \equiv (s+i)$$

Problems with Pointers

- Dangling pointers
  - Occurs when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used (by then the memory may have been re-assigned to another use)
    - A new heap-dynamic variable is created and pointer p1 is set to point at it
    - The heap-dynamic variable pointed to p2 is explicitly de-allocated but p1 is not changed by the operation. P1 is now a dangling pointer

```
Int * arrayPtr1;
Int *arrayPrt2 = new int [100];
arrayPrt1=arrayPrt2;
Delete [] arrayPrt2;
```

Problems with Pointers

- Lost heap-dynamic variable
  - Occurs when a program fails to free memory occupied by objects that will not be used again (memory leakage)
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable

```
Int * arrayPtr1 = new int [100];
…….
Int * arrayPrt1=new int [200];
```

Garbage Collection

Garbage collection, John McCarthy (1959) for LIPS

- Advantages: reduce certain bugs
  - Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used.
  - Double free bugs, which occur when the program attempts to free a region of memory that is already free.
  - Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that will not be used again, leading, over time, to memory exhaustion.

Garbage Collection

- Disadvantages:
  - Garbage collection is a process that consumes limited computing resources in deciding what memory is to be freed and when
  - In a logical memory leak, a region of memory is still referenced by a pointer, but is never actually used. Garbage collectors generally can do nothing about logical memory leaks.
  - Poor locality (interacting badly with cache and virtual memory systems), occupying more address space than the program actually uses at any one time, and touching otherwise idle pages.
    - Thrashing: a program spends more time copying data between various grades of storage than performing useful work

Data Types

- char → 1 byte
- short → 2 byes
- int → 4 bytes
- float → 4 bytes
- double → 8 bytes

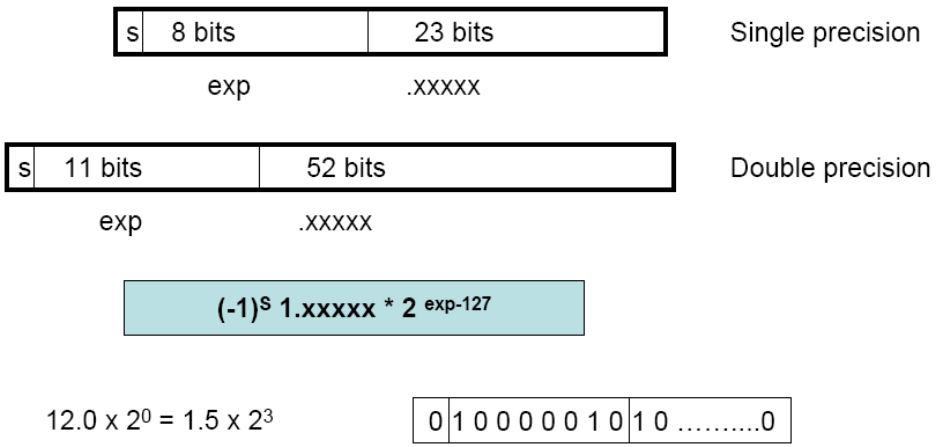
$$\text{int a} = 2^{23} + 2^{21} + 2^{14} + 7;$$
$$\text{short b} = \text{a};$$

a → 00000000 10100000 01000000 00000111

b → 01000000 00000111

Data Types

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more
- IEEE Floating-Point Standard 754



Data Types

```
Short a=45;
double b=*(double *) &a;
```


```
int i=37;
float f=*(float *)&i;
```

- F is a very small number! Why“?

Type Binding

Static

- Ada
- Java
- C, C++, C#,
- F#
- Fortran
- Haskell
- ML
- Objective-C
- Perl (built in types)

Dynamic

- Python
- Ruby
- Erlang
- Groovy
- JavaScript
- Lisp
- Objective-C
- Perl
- PhP
- Prolog
- Smalltalk

Type Binding

- A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

- *Static Type Binding*
  - if it occurs in compile time and remains unchanged throughout program execution.
  - Explicit/implicit declarations
  - May increase reliability
  - Restricts program flexibility
    - List elements in Haskell must be same type while array elements in JavaScript may be different type

- *Dynamic Type Binding*
  - if it first occurs during execution or can change during execution of the program
  - Reduce production cycle but slows down execution

Dynamic Type Binding

- Specified through an assignment statement
  e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage:
    - flexibility (generic program units)
    - Allows interpreters to dynamically load new code
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Reliability: Type error detection by the compiler is difficult

Explicit/Implicit Type Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables

- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
  - FORTRAN, JavaScript, Ruby, Python, and Perl provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability
  - Disadvantage: reliability

- A language can be Statically typed without requiring type declarations (e.g. Haskell, F#)

Type Inference

- Type Inference in ML, Miranda, Haskell, Ada, C# (3.0), F#, Visual Basic, Python
  - Guaranteed to produce most general type
  - Determine the best type for an expression based on known information about symbols in the expression
  - ML examples
    - `fun circumf(s)=3.14*s*s;`
    - `fun square(x)=x*x;`
    - `fun square(x): real = x*x;`
    - `fun square(x)=x*(x : real);`

Type Conversions

- A *mixed-mode expression* is one that has operands of different types

```
int a;
float b,c,d;
d=b*a; /* suppose we type a instead of c */
```

- *coercion* is an implicit type conversion
  - e.g. because mixed mode expression are legal in Java, the compiler will not detect an error. It will insert code to coerce the value of the `int` a to `float`
  - In most languages, all numeric types are coerced in expressions, using widening conversions
  - In Ada, there are virtually no coercions in expressions

- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler

Type Conversions

- Assignment statements can also be mixed-mode
  - In Fortran, Perl, C, and C++, any numeric type value can be assigned to any numeric type variable
  - In Java and C#, only widening assignment coercions are done

Type Checking

X=5

Y="37"

X+Y

Visual Basic → 42

JavaScript → 537

Python → Error

Strong Typing

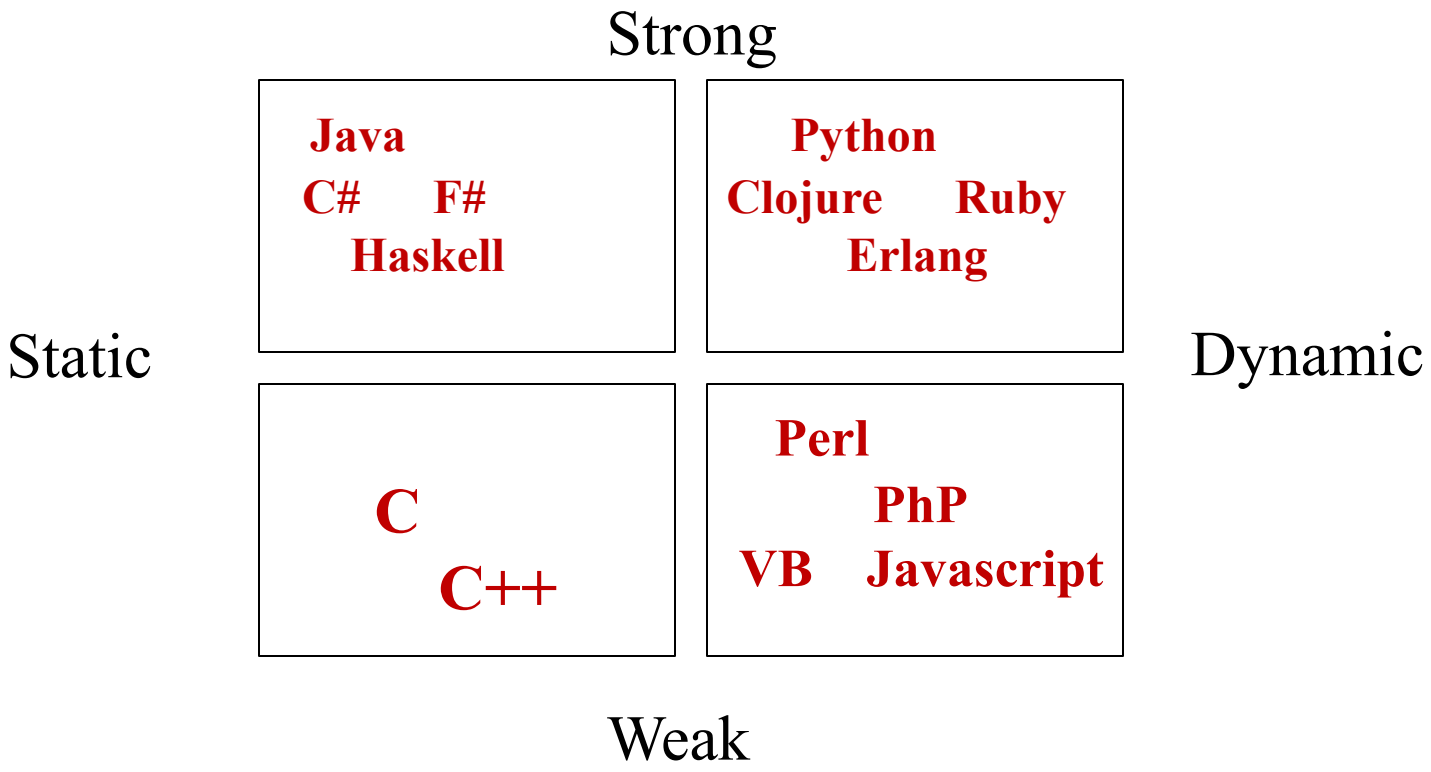
- Strong Typing
  - Python, Ruby, Haskell, F#
  - Ada, almost (`UNCHECKED CONVERSION` is loophole)
  - Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada
- Non strong Typing
  - Objective-C, Perl
  - FORTRAN 95 is not: parameters, `EQUIVALENCE`
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked

Weak Typing

```
# sample in Perl
a=2
b='2'
concatenate(a,b) # returns '22'
add(a,b) #returns 4
```

- Cost of weak type languages
  - TLS heartbeat buffer read overrun in OpenSSL (Heartbleed)

Type systems



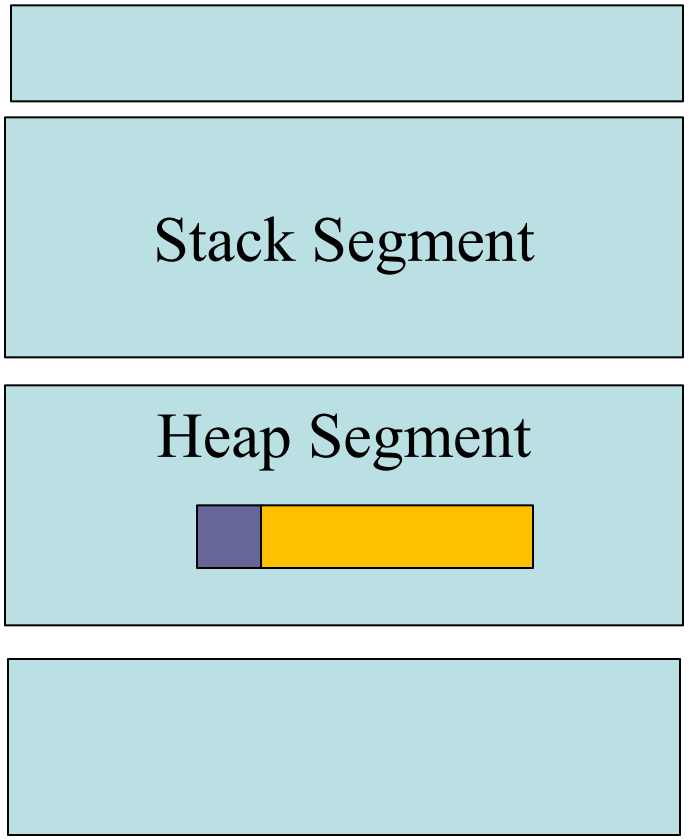
Gradual Typing

Choose how much typing is wanted

Language	Gradual Typing
PhP	Hack (Facebook)
Racket	Static Racked
Python	myPy
JavaScript	TypeScript
	Dart (Google)
	Swift (Apple)

Stack versus Heap

```
int *a = malloc(40*sizeof(int));
```



- Reserve 160+4 bytes of the heap segment
- Extra space (4-8 bytes) to include information about the memory

Stack versus Heap

```
int foo()
{
  char *pBuffer; // Allocated on the stack
  bool b = true; // Allocated on the stack.
  if(b)
  {
    //Creates 500 bytes on the stack
    char buffer[500];

    //Creates 500 bytes on the heap
    pBuffer = new char[500];

   }//<-- buffer is deallocated here, pBuffer is not
}//<memory leak, I should have called delete[] pBuffer;
```

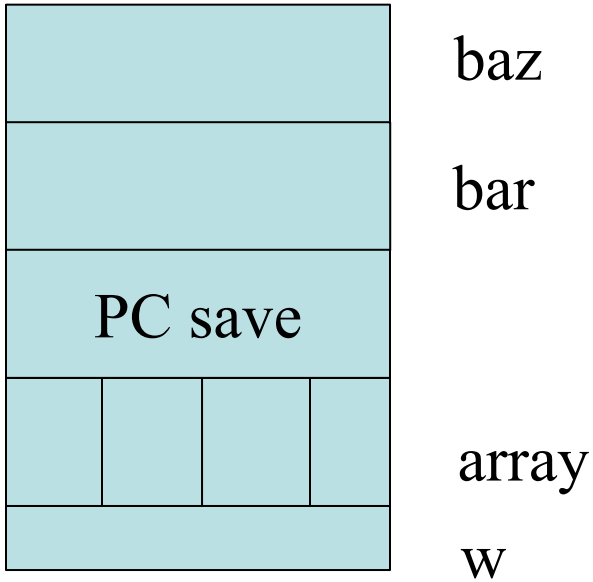
Stack versus Heap

- Stack
  - The stack is the memory set aside as scratch space for a thread of execution
  - The stack is always reserved in a LIFO order
    - The stack is faster because the access pattern makes it trivial to allocate and de-allocate memory
- Heap
  - The heap is memory set aside for dynamic allocation
  - There's no enforced pattern to the allocation and de-allocation of blocks from the heap
  - Important if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data

Activation Record

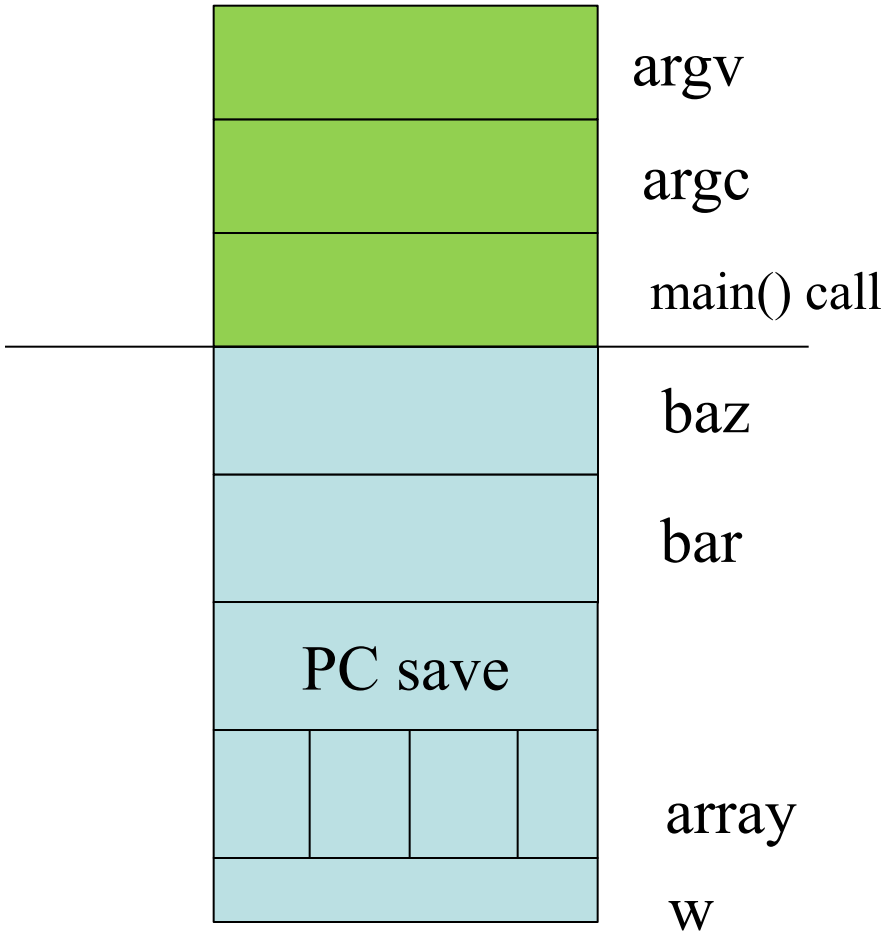
```
void foo(int bar,
         int *baz)
{
  char array[4];
  int *w;


}
```



Activation Record

```
main (int argc,
          char **argv)
{
    int i=4;
    foo(i, &i);
    return 0;
}
```



Parameter Passing

- Passing by value
- Passing by reference

Pass-by-Value

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Advantages: fast for scalars
  - Disadvantages
    - additional storage is required (stored twice) and the actual move can be expensive (for large parameters)

Pass-by-Reference

- Pass an access path
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

Parameter passing in C

```
void swap(int var1,
          int var2)
{
    int temp = var1;
    var1 = var2;
    var2 = temp;
}

main(){
int x=2;
int y=4;
swap(x,y);
printf("%d\n",x);
printf("%d\n",y);}
```

```
void swap(int *var1,
          int *var2)
{
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
}

main(){
int x=2;
int y=4;
swap(&x,&y);
printf("%d\n",x);
printf("%d\n",y);}
```

Parameter passing in Java

```
public class Swap1 {
    public static void main(String[] args){
        int x =7;
        int y = 3;

        swap(x,y);

        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }

    public static void swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
}
```

Parameter passing in Java

```
public class swap{

     public static void kernel(java.awt.Point arg1, java.awt.Point arg2) {
          arg1.x = 100;
          arg1.y = 100;
          java.awt.Point temp = arg1;
          arg1 = arg2;
          arg2 = temp;
          }

     public static void main(String [] args) {
          java.awt.Point pnt1 = new java.awt.Point(0,0);
          java.awt.Point pnt2 = new java.awt.Point(0,0);
          System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
          System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
          System.out.println(" ");          kernel(pnt1,pnt2);
          System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);
          System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);

          }
     }
```

Parameter passing in Python

- In Python lists, tuples, and directories pass around by reference
  - x=[1,2,3,4]
  - y=x
  - w=[x, x]
  - x.append(5)
  - y → [1,2,3,4,5]
  - w→ [[1,2,3,4,5], [1,2,3,4,5]]

From copy import deepcopy
z=deepcopy(x)

Parameter passing in Python

```
def ref_passing(x):
        print "x=",x," id=",id(x)
        x=42
        print "x=",x," id=",id(x)

def main():
        print "passing by object"
        x=9
        print id(x)
        ref_passing(x)
        print id(x)

main()
```

Parameter passing in Python

```
def no_side_effect(list):
        print list
        list = [47,11]
        print list

def side_effect(list):
        print list
        list += [47,11]
        print list
```

```
def main():
        print "\n\nNO side effect"
        x = [0,1,1,2,3,5,8]
        no_side_effect(x)
        print x

        print "\n\nSide effect"
        x = [0,1,1,2,3,5,8]
        side_effect(x)
        print x

        print "\n\n solving side effect"
        x = [0,1,1,2,3,5,8]
        side_effect(x[:])
        print x
```

Parameter Passing in PLs

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
- Java
  - All parameters are passed by value
  - Object parameters are manipulated as references
- Python
  - Pass by reference and by object

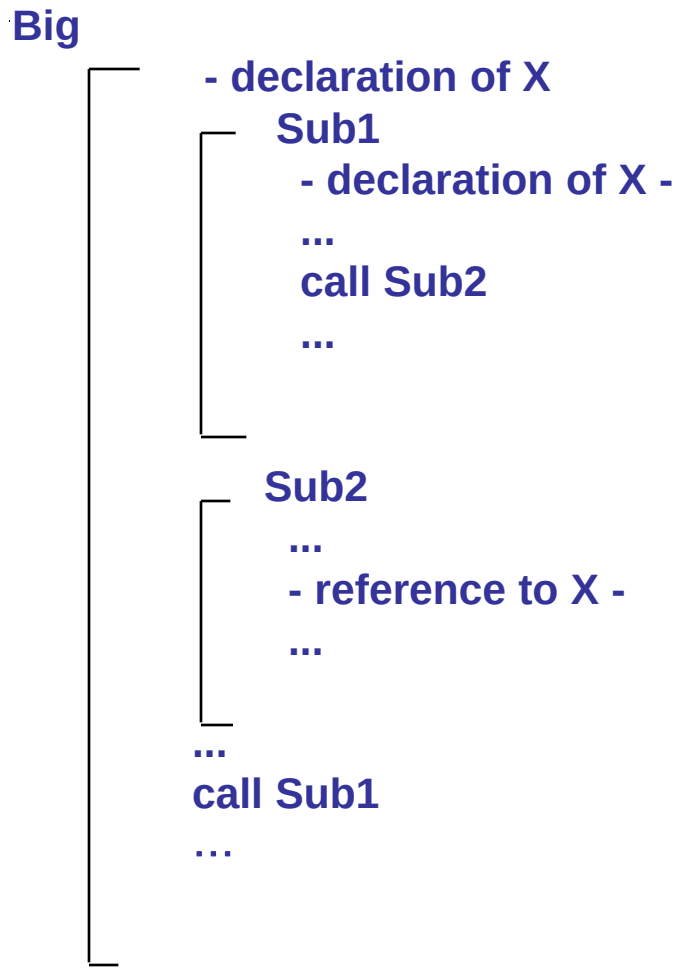
Scope

- The *scope* of a variable is the range of statements over which the variable is visible
- Types of Scope
  - Static
  - Dynamic

Static Scope

- Also called lexical scope
- The scope of the variable is determined prior the execution (compiler time)
- To connect a name reference to a variable, the compiler must find the declaration
  - *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name (code layout)
  - Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*

Static Scope



Case 1:
Big calls Sub1
Sub1 calls Sub2

Case2:
Big calls sub2
directly

Reference to X is to Big's X

Static Scope

- Works well in many situations
  - Allows the compiler to "hard code" information about the variable into the executable code
  - Allows the compiler to perform optimizations based on its knowledge of the variable.
- Problems:
  - In most cases, too much access is possible
    - e.g. all variables declared in the main program are visible to all the procedures whether or not that is desired
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Static Scope: Declaration

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used

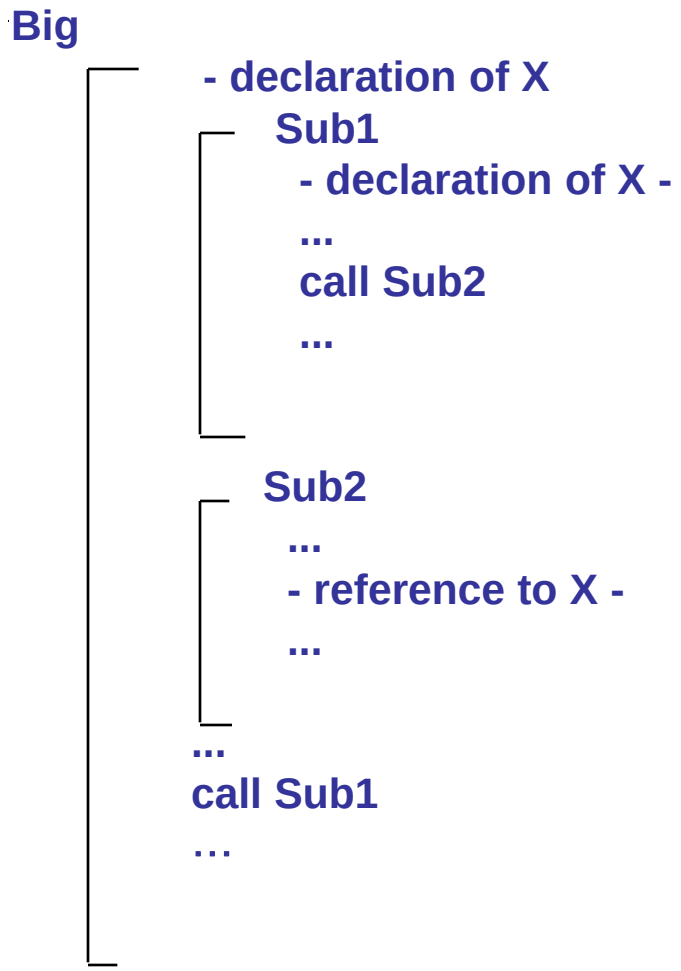
Static Scope: Global variables

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- C and C++ have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file

Dynamic Scope

- Based on calling sequences of program units
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
  - Advantage:
    - A variable's scope could change during the course of execution, or remain undetermined— very flexible.
    - Information about the variable is usually stored with it.
  - *Disadvantages:*
    - While a subprogram is executing, its variables are visible to all subprograms it calls
    - Poor readability– it is not possible to statically determine the type of a variable

Dynamic Scope



Case 1:
Big calls Sub1
Sub1 calls Sub2

Reference to X is to Sub1's X

Case2:
Big calls sub2 directly

Reference to X is to Big's X

Static vs. Dynamic

```
main()
{     int x = 3;

     void f(int x)
     {
          g ();
     }

     void g ()
     {
          print (x);
     }

     void doit ()
     {
          int x = 12;
          f(42);
          g();
     }
}
```

Static 3 and 3
Dynamic 42 and 12

Static vs. Dynamic: Scheme

```
(define add-a
  (let ((a 45))
    (lambda (n) (+ n a))))

(let ((a 12))
  (add-a 15))
```

Lexical: 60

Dynamic: 27

Summary

- Data Types
- Stack vs. Heap
- Passing parameters
- Scope
  - Static vs. Dynamic