Principles of Programming Languages Lecture 21

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig; PL Concepts and Constructs by Sethi; Concepts of PL by Sebesta; ML for the Working Prog. By Paulson; Prog. in Prolog by Clocksin and Mellish; PL Pragmatics by Scott

Components of an Imperative Language

- Data types
- Variables, operators & Expressions
- Iteration construct
- Branching construct

→ Subprogram construct

- Container construct

Subprograms: parameter passing

- **Implementation models:**
  - Pass by value
  - Pass by result
  - Pass by value-result
  - Pass by reference

- **Modern languages support pass-by-value and pass-by-reference**

- **Concurrency problems with other models.**

Subprograms: parameter passing

- Pass by Value-Result

```
public class Counters extends Thread {
   public     int m = 5;
   public     int n = 3;

   public void increment(int k, int j) {
     k = k + 1;
     j = j + 2;
    }

    public void run( ) {
      increment( m , n );
    }

}
```

```
public class User extends Thread {
   public Counters _counters;

   public User( Counters counters ) {
    _counters = counters;
   }

   public void run( ) {
    int x = _counters.m + _counters.n;
   }
}
```

```
public class App {

   public static void main( String strarrArgs[] ) {
      Counters counters = new Counters( );
      User user = new User( counters );

      counters.start( );
      user.start( );
   }
}
```

Result is either 5/3 or 6/5 or 5/5 !!

Subprograms: parameter passing

- **Java**
  - Primitives: pass by value
  - Mutable Objects: pass by reference
    - Vector, List, Map, Hashtable, StringBuffer
  - Immutable objects: pass by value
    - String!

```
void dothis(String str){
    String strInput = str;
}
....
 lTime1 = System.currentTimeMillis();
for(int nIndex=0;nIndex < 1000000;nIndex++){
   String  strVal = "alkasdfasdfasdfdjflkasdjfl";
   dothis(strVal);
}
 lTime1 = System.currentTimeMillis();
```

- **Most modern languages are following Java's model.**

- **Always check if the new language you are learning is passing objects by value or reference (and can you chose between them as in C/C++)**

Scope: introduction

- **The textual region of the program in which a specific set of variable bindings are active is called the *scope*.**

- **Variables and Scope:**
  - A variable is said to be *visible* in a statement if it can be referenced in that statement without a type error
  - If a variable outlives its binding it's garbage
  - If a binding outlives a variable it's a dangling reference

- **Elaboration:**
  - It is the process of opening a new scope and creating appropriate bindings.
  - Done upon entering a subroutine

Subprograms: scope & blocks

- **A block is a section of code in which local variables are allocated and de-allocated at the start/end of the block.**

- **Examples:**

```
fun cube x = x*x*x;

fun  f (a::b::_) = a+b

|     f  [] = 0;
```

```
declare
  LCL : FLOAT;
  begin
  ...
  end
```

```
while (i < 0) {
  int c = i*i*i;
  p += c;
  q += c;
  i -= step;
}
```

```
(let  ((a 1)
       (b foo)
       (c))
       (setq a (* a a))
      (bar a b c))
```

Subprograms: scope, blocks & nesting

- **What happens if a block contains another block, and both have definitions of the same name?**

- **Example:**

```
let
      val n = 1
 in
      let
          val n = 2
      in
          n
      end
end
```

Subprograms: static scope

- **Defines scope in terms of the lexical structure of the program**
  - A name begins life where it is declared and ends at the end of its block.
  - A scope of a variable is known before execution
    - Can be fully determined and bindings made at compile time
  - E.g. C, C++, Pascal, Java, Fortran, Basic, Python, Perl,…
- **E.g.**

```
int main(){ // main scope
    int k;
     …….
}
```

```
void testfunc(){
    int a; // a added to testfunc scope
    for ( int b=1; b<10; b++ ) { // b in scope
        int c; // c added/enters loop scope
        …
     } // b,c leave/deleted-from scope
} // a leaves/delete-from scope
```

Subprograms: static scope

- **A name begins life where it is declared and ends at the end of its block.**
- **E.g.**

```
{
  int x = 0;
  foo( x );
  {
      int x = 1;
      bar( x );
  }
 bar( x );
}
```

Classic Block Scope Rule: When using static scope, the scope of a definition is the block containing that definition, from the point of definition to the end of the block, minus the scopes of any redefinitions of the same name in interior blocks

Hence, innermost scope overrides declarations from outer scopes.

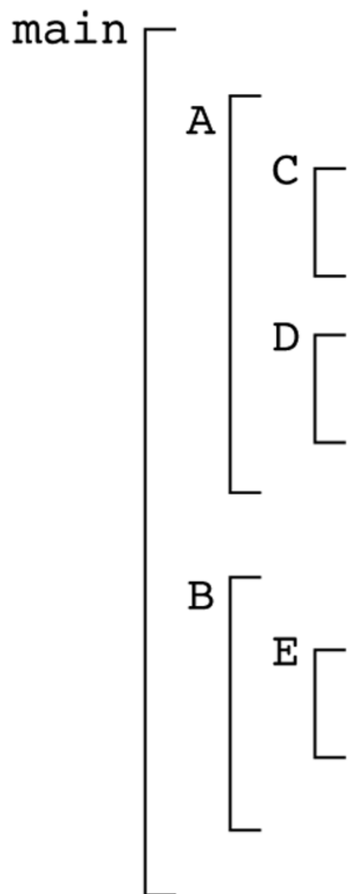
Subprograms: static scope

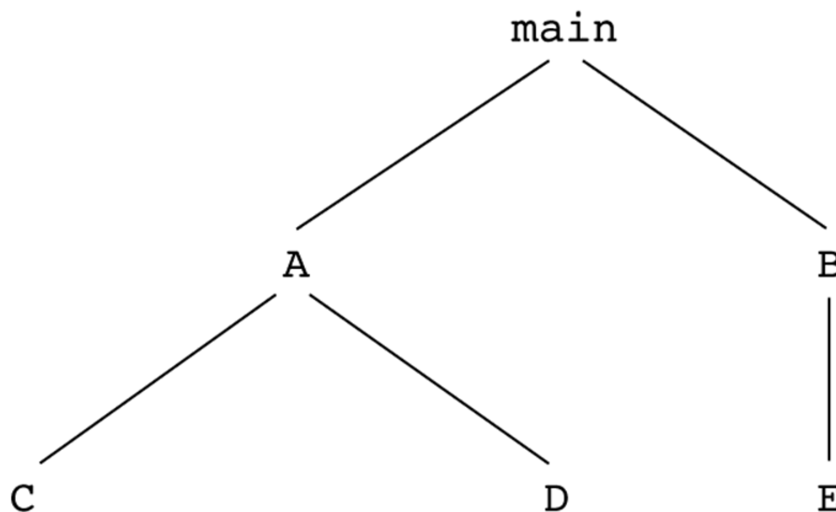
- **A name begins life where it is declared and ends at the end of its block.**
- **E.g.**

```
program test;
var a : integer;

    procedure proc1;
    var b : integer;
    begin
    end;              <----------  in scope: b (from proc1), a (from test)

    procedure proc2;
    var a, c : integer;
    begin             <----------  in scope: a, c (from proc2)
        proc1;
    end;
begin                 <----------  in scope: a (from test)
  proc2;
end.
```

Subprograms: static scope & lexical tree

- At compile time; The compiler/interpreter constructs a lexical tree from the source code.
- E.g. assuming A and B are classes while C,D and E are methods



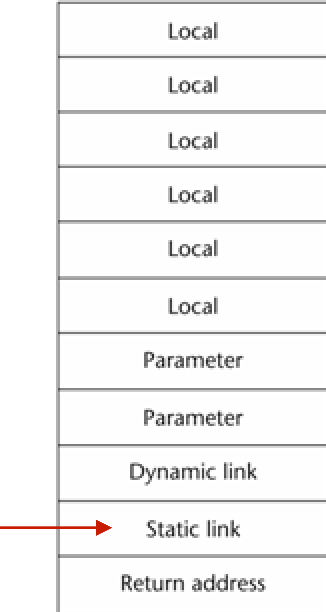


Subprograms: static scope implementation

- **At run time; The compiler/interpreter follow the static link in the activation record**

- **E.g.**

```
void testfunc(){     // nested scope
        int a; // a enters scope;
        for ( int b=1; b<10; b++ ) { // b in scope
             int c; // c enters scope
             if( c < 10 ){ // d enters scope
                 int  d = a + c;
              }   // d leaves scope
             …
         } // b,c leave scope
    } // a leaves scope
```



Subprograms: dynamic scope

- **A dynamically-scoped identifier refers to the closest enclosing definition in that specific activation**
  - Define scope based on the current state of program execution
  - Scope cannot always be determined by reading the program as we do with static scope
  - E.g. some dialects of Lisp/APL and SNOBOL

Subprograms: dynamic scope

- **A dynamically-scoped identifier refers to the closest enclosing definition in that specific activation**
- **E.g.**

```
program test;
var a : integer;

        procedure proc1;
        var b : integer;
        begin        <------------------  in scope: b (from proc1) a, c (from proc2)
        end;

        procedure proc2;
        var a, c : integer;
        begin        <------------------  in scope: a, c (from proc2)
                proc1;
        end;
begin        <---------------------------  in scope: a (from test)
  proc2;
end.
```

Dynamic Scope: introduction

- **Algorithm:**
  - Look first in the block in which the reference occurs, if that fails, look in the calling subprogram, continue until successful or you reach the top level block without finding a declaration.

- **Example:**

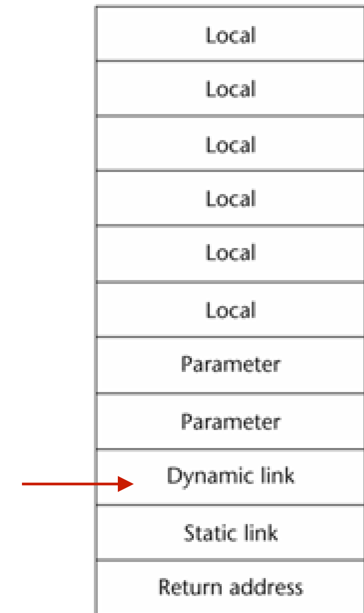
```
int x;

void sub1 (void) {
   printf("%d\n", x);
}

void sub2 (void) {
   int x = 8;
   sub1();
}

void main (void) {
   x = 10;
   sub1();
   sub2();
}
```

What is the output if scope was dynamic?

Subprograms: dynamic scope implementation

- The compiler/interpreter follow the dynamic link in the activation record
- E.g.

```
int x;

void sub1 (void) {
   printf("%d\n", x);
}

void sub2 (void) {
   int x = 8;
   sub1();
}

void main (void) {
   x = 10;
   sub1();
   sub2();
}
```



Dynamic vs. Static Scope

```
1:   a : integer          -- global declaration

2:   procedure first
3:       a := 1

4:   procedure second
5:       a : integer      -- local declaration
6:       first ()

7:   a := 2
8:   if read_integer () > 0
9:       second ()
10:  else
11:      first ()
12:  write_integer (a)
```

- **What is the value of a using static scoping/dynamic scoping?**
  - Static: for +ve input (e.g. 5):- output is 1 , for -ve input:- output is also 1
  - Dynamic: for +ve input (e.g. 5):- output is 2 , for -ve input:- output is also 1

Dynamic Vs. Static Scope

```
int w = 1;
int x = 2;
String y = "achoo";
String z = "ohm";

function p () {
    int w = 13;
    int x = -1;
    String z = "hello";
    print w, x, y, z;
    q();
}
function q () {
    int x = 99;
    String y = "goodbye";
    print w, x, y, z;
    r();
    s();
}
```

```
function r () {
    int x = 101;
    String z = "googoo";
    print w, x, y, z;
}
function s () {
    int x = -555;
    print w, x, y, z;
    t();
}
function t () {
    print w, x, y, z;
}
main() {
    p();
}
```

Dynamic Scope

13	-1	achoo	hello
13	99	goodbye	hello
13	101	goodbye	googoo
13	-555	goodbye	hello
13	-555	goodbye	hello

Static Scope

13	-1	achoo	hello
1	99	goodbye	ohm
1	101	achoo	googoo
1	-555	achoo	ohm
1	2	achoo	ohm

Subprograms: static vs. dynamic scope

- **Dynamic scope makes it easier to access variables with lifetime, but it is difficult to understand the semantics of code outside the context of execution.**
  - No need for implicit parameter passing

- **Static scope is more restrictive – therefore easier to read – but may force the use of more subprogram parameters or global identifiers to enable visibility when required.**

Subprograms: static vs. dynamic scope

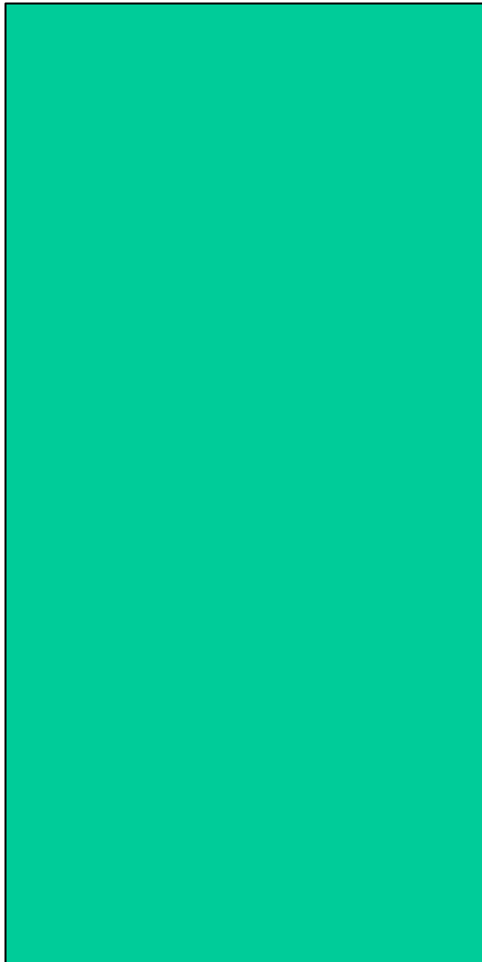
- **Static Scope:**
  - Defines scope in terms of the lexical structure of the program
  - A scope of a variable is ***known before execution***
  - Static scopes can be fully determined and bindings made at compile time
  - When writing a program one typically chooses the most recent, active binding made at compile time
  - Most compiled languages, C and Pascal included, employ static scope rules
- **Dynamic scope:**
  - Define scope based on the current state/flow of program execution
  - A scope of a variable is ***known at run time***
  - In this case the scope cannot always be determined by examining the program because it is dependent on (dynamic) calling sequences
  - E.g.:
    - To resolve a reference, the most recent, active binding made at run time is used
  - Dynamic scope rules are usually encountered in interpreted languages

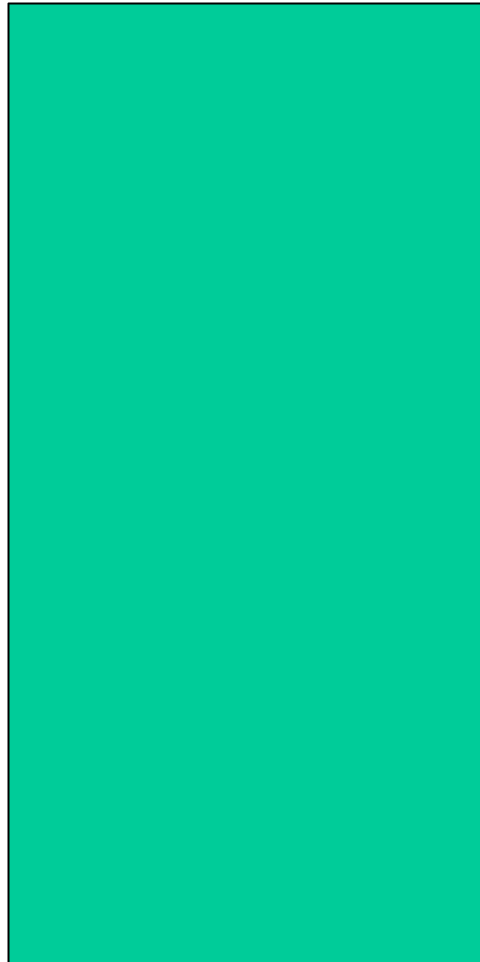
Dynamic Scope: pros & cons

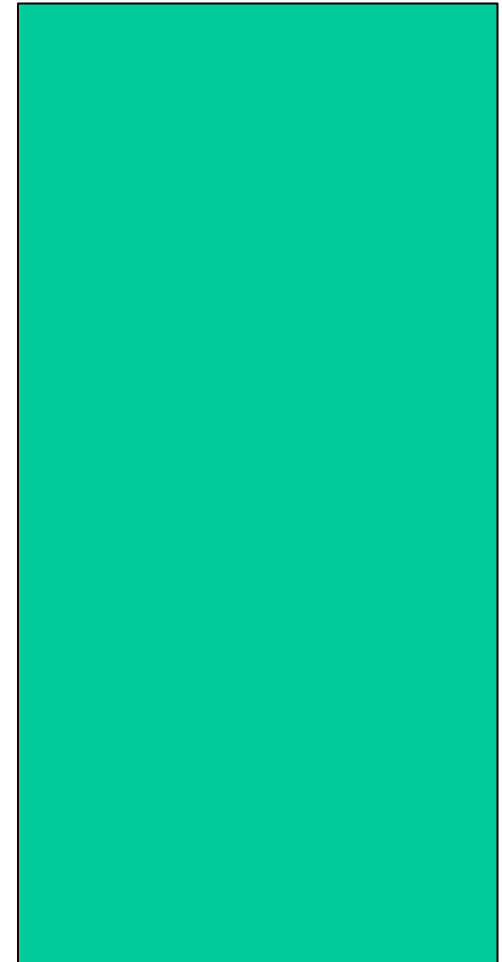
- **Prior to the advent of modern object-oriented programming techniques, dynamic scope facilitated the customization of subroutines**
  - Perform implicit parameter passing
  - E.g. early versions of Lisp/Scheme.

- **Problems with dynamic scope:**
  - It is hard to understand code with dynamic scope
  - You can change what the program does just be renaming variables!
  - Any subprogram you call, no matter where it is, can access your local variables.
  - It is slower to execute
- **This is no longer widely considered good programming practice – better solutions exist**
  - Use optional/default parameters (e.g., in C++)
  - Use function/method overloading
  - Use static variables

Dynamic Scope Merits


GUI layer


Logic layer


Data layer

Dynamic Scope Merits

```
void showCCard(){
   getCCard()
}

void showBills(){
  getBills()
}

void deposit(){
   doDeposit()
}
```

GUI layer

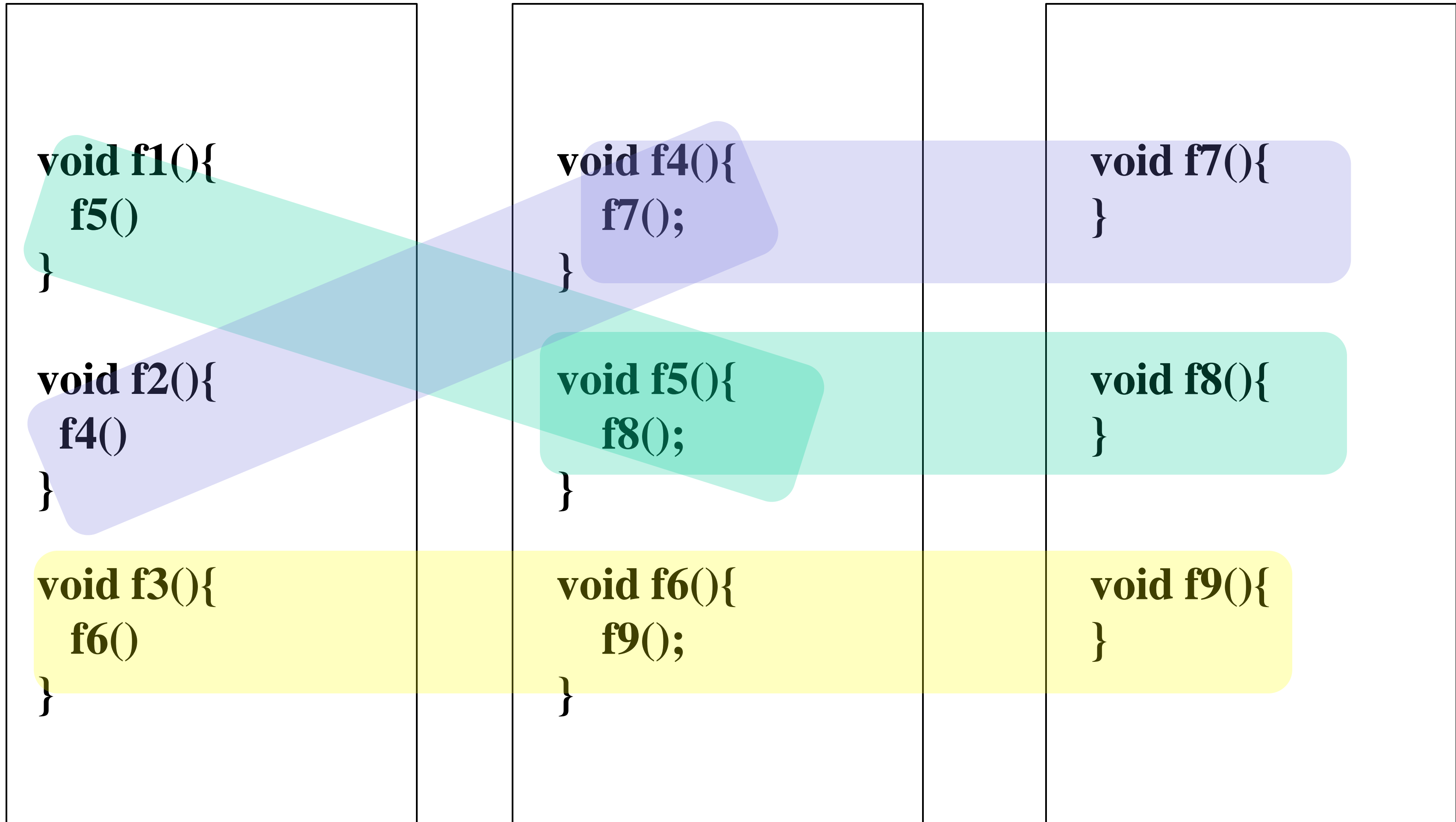
```
void getBills(){
    loadBills();
}

void getCCard(){
   loadCards();

}

void doDeposit(){
    loadSaving();
    incrementSaving();
}
```

Logic layer

```
void loadBills(){
}

void loadCards(){
}

void loadSaving(){
}

void updateSaving(){
}
```

Data layer

Dynamic Scope Merits



Dynamic Scope Problems

```
void f1(){
  int x  = 10;
  f5()
}
void f2(){
  f4()
}

void f3(){
   f6()
}
```

GUI layer

```
void f4(){
    f7();
}

void f5(){
 y = x / 2;
f8();
}
void f6(){
    f9();
}
```

Logic layer

```
void f7(){
}

void f8(){
   int m = y + x
}

void f9(){
}
```

Data layer

Decreases readability of Programs

Dynamic Scope Problems

```
void f1(){
  int x  = 10;
  f5()
}
void f2(){
  f4()
}

void f3(){
   f6()
}
```

GUI layer

```
void f4(){
    f7();
}

void f5(){
 y = x / 2;   Ok..
f8();
}
void f6(){
    f9();
}
```

Logic layer

```
void f7(){
}

void f8(){
}

void f9(){
}
```

Data layer

Dynamic Scope Problems

GUI layer

```
void f1(){
  f5()
  int x  = 10;
}
void f2(){
  f4()
}

void f3(){
   f6()
}
```

Logic layer

```
void f4(){
    f7();
}

void f5(){
 y = x / 2;   Error!
f8();
}
void f6(){
    f9();
}
```

Data layer

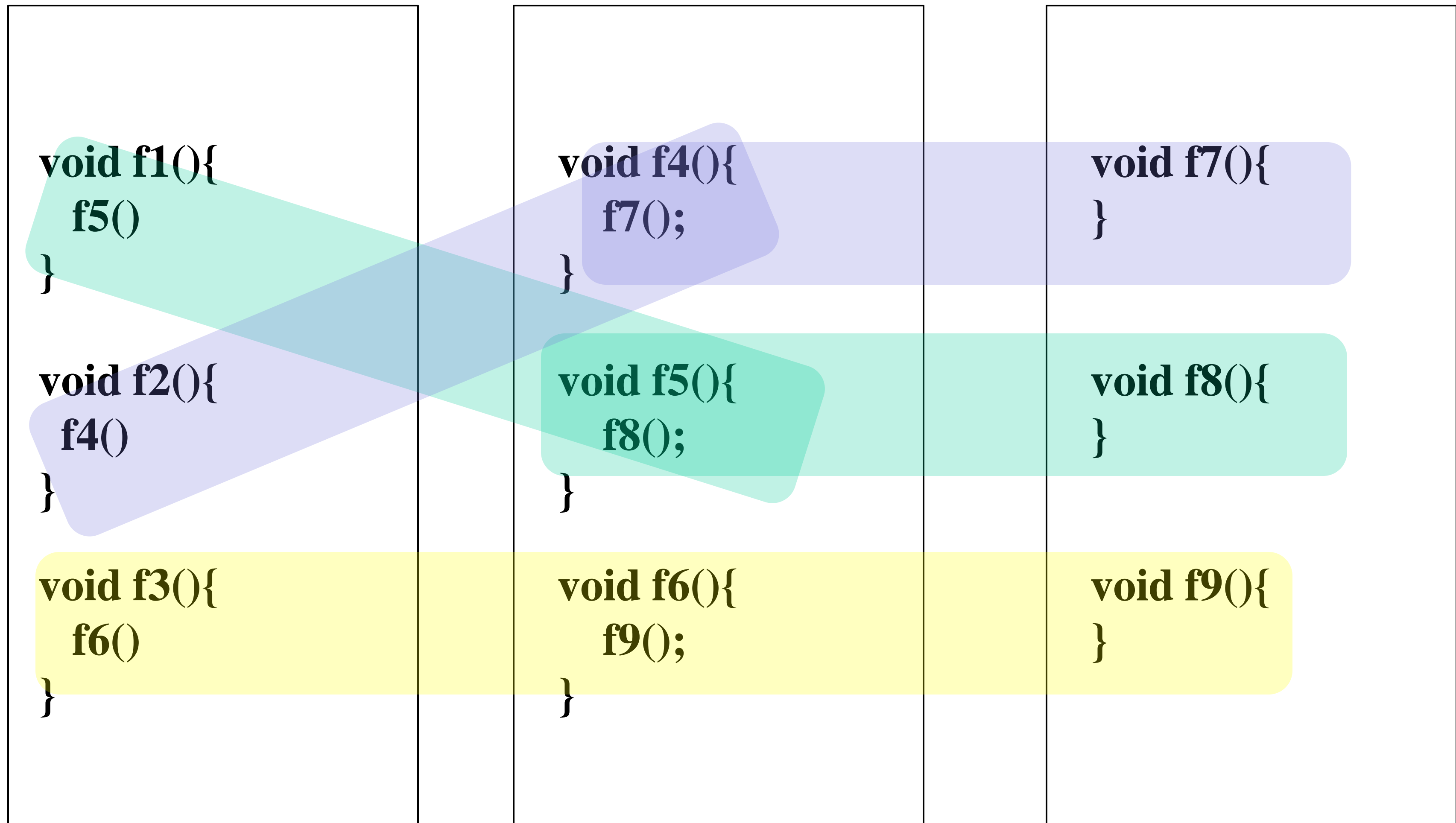
```
void f7(){
}

void f8(){
}

void f9(){
}
```

Changing statement order affects compilation

Aspect Oriented Programming

- Dynamic scope goal but a better approach
- Allowing the identification and separation of cross cutting concerns
- What are cross cutting concerns?
  - A concern is a particular set of behaviors needed by a computer program
  - cross-cutting concerns are aspects of a program which affect other concerns.

Cross cutting Concerns



Question: how can we identify related pieces of code?

GUI layer

```
void f1(){
  f5()
}

void f2(){
  f4()
}

void f3(){
  f6()
}
```

Logic layer

```
void f4(){
  f7();
}

void f5(){
  f8();
}

void f6(){
  f9();
}
```

Data layer

```
void f7(){
}

void f8(){
}

void f9(){
}
```

Aspect Oriented Programming

- Add separate code to document flow

```
void foo(char * a) {
    printf("inside foo, a = %s\n", a);
}

void foo2() {
     printf("in foo2, call foo\n");
     foo("ABCDE");
}

int main() {
    foo("abcde");
    foo2();
    return 0;
}
```

Foo.c

gcc foo.c

www.AspectC.net

```
before(): call(void foo(char *)) && infunc(main) {
        printf("aspect 1: call foo in main \n");
}

before(): call(void foo(char *)) && infunc(foo2) {
        printf("aspect 2: call foo in foo2\n");
}
```

Foo.acc

acc foo.c foo.acc