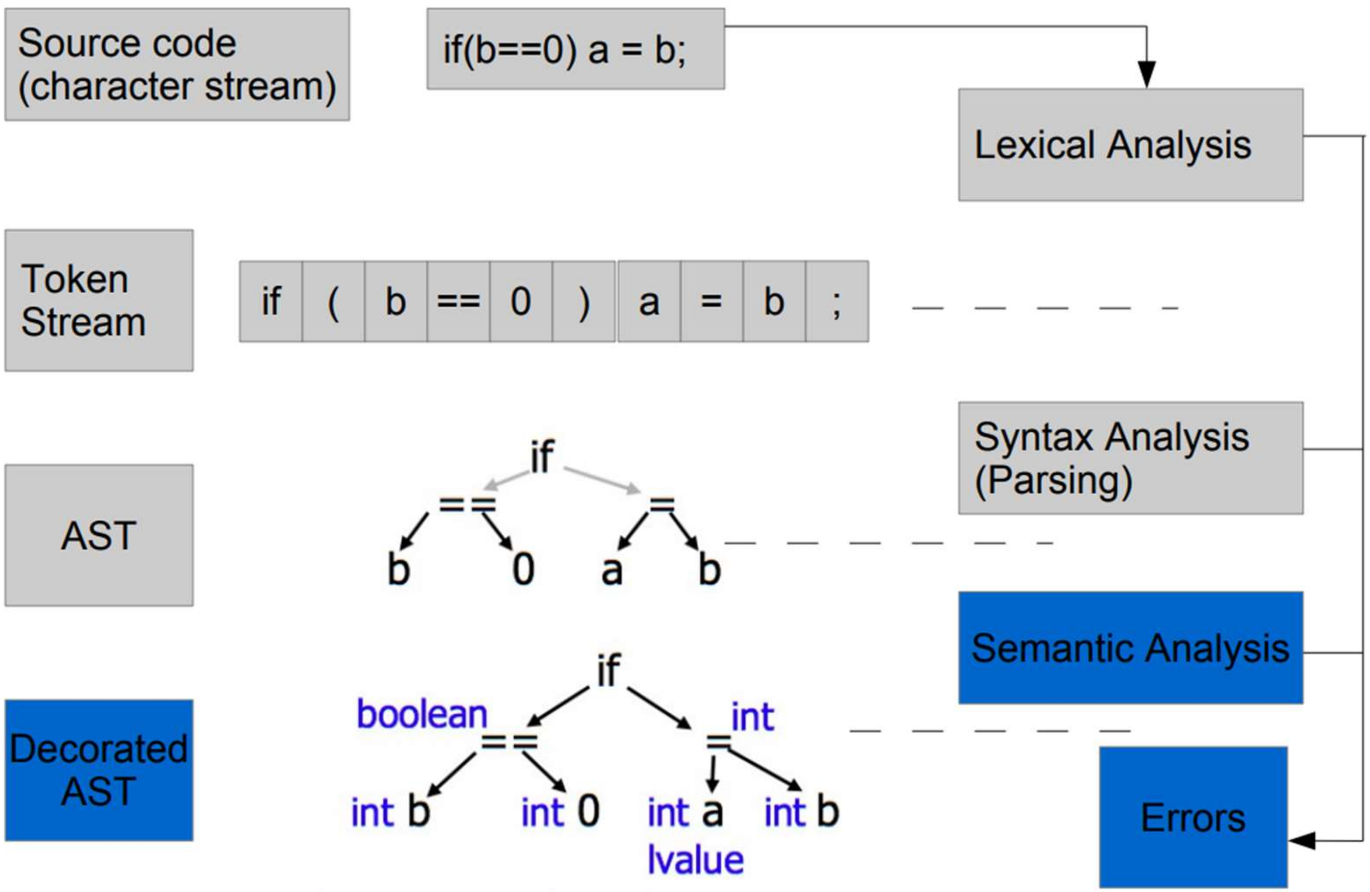
Semantic Analysis- Names, Bindings, and Scopes

Professor: Suman Saha

Where we are?



Source code
(character stream)
if(b==0) a = b;
Lexical Analysis
Token
Stream
if (b == 0) a = b ;
Syntax Analysis
(Parsing)
AST
if
==
=
b 0 a b
Semantic Analysis
Decorated
AST
if
boolean
==
int
=
int b
int 0
int a
lvalue
int b
Errors

Non-Context-Free Syntax

- Program that are correct with respect to the language's lexical and context-free syntactic rules may still contain other syntactic errors
- Lexical analysis and context-free syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc
- Non-context-free syntactic analysis is known as semantic analysis

Incorrect Programs

- Example 1: lexical analysis does not distinguish between different variable or function identifiers (it returns the same token for all identifiers)

```
int a;          int a;
a = 1;          b = 1;
```

- Example 2 : syntax analysis does not correlate the declarations with the uses of variables in the program:

```
int a;
a = 1;          b = 1;
```

Incorrect Programs

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:

```
int a;        int a;
a = 1;        a = 1.0;
```

Goal of Semantic Analysis

- Semantic analysis ensure that the program satisfies a set of additional rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be declared before being used
  - A variable should not be declared multiple times in same scope
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The condition of an if-statement must have type Boolean

Names in a Program

- A mnemonic string in high-level languages
- Identifiers in most languages
- An abstraction of low-level representation, such as memory address and register

Names in a Program

- The same name in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A) A);
    }
}
```

Names in a Program

- The same name in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {
     char A;
     A A(A A) {
          A.A = 'A';
          return A((A) A);
     }
}
```

Names in a Program

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```

Names in a Program

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
    int x = 137;
    {
        string x = "Scope!"
        if (float x = 0)
            double x = x;
    }
    if (x == 137) cout << "Y";
}
```

Binding

- An association between two entities, typically between a name and the object it refers to
  - Name and memory location (for a variable)
  - Name and function
  - Name and type
- *Referencing environment*: A complete set of bindings active at a certain point in a program
- *Scope of a binding*: The region of a program or time interval(s) in the program's execution during which the binding is active.

Binding Times

- Compile time:
  - Map high-level language constructs to machine code
  - Any variable, constant declared either at global scope (outside the main() function), static or as extern variable will occupy memory at compile time.
- Link time:
  - Resolve references between objects in separately compiled module
- Load time:
  - Assign machine addresses to static data
- Runtime:
  - Bind values to variables
  - Allocate dynamic data and assign it to variables
  - Allocate local variables on the stack

Importance of Binding Times

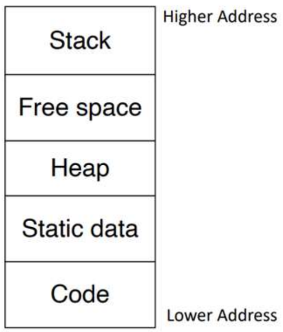
- Early binding (compile time, link time, load time):
  - Faster code
  - Typical in compiled languages
- Late binding (runtime):
  - Greater flexibility
  - Typical in interpreted languages

Object and Binding Lifetime

- Object lifetime: Time between creation and destruction of a dynamically allocated variable in C++ using new and delete.
- Binding lifetime: Period between the creation and destruction of a binding (name-to-object association)
- Two common mistakes:
  - Dangling reference: no object for a binding (E.g., a pointer refers to an object that has already been deleted)
  - Memory leak: No binding for an object (prevents the object from being deallocated)

Storage Allocation Mechanism and Object Lifetime

- An object's lifetime is tied to the mechanism used to manage the space where the object resides.

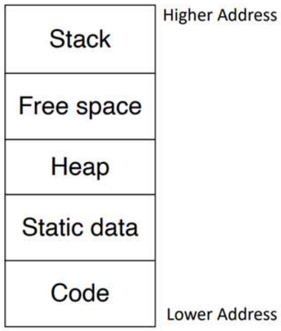


- Static Object
  - Stored at a fixed absolute address
  - Lifetime spans the whole execution of the program
  - Objects allocated in static data area

Object type	Lifetime
Local static	From first function/block execution until program exits
Global, namespace, class static	While program runs

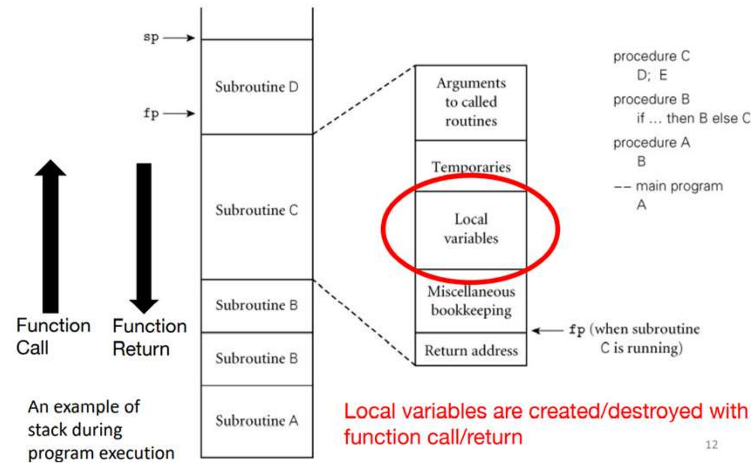
Storage Allocation Mechanism and Object Lifetime

- An object's lifetime is tied to the mechanism used to manage the space where the object resides.



- Object on Stack
  - Allocation on the stack in connection with a subroutine call (bound to local variable)
  - Lifetime spans period between invocation and return of the subroutine.

Object type	Lifetime
Local	While function or block is active
Temporary	Same as a local variable



Storage Allocation Mechanism and Object Lifetime

- An object's lifetime is tied to the mechanism used to manage the space where the object resides.

- Object on Stack
  - Allocation on the stack in connection with a subroutine call (bound to local variable)
  - Lifetime spans period between invocation and return of the subroutine.

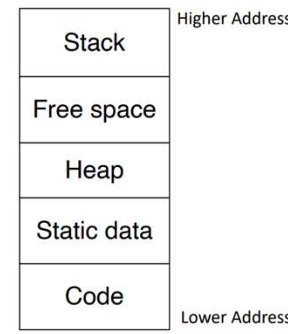
Object type	Lifetime
Local	While function or block is active
Temporary	During expression evaluation

```
int q(int q1, q2) {
  int qx;
  ...
}

int p(int p1) {
  int px;
  q(p1, px);
}

int main() {
  int x;
  p(x);
  q(x, x);
}
```

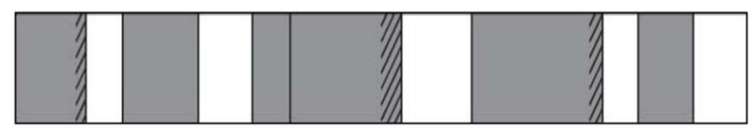
Storage Allocation Mechanism and Object Lifetime

- An object's lifetime is tied to the mechanism used to manage the space where the object resides.

- Object on Heap
  - Allocated on the heap
  - Object created/destroyed at arbitrary times
    - Explicitly by programmer
    - Implicitly by garbage collector



Object type	Lifetime
Heap	Arbitrary



An example of heap during program execution

Storage Allocation Mechanism and Object Lifetime

- An object's lifetime is tied to the mechanism used to manage the space where the object resides.

- Object on Heap
  - Allocated on the heap
  - Object created/destroyed at arbitrary times
    - Explicitly by programmer
    - Implicitly by garbage collector

Object type	Lifetime
Heap	Arbitrary

```
int q() {
   ...
   delete x;
   ...
}

int p() {
   ...
   x = new String[10];
   ...
}

String* x;
int main() {
   p();
   q();
}
```

Declarations and Definitions

- Declarations
  - Introduce a name; give its type (if in a typed language)
- Definitions
  - Fully define an entity
    - Specify value for variables, function body for functions
- Declaration before use
  - Makes it possible to write a one-pass compiler
    - When you call a function, you know its signature
- Definition before use
  - Avoids accessing an undefined variable

Scopes

- *Scope*: A maximal region of the program where the object is accessed (visible) (e.g., a function body)
  - Lexical (static) scoping:
    - Binding based on nesting of blocks
    - Can be determined at compile time
  - Dynamic Scoping:
    - Binding depends on flow of execution at runtime
    - Can only be determined at runtime
    - Scopes can be disjoint

Scopes

- Scope of variables in statement block:

```
{ int a;
  ...
    { int b;
    }
}
```

Scope of variable a

Scope of variable b

- Scope of variables in statement block:

```
int factorial(int n){
  ...
}
```

Scope of formal
parameter n

Scopes

- Scope of labels

```
void f() {
  … goto l;
  l: a=1;
  … goto l;
}
```

Scope of label l

Scope different from Lifetime

- C static local variables

```
void foo(void) {
    static int first=1; /* first is true */
    if (first) {
            first = 0; ...
    } else { ... }
}
```

first's scope is the function and lifetime is the whole program execution time

- A Java static field
  - Scope: the class that contains the field
  - allocated even without any object with that class

Scopes

- Defining scope: The scope in which a name is defined or declared is called its defining scope
- For static scoping, the scope of a name is its defining scope and all nested sub scopes
- The introduction of new variables into scope may hide older variables. How do we keep track of what's visible?

Symbol Tables

- Semantic checks refer to properties of identifiers in the program – their scope or type
- Need an environment to store the information about identifiers = symbol table
- Each entry in the symbol table contains
  - The name of an identifier
  - Additional information: its kind, type, if it is constant, …

Name	Kind	Type	Other
foo	fun	int-> bool	extern
m	par	int	auto
tmp	var	bool	const

Symbol Tables

- How to represent scope information in the symbol table?
- Idea:
  - One symbol table for each scope. It is a dictionary or hash map.
  - A symbol table contains the symbols declared in that lexical scope
  - There is a hierarchy of scope in the program
  - Use a similar hierarchy of symbol table. A symbol table may have a parent

Symbol Tables

```
int x;

void f(int m){
  float x, y;
  ...
{ int i, j; ...;}
{ int x; l: ...;}
}

int g(int n){
  bool t;
  ...;
}
```

Global Symbol Table

x	var	int
f	fun	int-> void
g	fun	int → int

func f Symbol Table

m	par	int
x	var	float
y	var	float

func g Symbol Table

n	par	int
t	var	bool

i	var	int
j	var	int

x	var	int
l	lab	

Identifiers with Same Name

- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes)
- To find the declaration of an identifier that is active at a program point:
  - Start from current scope
  - Go up in the hierarchy until you find an identifier with the same name, or fail

Example

```
int x;

void f(int m){
  float x, y;
  ...
{ int i, j; x=1;}
{ int x; l: x=2;}
}

int g(int n){
  bool t;
  x=3;
}
```

Global Symbol Table

x	**var**	**int**
f	fun	int-> void
g	fun	int → int

func f Symbol Table

m	par	int
x	**var**	**float**
y	var	float

func g Symbol Table

n	par	int
t	var	bool

x = 3

x = 2

x = 1

i	var	int
j	fun	int

x	**var**	**int**
l	lab	

Catching Semantic Errors

```
int x;

void f(int m){
  float x, y;
  ...
{ int i, j; x=1;}
{ int x; l: i=2;}
}

int g(int n){
  bool t;
  x=3;
}
```

Global Symbol Table

Error

x	**var**	**int**
f	fun	int-> void
g	fun	int → int

func f Symbol Table

m	par	int
x	**var**	**float**
y	var	float

func g Symbol Table

n	par	int
t	var	bool

x = 3

x = 1

i	var	int
j	fun	int

i = 2

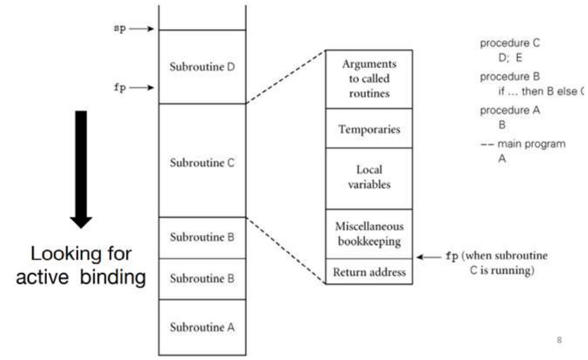
x	**var**	**int**
l	lab	

Top Hat

Dynamic Scope

- In lexical scoping, you search in the local function (the function which is running now), then you search in the function (or scope) in which that function was defined, then you search in the function (scope) in which that function was defined, and so forth. Most languages: Ada, C, Pascal

- However, in dynamic scoping, by contrast, you search in the local function first, then you search in the function that called the local function, then you search in the function that called that function, and so on, up the call stack. Used by some early dialects of Lisp



Static Scope VS Dynamic Scope

Static Scope

```
const int b = 5;
int foo(){
  int a = b + 5;
  return a;
}
int bar(){
  int b = 2;
  return foo();
}
int main(){
  foo(); // returns 10
  bar(); // returns 10
  return 0;
}
```

Dynamic Scope

```
const int b = 5;
int foo(){
  int a = b + 5;
  return a;
}
int bar(){
  int b = 2;
  return foo();
}
int main(){
  foo();// returns 10
  bar();// returns 7
  return 0;
}
```

Dynamic Scoping

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```

(global)

Name	Kind	Type
n	Id	int
first	fun	void ->void
second	fun	Void->void

(first)

Name	Kind	Type

Set global n to be 1

Dynamic Scoping

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```

(global)

Name	Kind	Type
n	Id	int
first	fun	void ->void
second	fun	Void->void

(second)

Name	Kind	Type
n	Id	int

(first)

Name	Kind	Type

Set n in “second” to be 1

Symbol tables changes at run time!
Always use most recent, active binding

Example

```
int x = 10;
 int f(){
   return x;
}
int g(){
   int x = 20;
   return f();
}
int main(){
  printf("%d", g());
  return 0;
}
```

1) What is output if code uses static scoping?

2) What is the output if code uses dynamic scoping?

Static VS Dynamic Scoping

- Similarity
  - One symbol table per scope
  - Names are resolved from bottom to top in symbol table tree
- Difference
  - Static: the symbol table tree is stable
  - Dynamic: the symbol table tree changes during execution

Subroutine as a Parameter

- If a subroutine is passed as a parameter, how do we resolve names?

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

which x?

Subroutine as a Parameter

- If a subroutine is passed as a parameter, how do we resolve names?

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

which x?

What's the parent of H's symbol table?

Subroutine as a Parameter

- What's the parent of H's symbol table?

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

Shallow Binding: when the subroutine is called

Deep Binding: when reference is created

Shallow Binding

- Use the environment at call time

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

Established at call time

(F)

Name	Kind	Type
x	para	int
G	fun	fun ->void
H	fun	void->void

(G)

Name	Kind	Type
fx	para	fun
x	Id	int

(H)

Name	Kind	Type

Deep Binding

PennState

- Use the environment when ref. is created

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

(F)

Name	Kind	Type
x	para	int
G	fun	fun ->void
H	fun	void->void

Established when ref. is created

(H)

Name	Kind	Type

Implementation of Deep Binding: Function Closures

- A closure is a pair of a function and a referencing environment

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

Pass in a function closure

A function closure contains:

- A pointer to the function
- The current symbol table (or all symbol tables, depending on implementation)

Implementation of Deep Binding: Function Closures

- Closures are necessary to implement deep binding in languages that allow functions to be passed around as function arguments and return values:
  - When the function is invoked, the referencing environment it refers to may no longer exist and thus needs to be preserved explicitly in the closure.

Example

```
int x=5;
void f() {
        x = x+50;
}
 void g(h) {
        int x=10;
        h();
        print(x);
}
void main() {
        g(f());
}
```

1) What is output if code uses deep binding?

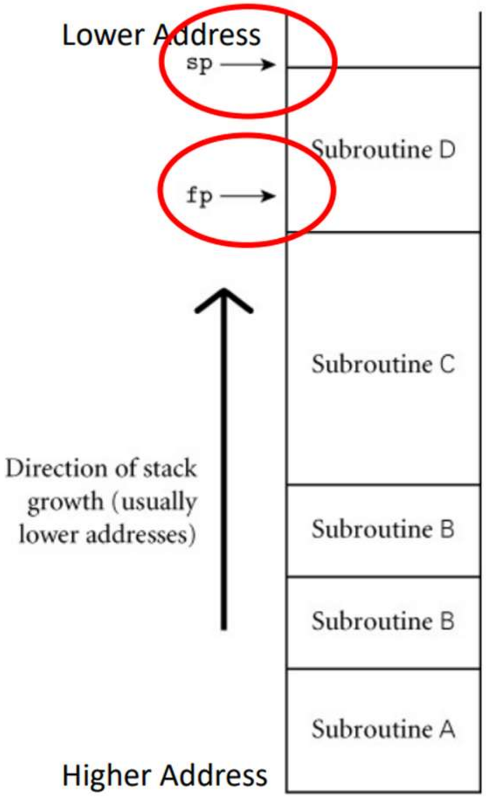
2) What is the output if code uses shallow binding?

Deep Binding

- Dynamic scoping
  - Both shallow and deep binding are implemented
  - Deep binding has a higher cost at run time
- Static scoping
  - Shallow binding has never been implemented

Top Hat

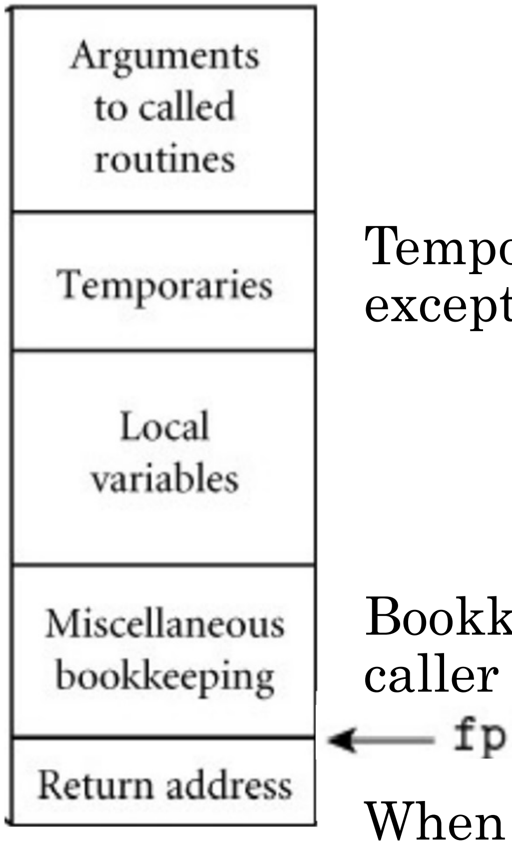
Scope and Stack




```
procedure C
    D;  E
procedure B
    if ... then B else C
procedure A
    B
-- main program
    A
```

- Each instance of a subroutine has a frame (activation record) at run time
  - Compiler generates code that setup frame, call routine, and destroy frame
- Frame pointer (fp): currently active frame
- Stack pointer (sp): free space on stack

Typical Frame Layout



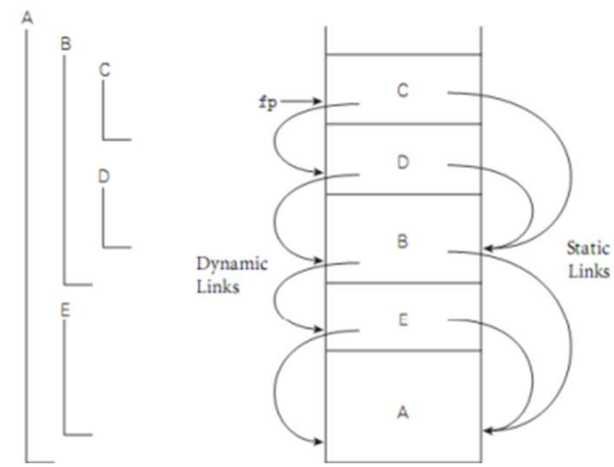
Temporaries.: register spill area, language-specific exception-handling context, and so on (not covered)

Bookkeeping info.: a reference to the stack frame of the caller (also called the dynamic link) and so on

When the frame is active

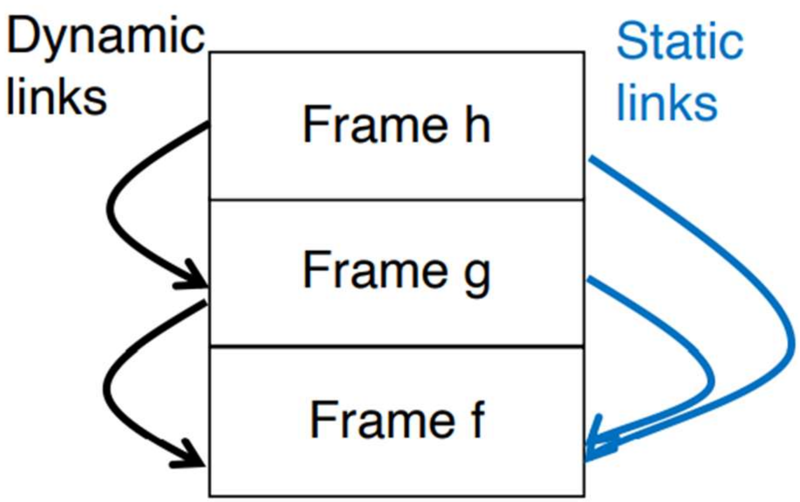
Nested Scopes

- Languages with nested functions:
  - How do we access local variables from other frames?
- Static Scoping:
  - Static link: a pointer to the frame of enclosing function
- Dynamic Scoping:
  - Dynamic link: pointer to the previous frame in the current execution



Static Link and Dynamic Link

```
void f (int i) {
   int a;
   void h (int j) {
     a = j;
   }
   void g (int k) {
     h(k);
   }
   g(i+2);
}
```



Where is variable a in h?

PennState

Static Scoping

```
void f (int i) {
    int a;
    void h (int j) {
      a = j;
    }
    void g (int k) {
      h(k);
    }
    g(i+2);
```

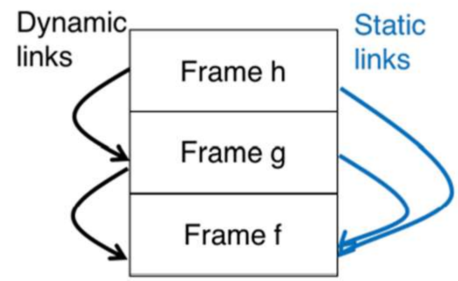
Name	Kind
a	id
i	para
h	fun
g	fun

Symbol tables

Name	Kind
k	para

Name	Kind
j	para

Where is a? From symbol table of h, go up one hop



Where is a? From frame of h, go up one hop following static links

Where is variable a in h?

Dynamic Scoping

Name	Kind
a	id
i	para
h	fun
g	fun

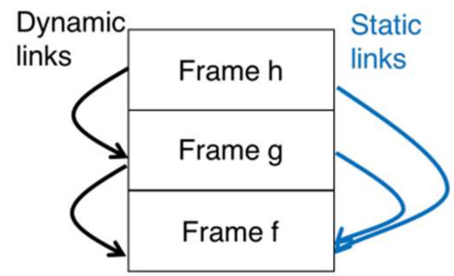
Name	Kind
k	para

Name	Kind
j	para

Where is a? From symbol table of h, go up two hops

```
void f (int i) {
    int a;
    void h (int j) {
      a = j;
    }
    void g (int k) {
      h(k);
    }
    g(i+2);
```

Symbol tables



Where is a? From frame of h, go up two hops following dynamic link

Static & Dynamin Link (Example)

```
program MAIN_2;
    procedure BIGSUB;
        procedure SUB1;
        begin … end SUB1;
        procedure SUB2;
            procedure SUB3;
            begin … end SUB3;
        begin … end SUB2;
    begin … end BIGSUB;
begin … end MAIN_2;
```

Static & Dynamin Link (Example)

```
program MAIN_2;
        procedure BIGSUB;
                procedure SUB1;
                begin … end SUB1;
                procedure SUB2;
                        procedure SUB3;
                        begin … end SUB3;
                begin … end SUB2;
        begin … end BIGSUB;
begin … end MAIN_2;
```

Static & Dynamin Link (Example)

```
program MAIN_2;
    procedure BIGSUB;
        procedure SUB1;
        begin … end SUB1;
        procedure SUB2;
            procedure SUB3;
            begin … end SUB3;
        begin … end SUB2;
    begin … end BIGSUB;
begin … end MAIN_2;
```

Static & Dynamin Link (Example)

```
program MAIN_2;
    procedure BIGSUB;
        procedure SUB1;
        begin … end SUB1;
        procedure SUB2;
            procedure SUB3;
            begin … end SUB3;
        begin … end SUB2;
    begin … end BIGSUB;
begin … end MAIN_2;
```

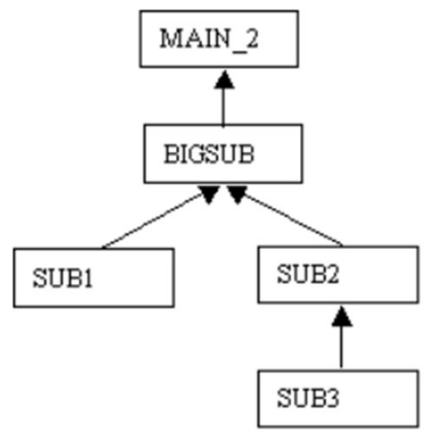
Static & Dynamin Link (Example)

```
program MAIN_2;
    procedure BIGSUB;
        procedure SUB1;
        begin … end SUB1;
        procedure SUB2;
            procedure SUB3;
            begin … end SUB3;
        begin … end SUB2;
    begin … end BIGSUB;
begin … end MAIN_2;
```

Static & Dynamin Link (Example)

```
program MAIN_2;
        procedure BIGSUB;
                procedure SUB1;
                begin … end SUB1;
                procedure SUB2;
                        procedure SUB3;
                        begin … end SUB3;
                begin … end SUB2;
        begin … end BIGSUB;
begin … end MAIN_2;
```

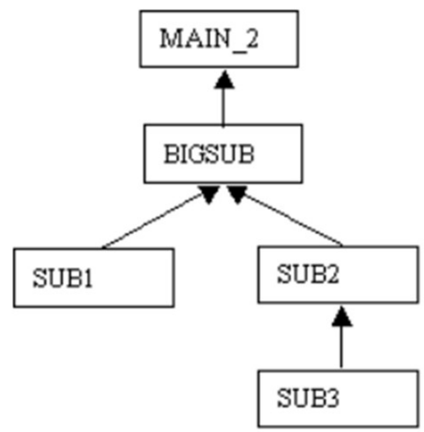
Static Structure



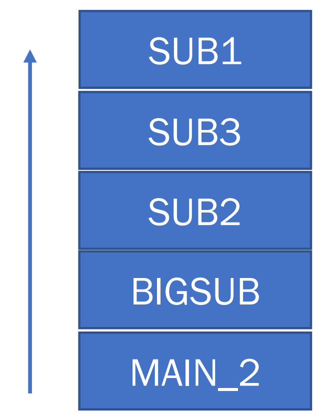
Static & Dynamin Link (Example)

```
program MAIN_2;
        procedure BIGSUB;
                procedure SUB1;
                begin … end SUB1;
                procedure SUB2;
                        procedure SUB3;
                        begin … end SUB3;
                begin … end SUB2;
        begin … end BIGSUB;
begin … end MAIN_2;
```

Static Structure



MAIN_2 calls BIGSUB calls SUB2 calls SUB3 calls SUB1

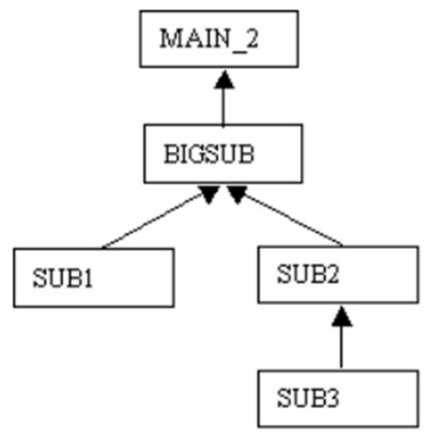


Stack

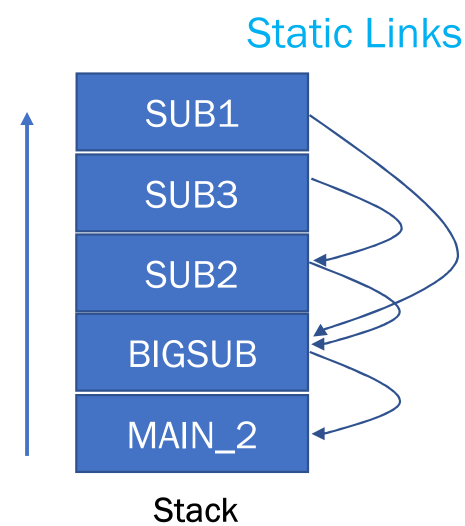
Static & Dynamin Link (Example)

```
program MAIN_2;
    procedure BIGSUB;
        procedure SUB1;
        begin … end SUB1;
        procedure SUB2;
            procedure SUB3;
            begin … end SUB3;
        begin … end SUB2;
    begin … end BIGSUB;
begin … end MAIN_2;
```

Static Structure



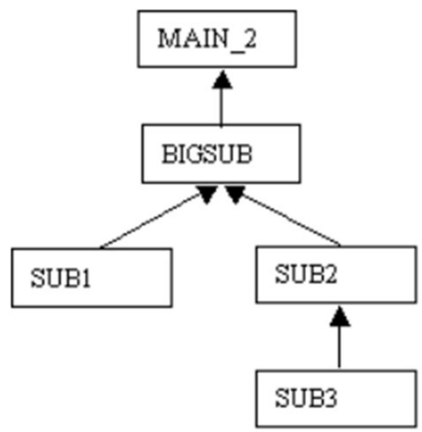
MAIN_2 calls BIGSUB calls SUB2 calls SUB3 calls SUB1



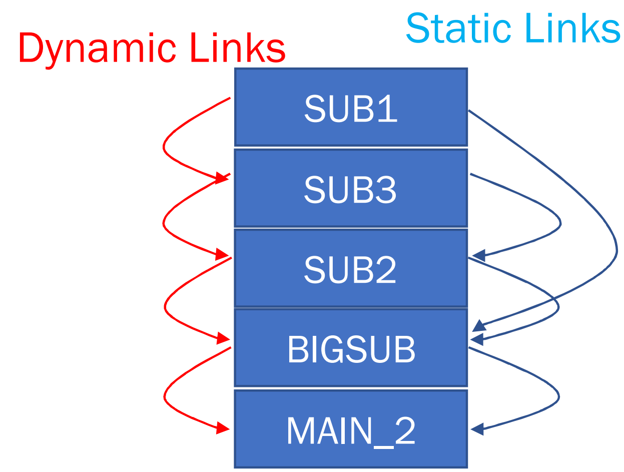
Static & Dynamin Link (Example)

```
program MAIN_2;
        procedure BIGSUB;
                procedure SUB1;
                begin … end SUB1;
                procedure SUB2;
                        procedure SUB3;
                        begin … end SUB3;
                begin … end SUB2;
        begin … end BIGSUB;
begin … end MAIN_2;
```

Static Structure



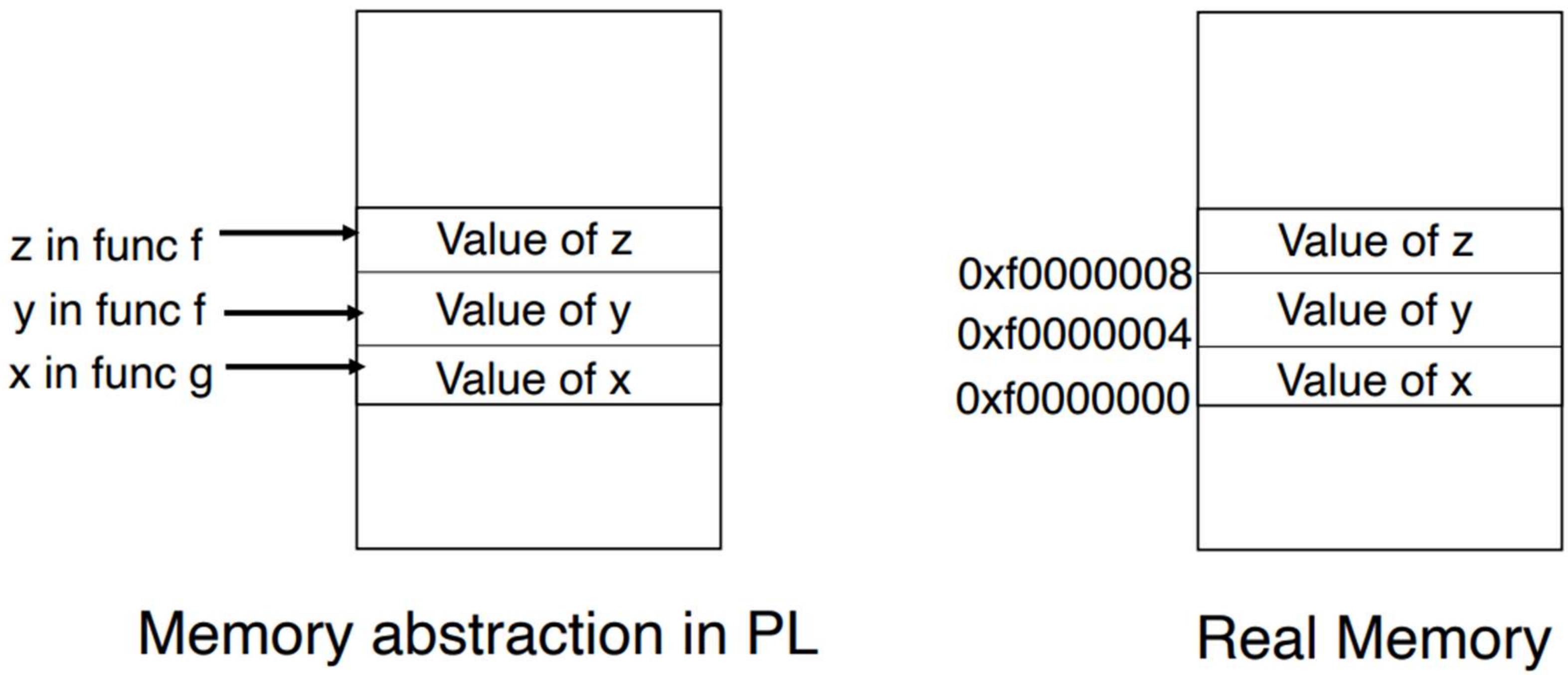
MAIN_2 calls BIGSUB calls SUB2 calls SUB3 calls SUB1



Stack

Variable and Memory Address

- How does the compiled code locate a variable used in a function at run time?

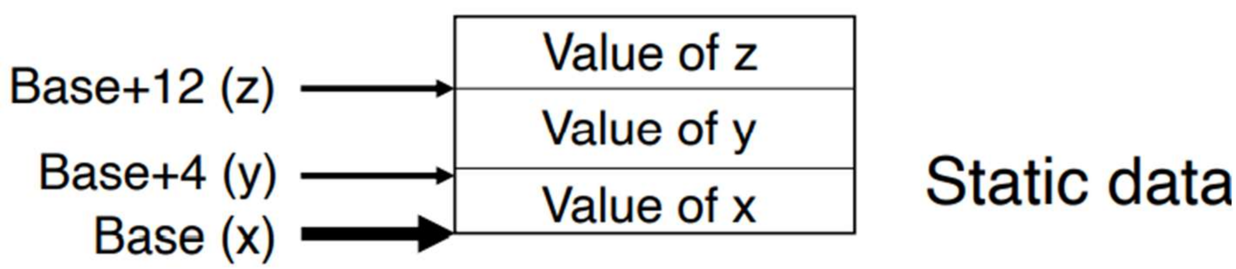


Establishing Addressability

- How does the compiled code locate a variable used in a function at run time?
  - What is the start address?

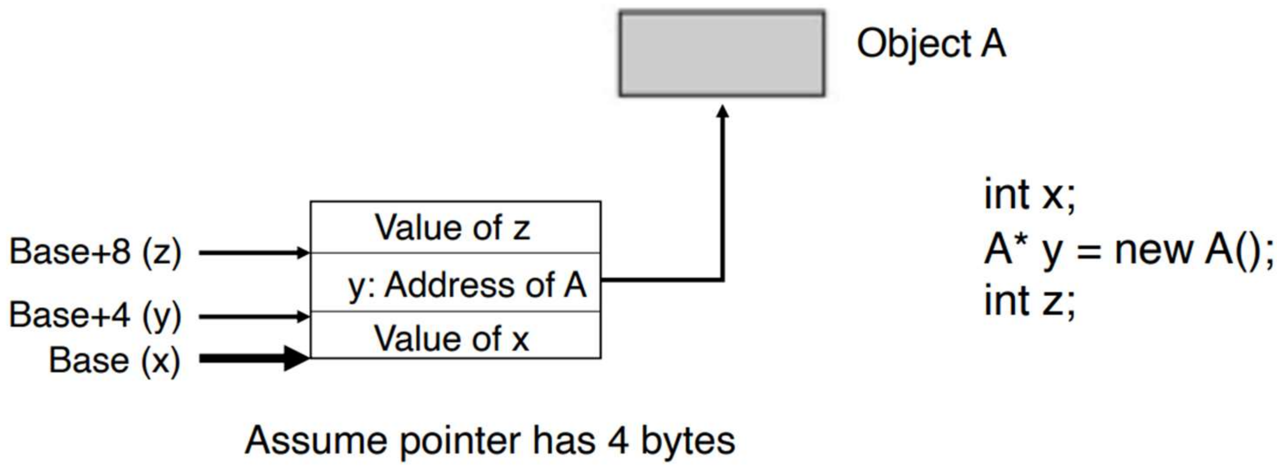
    Start address = base address (of a scope) + offset (in the scope)

- Base Address: fixed throughout the run
- Offset: known at compile time (length of each variable is determined by its type)

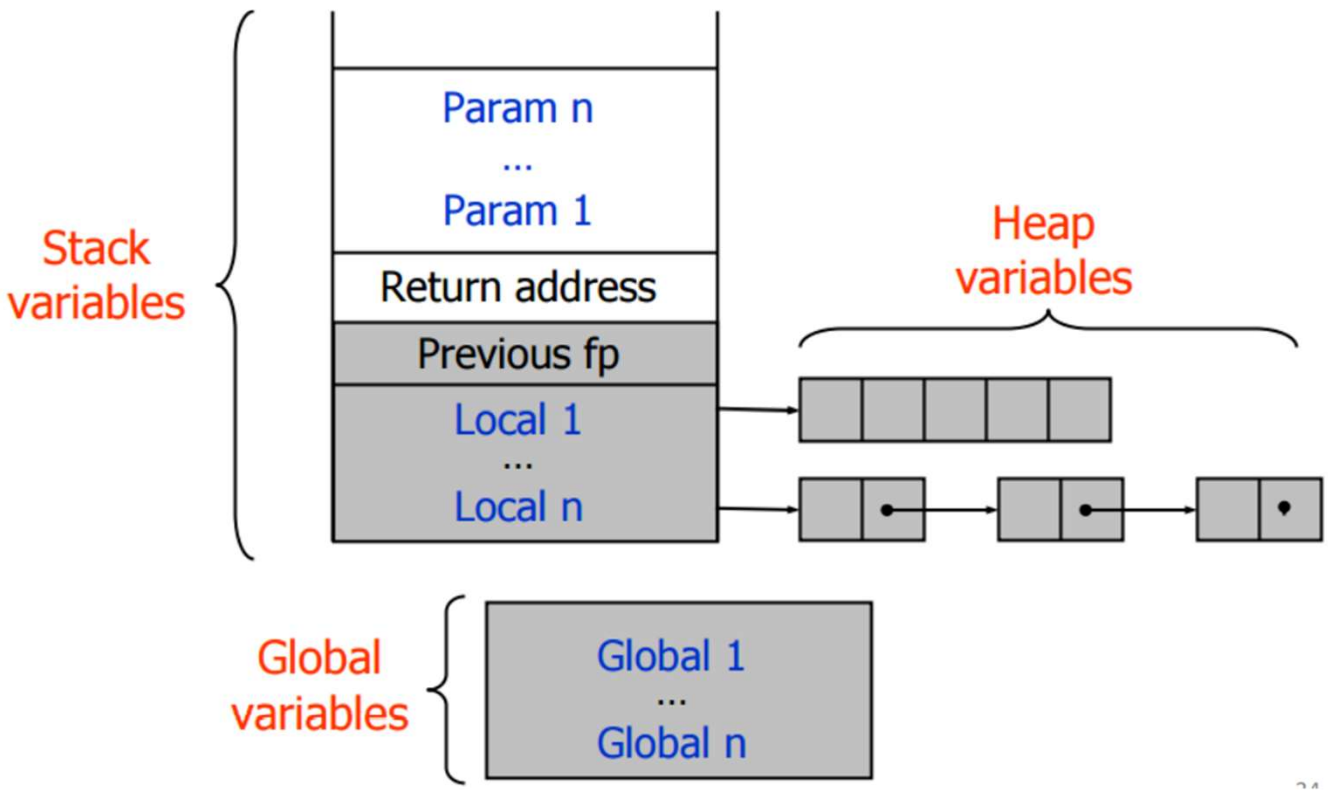


Heap-Based Variable

- Like a normal variable, except that the memory cell stores the address of heap space



Big Picture: Data Memory Layout



Reading and Exercises

Reading

- Chapter: 3.1, 3.2, 3.3, and 2.3.3 (Michael Scott Book)

Exercises

- Exercises: 3.1 - 3.7, 3.11-3.14, 3.18, 3.19 (Michael Scott Book)