Runtime Support

Uday Khedker

(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

March 2025

Introduction

The Issues Addressed by Runtime Support

To decide on the organization of data objects, so that their addresses can be resolved at compile time

- The data objects (represented by variables) come into existence during the execution of the program
- The addresses of data objects depend on their organization in memory. This is, to a great extent, decided by source language features
- The generated code must refer to the data objects using their addresses, which must be decided during compilation
- The compiler generates code with addresses and runtime support facilitates creation of data and its access at run time

Information recorded in the symtabs is added to the code which allocates and accesses the memory at run time

Other Responsibilities of Runtime Support

Some examples of other roles and responsibilities of runtime support are

- Dynamic memory allocation and deallocation
- Garbage collection
- Exception handling,
- Virtual function resolution

Implementing Runtime Support

Run time support is implemented in two ways:

- For some activities, dedicated library is used at run time and the compiler merely calls the library functions
- For other activities, a compiler generates custom code using the information that is available

Implementing Runtime Support

Run time support is implemented in two ways:

- For some activities, dedicated library is used at run time and the compiler merely calls the library functions
- For other activities, a compiler generates custom code using the information that is available

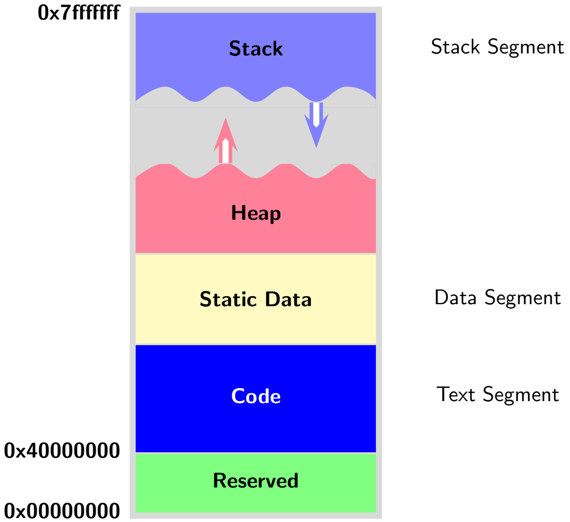
  We will study this

Primary Requirements of Runtime Support

The nature of run time support needed for executing a program is governed by

- the characteristics of data, and
- the characteristics of procedure invocations

SPIM Memory Architecture



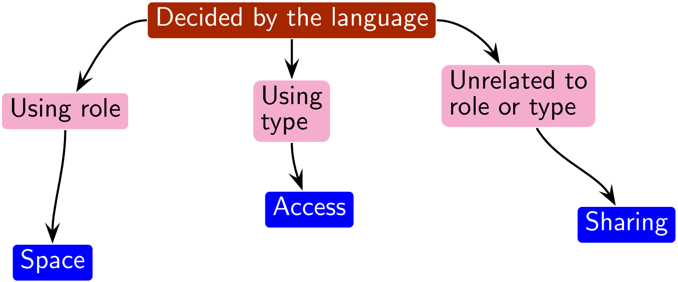
Characteristics of Data Specified by a Programmer

1. Programmer specifies the following properties of a data item
   - Type. Basic type, derived type
   - Role.
     - Named data declared statically
       global variables, local variables, or formal parameters
     - Unnamed data created dynamically
2. The language decides some further properties of the data
   (possibly using types and roles)
3. The properties are implemented by a compiler and its runtime support

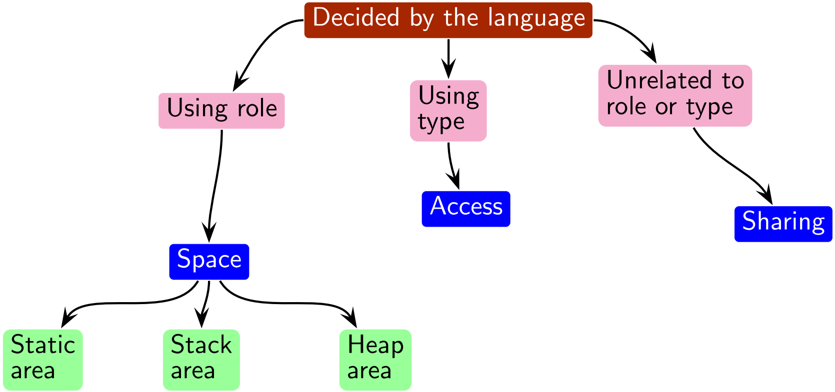
Characteristics of Data Decided by the Language

- Decided by the language
  - Using role
  - Using type
  - Unrelated to role or type

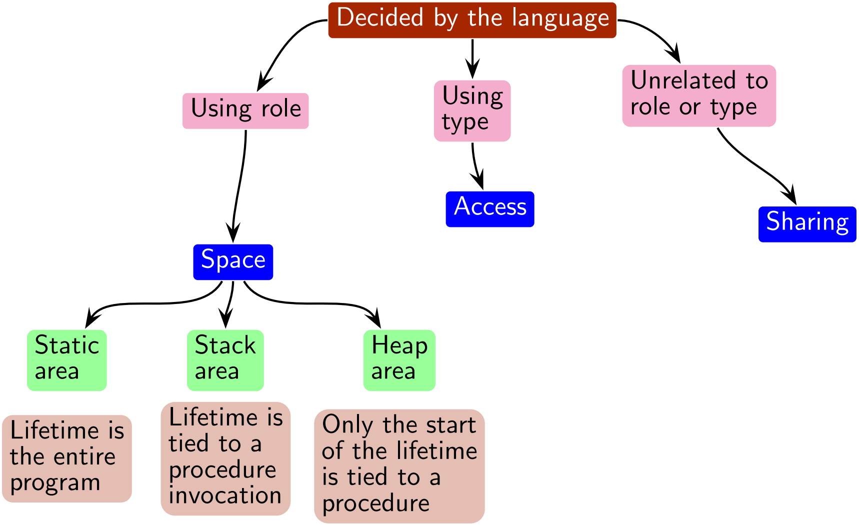
Characteristics of Data Decided by the Language



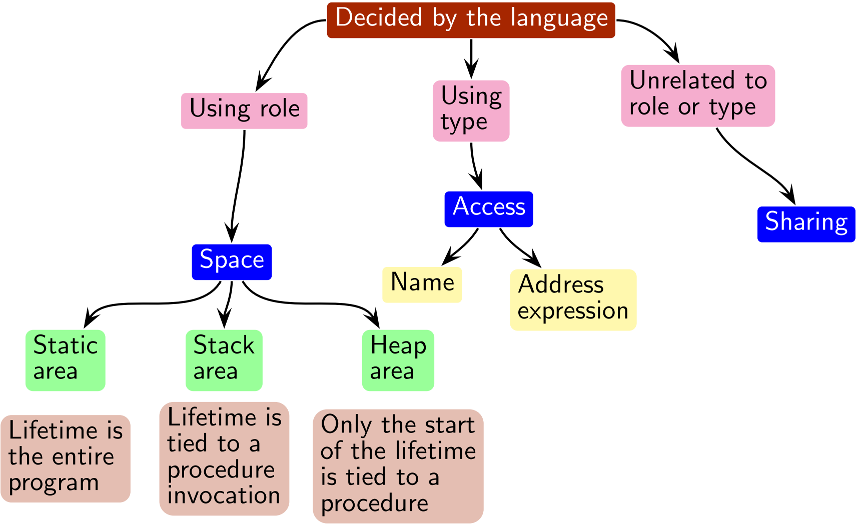
Characteristics of Data Decided by the Language



Characteristics of Data Decided by the Language



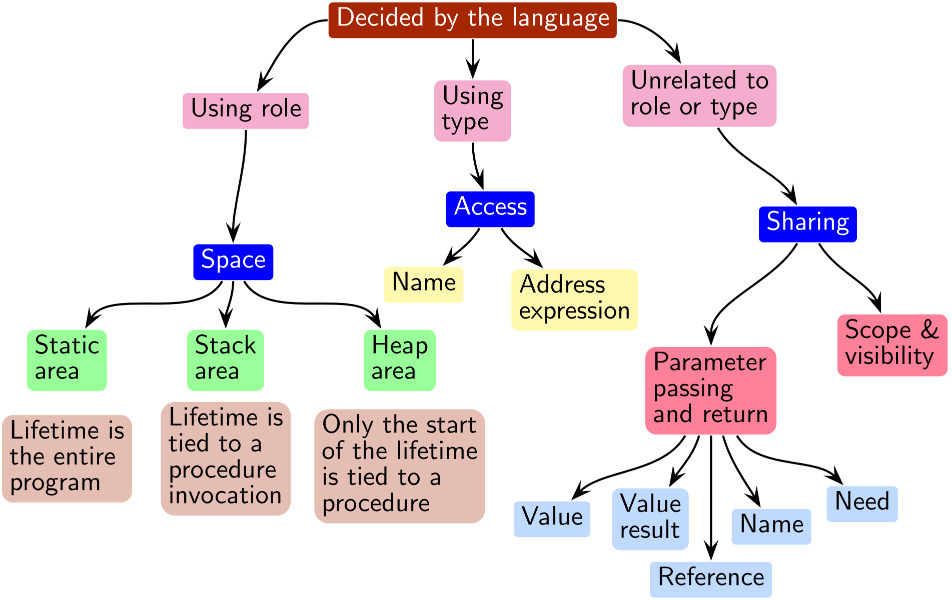
Characteristics of Data Decided by the Language



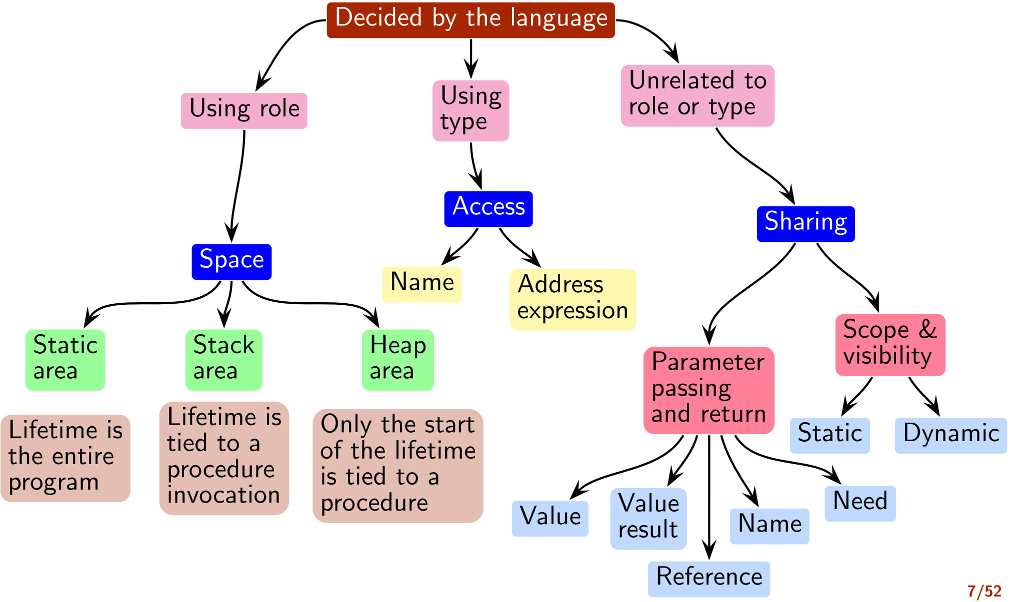
Characteristics of Data Decided by the Language

- Decided by the language
  - Using role
    - Space
      - Static area
        - Lifetime is the entire program
      - Stack area
        - Lifetime is tied to a procedure invocation
      - Heap area
        - Only the start of the lifetime is tied to a procedure
  - Using type
    - Access
      - Name
      - Address expression
  - Unrelated to role or type
    - Sharing
      - Parameter passing and return
      - Scope & visibility

Characteristics of Data Decided by the Language



Characteristics of Data Decided by the Language



Activation Records for Procedure Invocation

- Every invocation of a procedure requires creating an *activation record*
- An activation record provides space for
  1. local variables,
  2. parameters,
  3. saved registers (for values to be used across calls),
  4. return value,
  5. return address, and
  6. pointers to activation records of the calling procedures

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be
  - invoked only as subroutines (strict nesting of lifetimes of procedures)
  - invoked recursively (strict nesting of the lifetimes of the same procedure)
  - invoked indirectly through a function pointer or passed as a parameter

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be
  - invoked only as subroutines (strict nesting of lifetimes of procedures)
    Stack or static memory suffices for storing activation records
  - invoked recursively (strict nesting of the lifetimes of the same procedure)
  - invoked indirectly through a function pointer or passed as a parameter

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be
  - invoked only as subroutines (strict nesting of lifetimes of procedures)
    Stack or static memory suffices for storing activation records
  - invoked recursively (strict nesting of the lifetimes of the same procedure)
    Stack memory is required for organizing data for storing activation records
  - invoked indirectly through a function pointer or passed as a parameter

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be
  - invoked only as subroutines (strict nesting of lifetimes of procedures)
    Stack or static memory suffices for storing activation records
  - invoked recursively (strict nesting of the lifetimes of the same procedure)
    Stack memory is required for organizing data for storing activation records
  - invoked indirectly through a function pointer or passed as a parameter
    Access to non-local data of the procedure needs to be provided

Characteristics of Procedure Invocation

- A (sequential) language may allow procedures to be
  - invoked only as subroutines (strict nesting of lifetimes of procedures)
    Stack or static memory suffices for storing activation records
  - invoked recursively (strict nesting of the lifetimes of the same procedure)
    Stack memory is required for organizing data for storing activation records
  - invoked indirectly through a function pointer or passed as a parameter
    Access to non-local data of the procedure needs to be provided
- Support for parallelism and concurrency is a different ball game

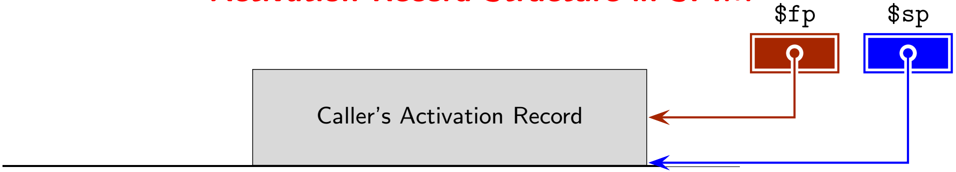
Compiling Procedure Calls

Activation Records in SCLP

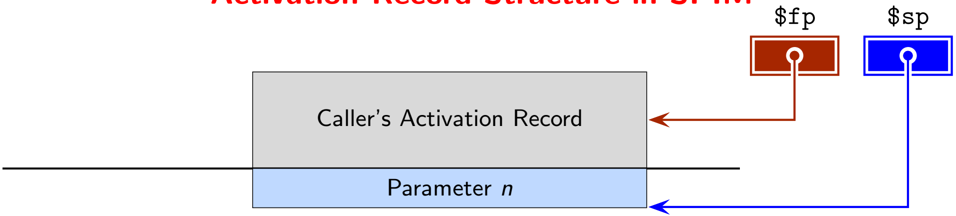
- General. An activation record provides space for
  1. local variables,
  2. parameters,
  3. saved registers (for values to be used across calls),
  4. return value,
  5. return address, and
  6. pointers to activation records of the calling procedures
- SCLP. An activation record provides space for
  1. local variables,
  2. parameters,
  3. return address, and
  4. pointers to activation records of the calling procedures

  Return value is in register `$v1` and no registers are live across calls

Activation Record Structure in SPIM

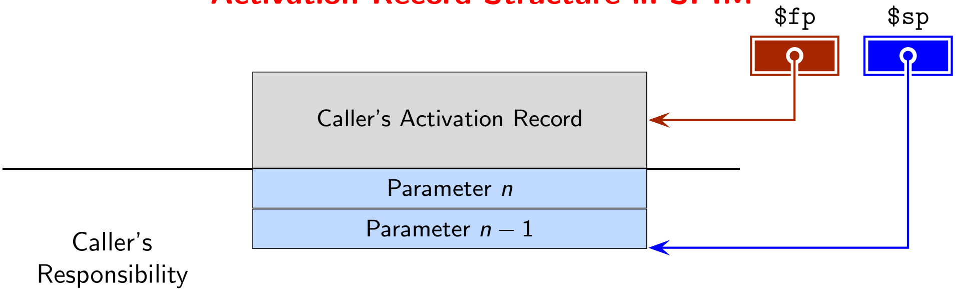


Activation Record Structure in SPIM

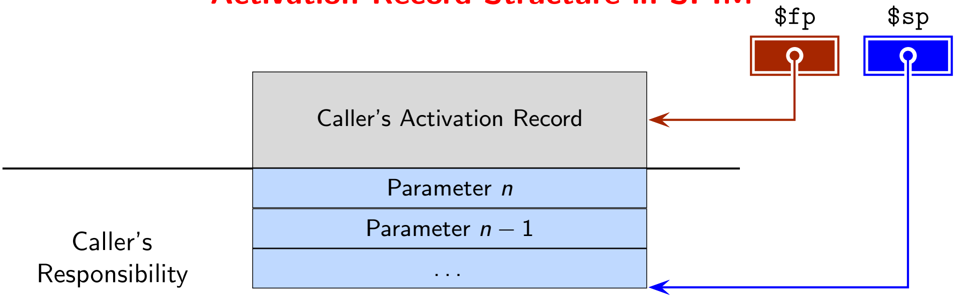


Caller's Responsibility

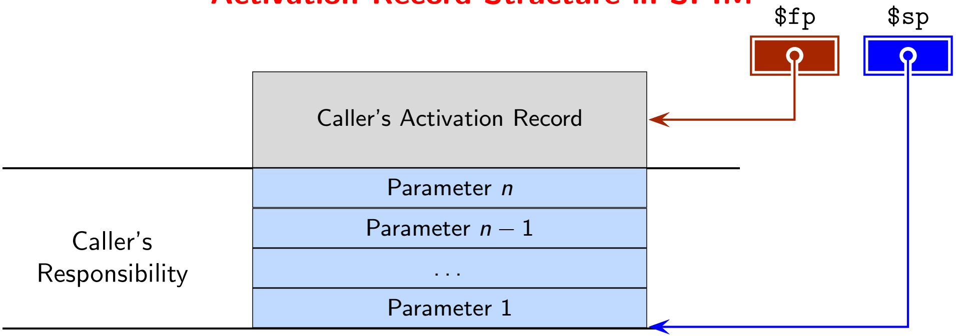
Activation Record Structure in SPIM



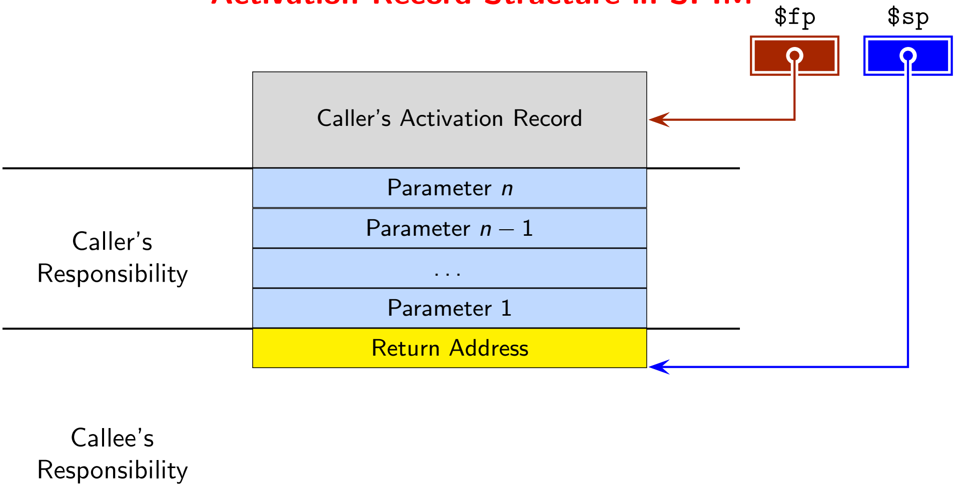
Activation Record Structure in SPIM



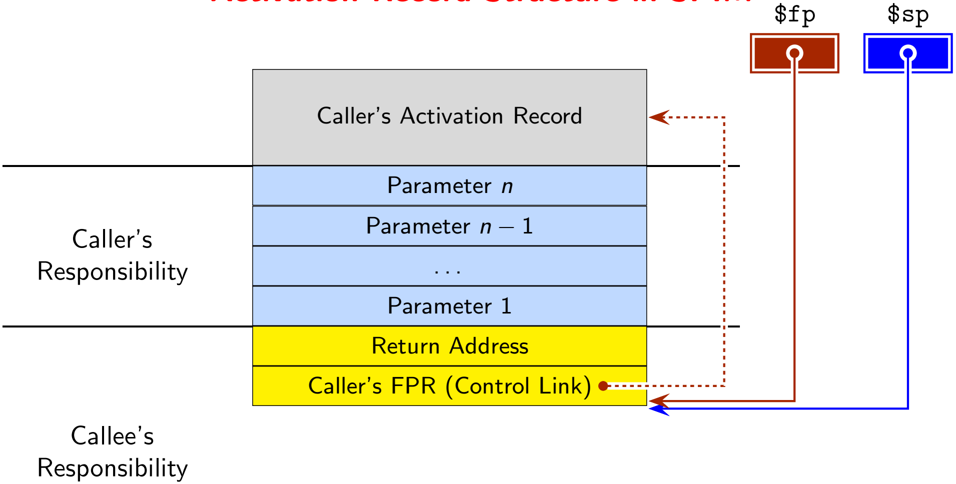
Activation Record Structure in SPIM



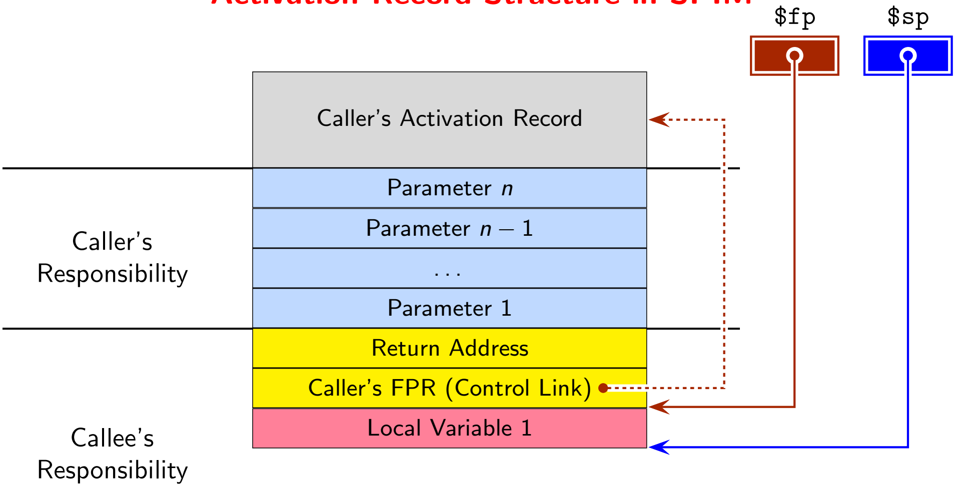
Activation Record Structure in SPIM



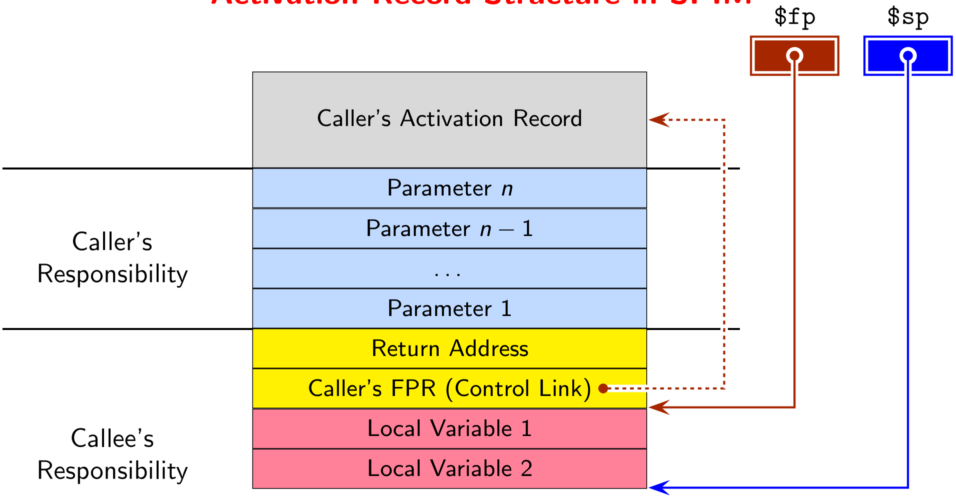
Activation Record Structure in SPIM



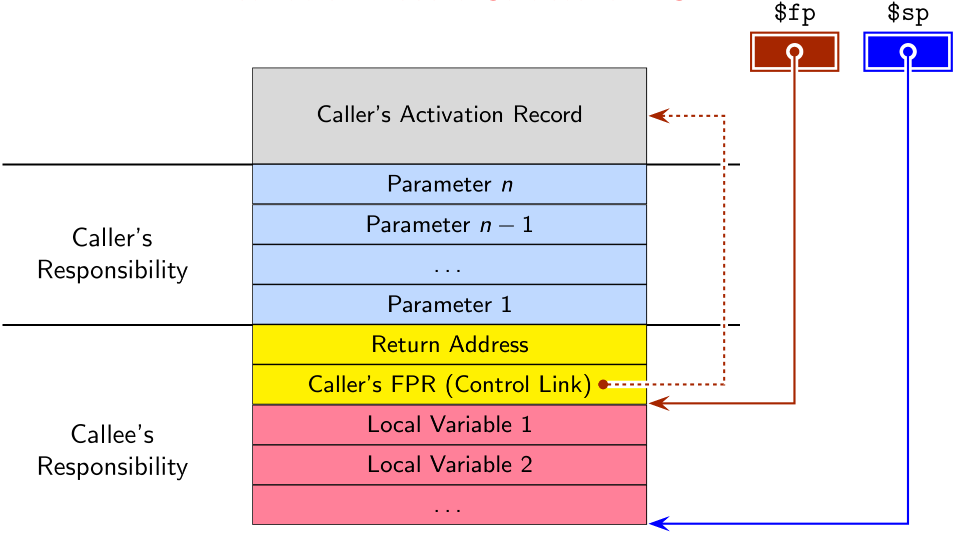
Activation Record Structure in SPIM



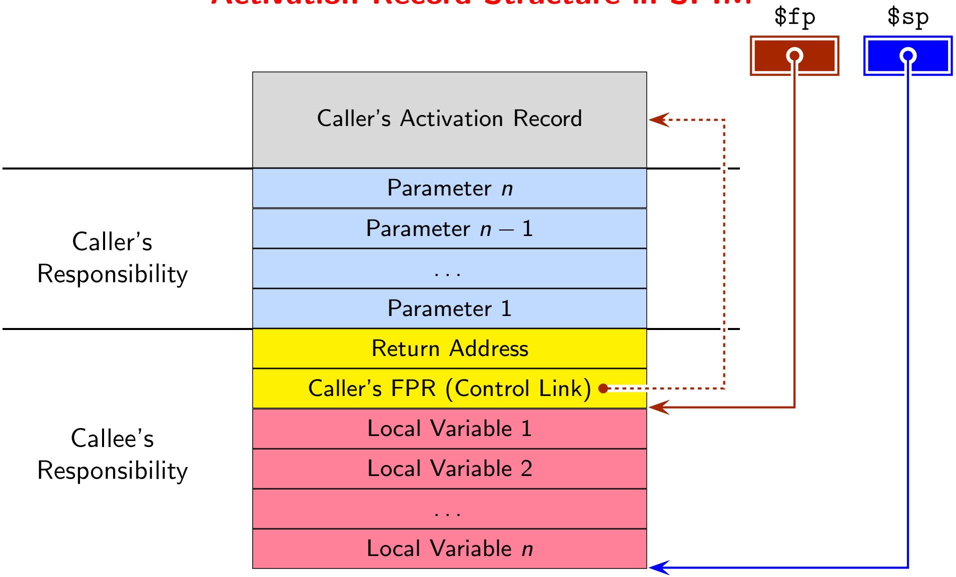
Activation Record Structure in SPIM



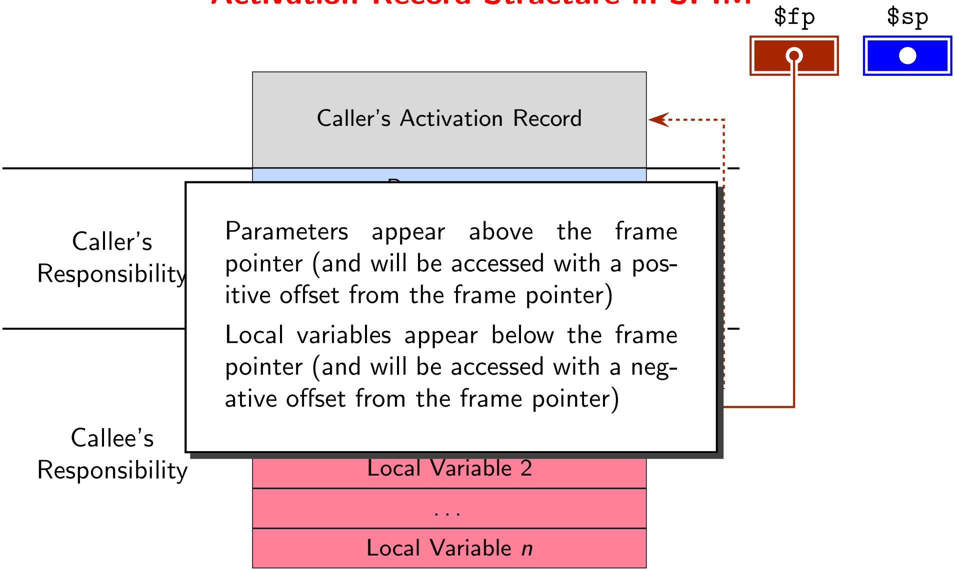
Activation Record Structure in SPIM



Activation Record Structure in SPIM



Activation Record Structure in SPIM



Parameters appear above the frame pointer (and will be accessed with a positive offset from the frame pointer)

Local variables appear below the frame pointer (and will be accessed with a negative offset from the frame pointer)

Activation Record Structure in SPIM

$fp $sp

Caller's Activation Record

Caller's Responsibility

Callee's Responsibility

$fp and $ra need to be saved because

- they are modified by a call,
- their new values are required for the callee's execution, and
- their previous values are required for the caller's execution after the call.

. . .

Local Variable *n*

Setting Up the Activation Record

- Caller's activities are done by the code just before and just after a call
  - Pushing of parameters happens just before the call
  - Popping of parameters happens just after the call

  These statements appear in both RTL IR and assembly code emitted by the current reference implementation

Setting Up the Activation Record

- Caller's activities are done by the code just before and just after a call
  - Pushing of parameters happens just before the call
  - Popping of parameters happens just after the call

  These statements appear in both RTL IR and assembly code emitted by the current reference implementation

- Callee's activities are done by the code in the beginning and in the end of code of the callee
  - Saving of return address (register `$ra`), frame pointer (register `$fp`), and making space for local variables happens at the start
  - The stack is restored at the end

  These statements appear only in the assembly code and not RTL IR emitted by the current reference implementation

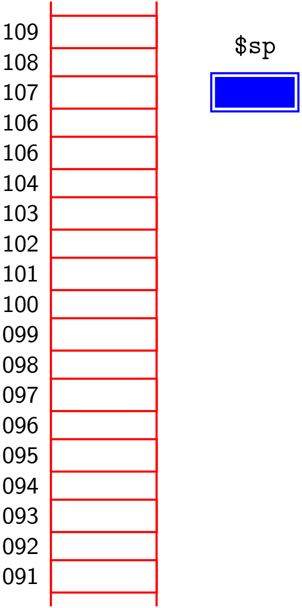
Setting Up the Activation Record

- Caller's activities are done by the code just before and just after a call
  - Pushing of parameters happens just before the call
  - Popping of parameters happens just after the call

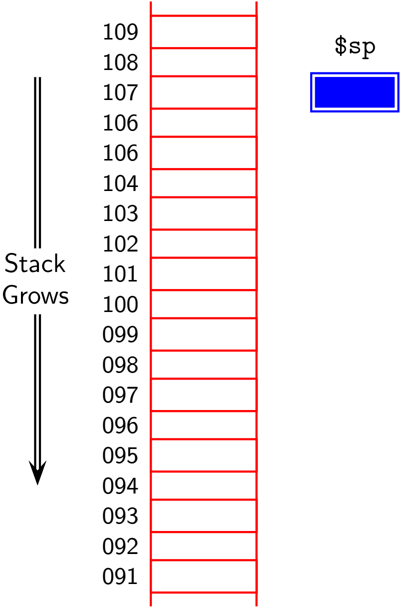
  These statements appear in both RTL IR and assembly code emitted by the current reference implementation
- Callee's activities are done by the code in the beginning and in the end of code of the callee
  - Saving of return address (register `$ra`), frame pointer (register `$fp`), and making space for local variables happens at the start
  - The stack is restored at the end

  These statements appear only in the assembly code and not RTL IR emitted by the current reference implementation
- The push operation is implemented by decrementing register `$sp` using `sub` instruction whereas the pop operation is implemented by incrementing register `$sp` using `add` instruction

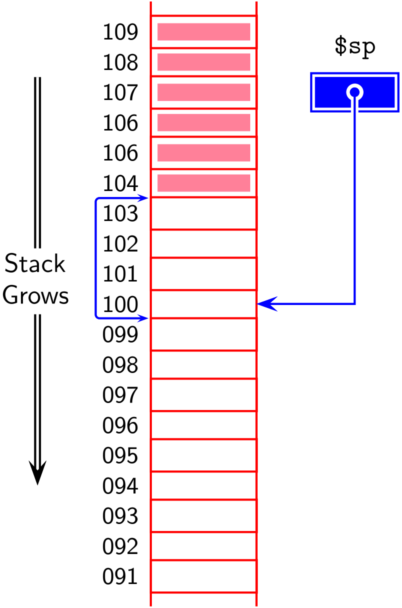
Accessing the Memory



Accessing the Memory

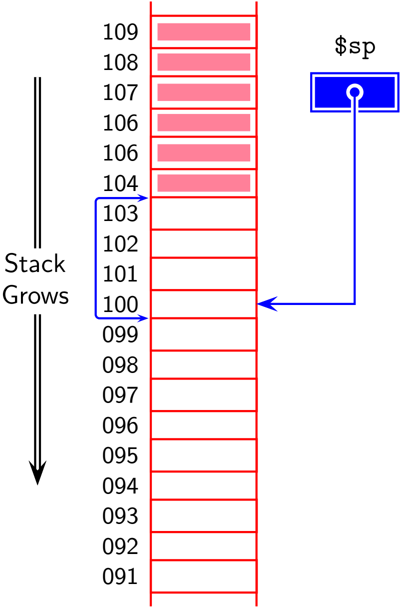


Accessing the Memory



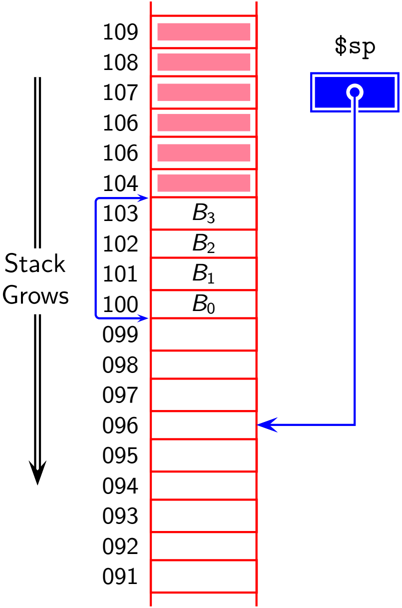
- The address of a word is always the address of the lowest byte of the word
- The Endinaness is orthogonal; the most significant byte may be at the lower or the higher address depending upon the hardware on which the simulator runs
- The stack pointer `$sp` points to the lower address of the next free location

Accessing the Memory



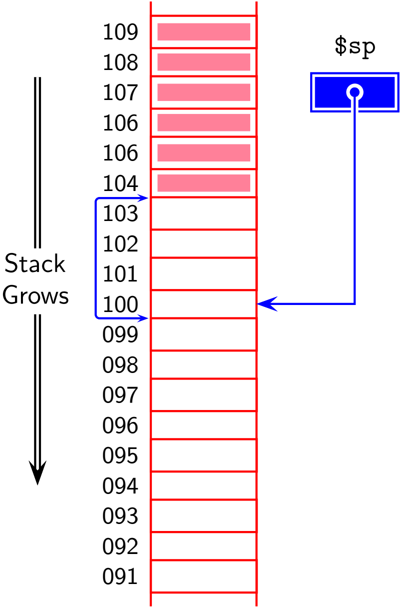
- The address of a word is always the address of the lowest byte of the word
- The Endinaness is orthogonal; the most significant byte may be at the lower or the higher address depending upon the hardware on which the simulator runs
- The stack pointer `$sp` points to the lower address of the next free location
  - An integer value needs 4 bytes and is stored using address `0($sp)`, `$sp` is decremented by 4

Accessing the Memory



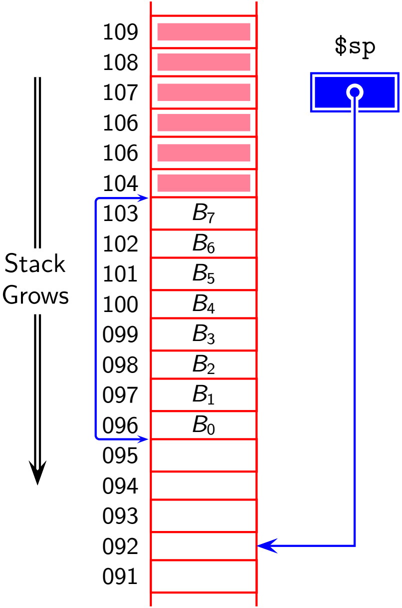
- The address of a word is always the address of the lowest byte of the word
- The Endinaness is orthogonal; the most significant byte may be at the lower or the higher address depending upon the hardware on which the simulator runs
- The stack pointer `$sp` points to the lower address of the next free location
    - An integer value needs 4 bytes and is stored using address `0($sp)`, `$sp` is decremented by 4

Accessing the Memory



- The address of a word is always the address of the lowest byte of the word
- The Endinaness is orthogonal; the most significant byte may be at the lower or the higher address depending upon the hardware on which the simulator runs
- The stack pointer `$sp` points to the lower address of the next free location
  - An integer value needs 4 bytes and is stored using address `0($sp)`, `$sp` is decremented by 4
  - A double value needs 8 bytes and is stored using address `-4($sp)`, `$sp` is decremented by 8

Accessing the Memory



- The address of a word is always the address of the lowest byte of the word
- The Endinaness is orthogonal; the most significant byte may be at the lower or the higher address depending upon the hardware on which the simulator runs
- The stack pointer `$sp` points to the lower address of the next free location
    - An integer value needs 4 bytes and is stored using address `0($sp)`, `$sp` is decremented by 4
    - A double value needs 8 bytes and is stored using address `-4($sp)`, `$sp` is decremented by 8
- Unlike the stack pointer, the frame pointer (`$fp`) holds the address of an occupied word

Accessing Data in an Activation Record

	Parameter n
	Parameter $n-1$
	...
	Parameter 1
	Return Address
	Caller's FPR (Control Link)
`$fp`	Local Variable 1
	Local Variable 2
	...
	Local Variable n
`$sp`	

Accessing Data in an Activation Record

	Parameter n
	Parameter $n-1$
	. . .
	Parameter 1
	Return Address
	Caller's FPR (Control Link)
`$fp`	Local Variable 1
	Local Variable 2
	. . .
`$sp`	Local Variable n

Let the size of i^{th} parameter be denoted by p_i and that of i^{th} local variable be denoted by l_i

Accessing Data in an Activation Record

	Parameter n
	Parameter $n-1$
	...
	Parameter 1
	Return Address
	Caller's FPR (Control Link)
`$fp`	Local Variable 1
	Local Variable 2
	...
`$sp`	Local Variable n

Let the size of i^{th} parameter be denoted by p_i and that of i^{th} local variable be denoted by l_i

Variable	Address	Size
Parameter 1	`8($fp)`	p_1
Parameter 2	$(8{+}p_1)$`($fp)`	p_2
Parameter 3	$(8{+}p_1{+}p_2)$`($fp)`	p_3

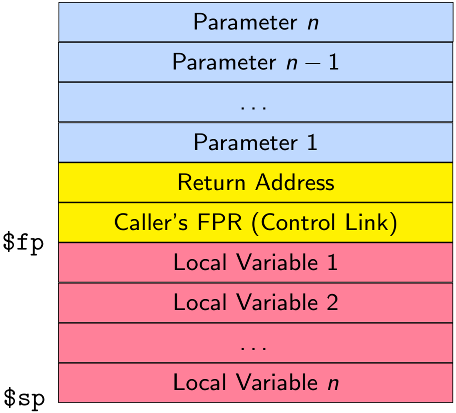
Accessing Data in an Activation Record

	Parameter n
	Parameter $n-1$
	...
	Parameter 1
	Return Address
	Caller's FPR (Control Link)
`$fp`	Local Variable 1
	Local Variable 2
	...
`$sp`	Local Variable n

Let the size of i^{th} parameter be denoted by p_i and that of i^{th} local variable be denoted by l_i

Variable	Address	Size
Parameter 1	`8($fp)`	p_1
Parameter 2	$(8+p_1)$`($fp)`	p_2
Parameter 3	$(8+p_1+p_2)$`($fp)`	p_3
Local 1	$-l_1$`($fp)`	l_1
Local 2	$-(l_1+l_2)$`($fp)`	l_2
Local 3	$-(l_1+l_2+l_3)$`($fp)`	l_3

Accessing Data in an Activation Record



Let the size of i^{th} parameter be denoted by p_i and that of i^{th} local variable be denoted by l_i

Variable	Address	Size
Parameter 1	8($fp)	p_1
Parameter 2	$(8+p_1)$($fp)	p_2
Parameter 3	$(8+p_1+p_2)$($fp)	p_3
Local 1	$-l_1$($fp)	l_1
Local 2	$-(l_1+l_2)$($fp)	l_2
Local 3	$-(l_1+l_2+l_3)$($fp)	l_3

The expressions representing the offsets are computed at compile time and the code contains the generated numbers

Accessing Data in an Activation Record

	Parameter n
	Parameter $n-1$
	...
	Parameter 1
	Return Address
	Caller's FPR (Control Link)
`$fp`	Local Variable 1
	Local Variable 2
	...
`$sp`	Local Variable n

Let the size of i^{th} parameter be denoted by p_i and that of i^{th} local variable be denoted by l_i

Variable	Address	Size
Parameter 1	`8($fp)`	p_1
Parameter 2	$(8+p_1)$`($fp)`	p_2
Parameter 3	$(8+p_1+p_2)$`($fp)`	p_3
Local 1	$-l_1$`($fp)`	l_1
Local 2	$-(l_1+l_2)$`($fp)`	l_2
Local 3	$-(l_1+l_2+l_3)$`($fp)`	l_3

The expressions representing the offsets are computed at compile time and the code contains the generated numbers

Use option `--show-symtab` to see the addresses assigned by sclp

Examining the Offsets Assigned in SCLP


```
int f(int a,
       int b)
{
  int c;
  c = a+b;
  return c;
}

int main()
{
  int x, y, z;

  z = f(x,y);

  print z;
  return 0;
}
```

```
Global Declarations:

**PROCEDURE: f_, Return Type:<int>
Formal Parameters
 Name: a_<int> Entity Type:VAR Start Offset:8 End Offset:12
 Name: b_<int> Entity Type:VAR Start Offset:12 End Offset:16
Local Declarartions
 Name: c_<int> Entity Type:VAR Start Offset:-4 End Offset: 0

**PROCEDURE: main, Return Type:<int>
Formal Parameters
Local Declarartions
 Name: x_<int> Entity Type:VAR Start Offset:-4 End Offset:0
 Name: y_<int> Entity Type:VAR Start Offset:-8 End Offset:-4
 Name: z_<int> Entity Type:VAR Start Offset:-12 End Offset:-8
```

Example of Function Prologue, Epilogue, and Call in SCLP

```
int f(int a, int b)
{
        int c;
        c = a+b;
        return c;
}

int main()
{
        int x, y, z;

        z = f(x,y);

        print z;
        return 0;
}
```

Prologue, Epilogue, and Formal Parameter Access in Function f

```
f_:
# Prologue begins
sw $ra, 0($sp)       # Save the return address
sw $fp, -4($sp)      # Save the frame pointer
sub $fp, $sp, 4      # Update the frame pointer
sub $sp, $sp, 12     # Make space for locals, $ra, and $sp
# Prologue ends
  ...
lw $v0, 8($fp)       # Source:a_ (first parameter)
lw $t1, 12($fp)      # Source:b_ (second parameter)
add $t0, $v0, $t1    # Result: $t0, Opd1: $v0, Opd2:$t1
sw $t0, -4($fp)      # Dest: c_ (the lone local)
  ...
epilogue_f_:
add $sp, $sp, 12     # Remove the space of locals, $ra, and $sp
lw $fp, -4($sp)      # Set $fp to $sp-4
lw $ra, 0($sp)       # Restore ra
jr $ra               # Jump back to the called procedure
# Epilogue Ends
```

Code For the Call in Function main

```
lw $v0, -8($fp)      # Source:y_ (second local)
sw $v0, 0($sp)       # store formal parameter at sp
sub $sp, $sp, 4      # decrement the stack pointer by 4
lw $v0, -4($fp)      # Source:x_ (first local)
sw $v0, 0($sp)       # store formal parameter at sp
sub $sp, $sp, 4      # decrement the stack pointer by 4
jal f_
add $sp, $sp, 4      # increment the stack pointer by 4 (pop x_)
add $sp, $sp, 4      # increment the stack pointer by 4 (pop y_)
move $v0, $v1        # store function call result from $v1 in $v0
sw $v0, -12($fp)     # Dest: z_ (third local)
```

Arrays as Parameters in C/C++

- Only the address of the array is passed as the actual parameter
  Space requirement in an activation record: space of a pointer
- Accesses in the callee procedure require address calculation
  - For a 1-D array, the size is not needed for address calculation
    A call could look like `foo(int arr[]);`
  - For 2-D array, the size of the second dimension is needed
    A call could look like `foo(int arr[][10]);`
    The offset of the element accessed using `arr[i][j]);` is $\boxed{i \times 10 + j}$
- Sizes of all dimensions except for the first dimension, are needed in the call
- In C++, references to arrays can be passed which retain the size information while passing a pointer to the array
- In C++, when containers (`std::array` and `std::vector`) are passed they involve additional space allocation

Accessing Non-Local Data

- Under static scoping

  Use access link (aka static link) in an activation record which points to the base of the activation record of the enclosing procedure

  Compile-time relationship

- Under dynamic scoping

  Use dynamic link (aka control link) in an activation record which points to the base of the activation record of the callee procedure

  Runtime relationship

Accessing Non-Local Data Under Static Scoping

- Assign levels to procedures as follows
  - The level of the main procedure, denoted l_{main} is 1
  - If procedure P is contained immediately within Q, then $l_P = l_Q + 1$
- Let $symtab_P$ denote the symbol table of P — Compile-time data structure
- Let AR_P denote the activation record of procedure P — Runtime data structure
- If some (non-local) variable x is accessed in procedure P, the compiler searches for its entry in a symtab starting from the top of the symtab stack

  If an entry of x is found in $symtab_Q$ but not in any $symtab_R$ that is above $symtab_Q$ in the stack, the compiler generates code to
  - Access AR_Q by traversing $l_P - l_Q$ access links starting from AR_P
  - Access x by using the offset of x with respect to the base of AR_Q

Accessing Non-Local Data Under Dynamic Scoping

- Let AR_P must be augmented with information populated in $symtab_P$ (such as name, type, offset, etc.)
- If some (non-local) variable x is accessed in procedure P the compiler generates code for searching for its entry in an activation record on the control stack
- This involves generating code for
  - repeatedly accessing, starting from AR_P, the dynamic link (aka control link) until an entry of x is found in some AR_Q
  - accessing x by using the offset of x with respect to the base of AR_Q in which the entry of x is found

Runtime data structure

Using the Static Link or Access Link

```
int main()              // L_main = 1
{ void S()              // L_S = 2
  { int x, z;
    void R()            // L_R = 3
    { int i;
      int T()           // L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            // L_E = 3
    { // body of E    }
    void Q()            // L_Q = 3
    { int a, x;
      int P(int y)      // L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence
$main \rightarrow S \rightarrow Q \rightarrow Q \rightarrow P \rightarrow R \rightarrow T$

Using the Static Link or Access Link

```
int main()                    //L_main = 1
{ void S()                    //L_S = 2
  { int x, z;
    void R()                  //L_R = 3
    { int i;
      int T()                 //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                  //L_E = 3
    { // body of E   }
    void Q()                  //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence
$main \rightarrow S \rightarrow Q \rightarrow Q \rightarrow P \rightarrow R \rightarrow T$

AR_{main}	
AR_S	x, z
AR_Q	a, x
AR_Q	a, x
AR_P	y, i, j
AR_R	i
AR_T	m, n

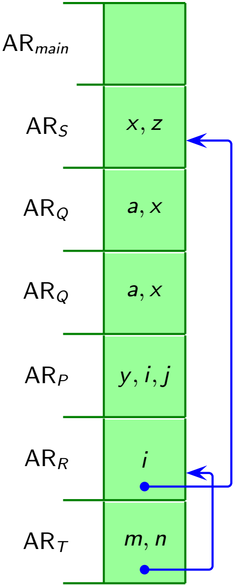
Using the Static Link or Access Link

```
int main()              //L_main = 1
{ void S()              //L_S = 2
  { int x, z;
    void R()            //L_R = 3
    { int i;
      int T()      //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            //L_E = 3
    { // body of E   }
    void Q()            //L_Q = 3
    { int a, x;
      int P(int y) //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence
$main \rightarrow S \rightarrow Q \rightarrow Q \rightarrow P \rightarrow R \rightarrow T$

- x in T corresponds to $S : x$

  The compiler generates code to traverse $l_T - l_S = 4 - 2 = 2$ access links from T, reaching the base of AR_S

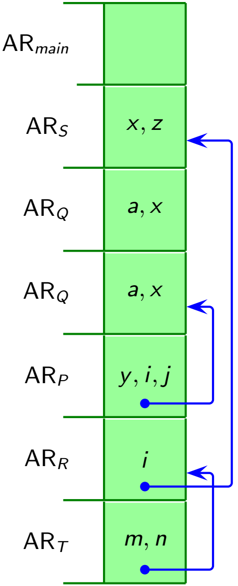


Using the Static Link or Access Link

```
int main()              //L_main = 1
{ void S()              //L_S = 2
  { int x, z;
    void R()            //L_R = 3
    { int i;
      int T()       //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            //L_E = 3
    { // body of E   }
    void Q()            //L_Q = 3
    { int a, x;
      int P(int y) //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence
$main \rightarrow S \rightarrow Q \rightarrow Q \rightarrow P \rightarrow R \rightarrow T$

- x in T corresponds to $S : x$
  The compiler generates code to traverse $l_T - l_S = 4 - 2 = 2$ access links from T, reaching the base of AR_S
- x in P corresponds to $Q : x$
  The compiler generates code to traverse $l_P - l_Q = 4 - 3 = 1$ access link from P, reaching the base of AR_Q

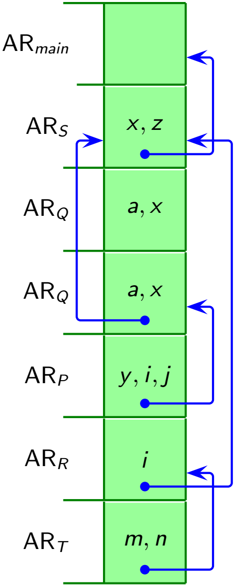


Using the Static Link or Access Link

```
int main()                 //L_main = 1
{ void S()                 //L_S = 2
  { int x, z;
    void R()               //L_R = 3
    { int i;
      int T()        //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()               //L_E = 3
    { // body of E    }
    void Q()               //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence
$main \rightarrow S \rightarrow Q \rightarrow Q \rightarrow P \rightarrow R \rightarrow T$

- x in T corresponds to $S : x$
  The compiler generates code to traverse $l_T - l_S = 4 - 2 = 2$ access links from T, reaching the base of AR_S
- x in P corresponds to $Q : x$
  The compiler generates code to traverse $l_P - l_Q = 4 - 3 = 1$ access link from P, reaching the base of AR_Q
- z in P corresponds to $S : z$
  The compiler generates code to traverse $l_P - l_S = 4 - 2 = 2$ access links from P, reaching the base of AR_S



Setting the Static Link

```
int main()              //L_main = 1
{ void S()              //L_S = 2
  { int x, z;
    void R()            //L_R = 3
    { int i;
      int T()           //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            //L_E = 3
    { // body of E   }
    void Q()            //L_Q = 3
    { int a, x;
      int P(int y)      //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence $main \rightarrow S \rightarrow Q_1 \rightarrow Q_2 \rightarrow P \rightarrow R \rightarrow T$ where we distinguish between the two invocations of Q by Q_1 and Q_2

Set up the access link in the callee's activation record from those in the caller's activation record as follows

Setting the Static Link

```
int main()              // L_main = 1
{ void S()              // L_S = 2
  { int x, z;
    void R()            // L_R = 3
    { int i;
      int T()     // L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            // L_E = 3
    { // body of E   }
    void Q()            // L_Q = 3
    { int a, x;
      int P(int y) // L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence $main \rightarrow S \rightarrow Q_1 \rightarrow Q_2 \rightarrow P \rightarrow R \rightarrow T$ where we distinguish between the two invocations of Q by Q_1 and Q_2

Set up the access link in the callee's activation record from those in the caller's activation record as follows

- Consider l_{caller} and l_{callee}

$$\begin{aligned} l_{callee} &\leq l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

A callee procedure must be either

- outside of the caller procedure ($l_{callee} < l_{caller}$), or
- at the same level, just before the caller in the enclosing procedure ($l_{callee} = l_{caller}$), or
- within the caller procedure ($l_{callee} = l_{caller} + 1$).

Setting the Static Link

```
int main()                    //L_main = 1
{ void S()                    //L_S = 2
  { int x, z;
    void R()                  //L_R = 3
    { int i;
      int T()       //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                  //L_E = 3
    { // body of E   }
    void Q()                  //L_Q = 3
    { int a, x;
      int P(int y) //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence $main \rightarrow S \rightarrow Q_1 \rightarrow Q_2 \rightarrow P \rightarrow R \rightarrow T$ where we distinguish between the two invocations of Q by Q_1 and Q_2

Set up the access link in the callee's activation record from those in the caller's activation record as follows

- Consider l_{caller} and l_{callee}

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to

Setting the Static Link

```
int main()                    //L_main = 1
{ void S()                    //L_S = 2
  { int x, z;
    void R()                  //L_R = 3
    { int i;
      int T()          //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                  //L_E = 3
    { // body of E    }
    void Q()                  //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence $main \rightarrow S \rightarrow Q_1 \rightarrow Q_2 \rightarrow P \rightarrow R \rightarrow T$ where we distinguish between the two invocations of Q by Q_1 and Q_2

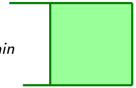
Set up the access link in the callee's activation record from those in the caller's activation record as follows

- Consider l_{caller} and l_{callee}

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to

AR_{main}



Setting the Static Link

```
int main()              // L_main = 1
{ void S()              // L_S = 2
  { int x, z;
    void R()            // L_R = 3
    { int i;
      int T()           // L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            // L_E = 3
    { // body of E   }
    void Q()            // L_Q = 3
    { int a, x;
      int P(int y)      // L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

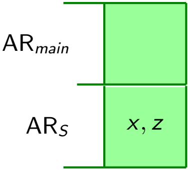
$l_{\text{caller}} = l_{main} = 1$
$l_{\text{callee}} = l_S = 2$
$n = 1 - 2 + 1 = 0$

Traverse 0 access links from AR_{main}

Set the access link of AR_S to the base of AR_{main}

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                //L_main = 1
{ void S()                //L_S = 2
  { int x, z;
    void R()              //L_R = 3
    { int i;
      int T()       //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()              //L_E = 3
    { // body of E   }
    void Q()              //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

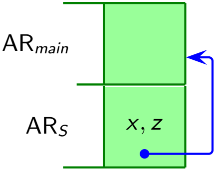
$l_{caller} = l_{main} = 1$
$l_{callee} = l_S = 2$
$n = 1 - 2 + 1 = 0$

Traverse 0 access links from AR_{main}

Set the access link of AR_S to the base of AR_{main}

$$\begin{aligned} l_{callee} &\leq l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

- Traverse $n = l_{caller} - l_{callee} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

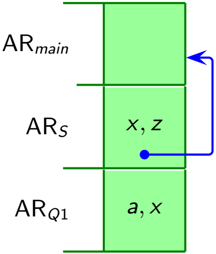
```
int main()                   //L_main = 1
{ void S()                   //L_S = 2
  { int x, z;
    void R()                 //L_R = 3
    { int i;
      int T()       //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                 //L_E = 3
    { // body of E   }
    void Q()                 //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence main →

$l_{\text{caller}} = l_S = 2$
$l_{\text{callee}} = l_Q = 3$
$n = 2 - 3 + 1 = 0$

Traverse 0 access link from AR_S

Set the access link of AR_{Q1} to the base of AR_S



$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to

Setting the Static Link

```
int main()               //L_main = 1
{ void S()               //L_S = 2
  { int x, z;
    void R()             //L_R = 3
    { int i;
      int T()            //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()             //L_E = 3
    { // body of E   }
    void Q()             //L_Q = 3
    { int a, x;
      int P(int y)       //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

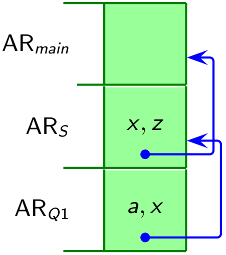
$l_{\text{caller}} = l_S = 2$
$l_{\text{callee}} = l_Q = 3$
$n = 2 - 3 + 1 = 0$

Traverse 0 access link from AR_S

Set the access link of AR_{Q1} to the base of AR_S

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()              //L_main = 1
{ void S()              //L_S = 2
  { int x, z;
    void R()            //L_R = 3
    { int i;
      int T()           //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            //L_E = 3
    { // body of E   }
    void Q()            //L_Q = 3
    { int a, x;
      int P(int y)      //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

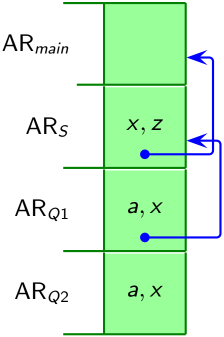
$l_{\text{caller}} = l_Q = 3$
$l_{\text{callee}} = l_Q = 3$
$n = 3 - 3 + 1 = 1$

Traverse 1 access link from AR_{Q1} of caller Q

Set the access link of AR_{Q2} of callee Q to the base of AR_S

$$
\begin{aligned}
l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\
l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\
n &= l_{\text{caller}} - l_{\text{callee}} + 1
\end{aligned}
$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                  //L_main = 1
{ void S()                  //L_S = 2
  { int x, z;
    void R()                //L_R = 3
    { int i;
      int T()           //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                //L_E = 3
    { // body of E   }
    void Q()                //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

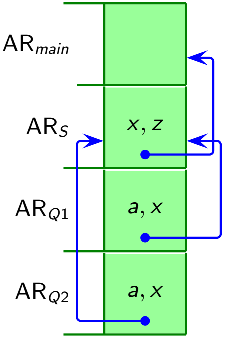
$l_{caller} = l_Q = 3$
$l_{callee} = l_Q = 3$
$n = 3 - 3 + 1 = 1$

Traverse 1 access link from AR_{Q1} of caller Q

Set the access link of AR_{Q2} of callee Q to the base of AR_S

$$\begin{aligned} l_{callee} &\leq l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

- Traverse $n = l_{caller} - l_{callee} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                    //L_main = 1
{ void S()                    //L_S = 2
  { int x, z;
    void R()                  //L_R = 3
    { int i;
      int T()        //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                  //L_E = 3
    { // body of E   }
    void Q()                  //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

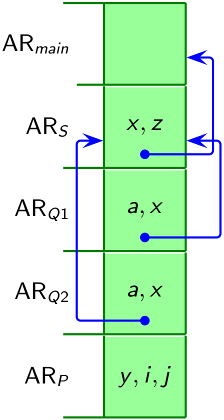
$l_{\text{caller}} = l_Q = 3$
$l_{\text{callee}} = l_P = 4$
$n = 3 - 4 + 1 = 0$

Traverse 0 access links from AR_Q

Set the access link of AR_P to the base of AR_{Q2}

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()              //L_main = 1
{ void S()              //L_S = 2
  { int x, z;
    void R()            //L_R = 3
    { int i;
      int T()           //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()            //L_E = 3
    { // body of E    }
    void Q()            //L_Q = 3
    { int a, x;
      int P(int y)      //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

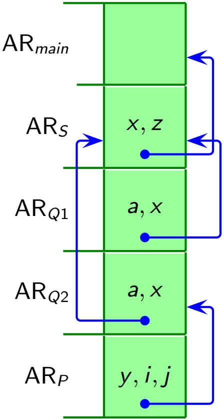
$l_{\text{caller}} = l_Q = 3$
$l_{\text{callee}} = l_P = 4$
$n = 3 - 4 + 1 = 0$

Traverse 0 access links from AR_Q

Set the access link of AR_P to the base of AR_{Q2}

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                  //L_main = 1
{ void S()                  //L_S = 2
  { int x, z;
    void R()                //L_R = 3
    { int i;
      int T()       //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                //L_E = 3
    { // body of E    }
    void Q()                //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

Consider the call sequence main →

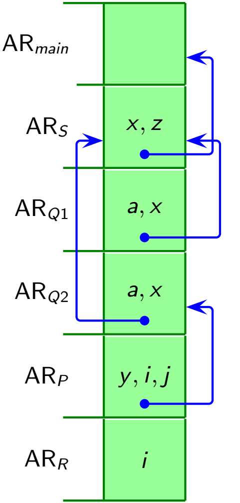
$l_{caller} = l_P = 4$
$l_{callee} = l_R = 3$
$n = 4 - 3 + 1 = 2$

Traverse 2 access links from AR_P

Set the access link of AR_R to the base of AR_S

$$\begin{aligned} l_{callee} &\le l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

- Traverse $n = l_{caller} - l_{callee} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                  //L_main = 1
{ void S()                  //L_S = 2
  { int x, z;
    void R()                //L_R = 3
    { int i;
      int T()        //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()                //L_E = 3
    { // body of E   }
    void Q()                //L_Q = 3
    { int a, x;
      int P(int y)  //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

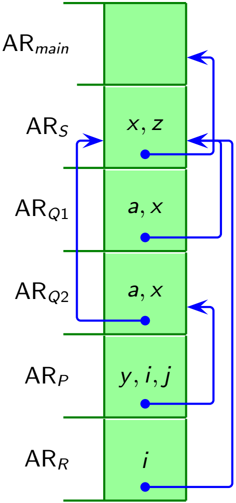
$l_{\text{caller}} = l_P = 4$
$l_{\text{callee}} = l_R = 3$
$n = 4 - 3 + 1 = 2$

Traverse 2 access links from AR_P

Set the access link of AR_R to the base of AR_S

$$\begin{aligned} l_{\text{callee}} &\leq l_{\text{caller}} + 1 \\ l_{\text{callee}} + n &= l_{\text{caller}} + 1 \\ n &= l_{\text{caller}} - l_{\text{callee}} + 1 \end{aligned}$$

- Traverse $n = l_{\text{caller}} - l_{\text{callee}} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                //Lmain = 1
{ void S()                //LS = 2
  { int x, z;
    void R()              //LR = 3
    { int i;
      int T()             //LT = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()              //LE = 3
    { // body of E   }
    void Q()              //LQ = 3
    { int a, x;
      int P(int y)        //LP = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

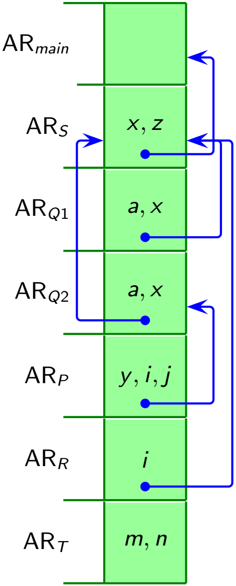
$l_{caller} = l_R = 3$
$l_{callee} = l_T = 4$
$n = 3 - 4 + 1 = 0$

Traverse 0 access links from AR_R

Set the access link of AR_T to the base of AR_R

$$\begin{aligned} l_{callee} &\leq l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

- Traverse $n = l_{caller} - l_{callee} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Setting the Static Link

```
int main()                //L_main = 1
{ void S()                //L_S = 2
  { int x, z;
    void R()              //L_R = 3
    { int i;
      int T()             //L_T = 4
      { int m,n;
        // body of T
      }
      T(); }
    void E()              //L_E = 3
    { // body of E   }
    void Q()              //L_Q = 3
    { int a, x;
      int P(int y)        //L_P = 4
      { int i,j;
        R();
      }
      if (.) Q(); else P(x); }
    Q();
  }
  S();
}
```

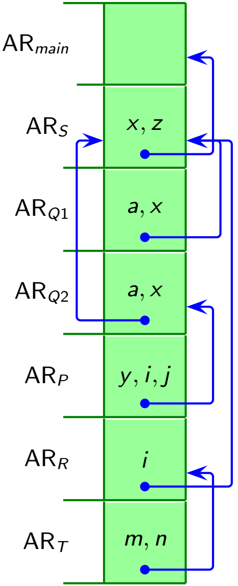
$l_{caller} = l_R = 3$
$l_{callee} = l_T = 4$
$n = 3 - 4 + 1 = 0$

Traverse 0 access links from AR_R

Set the access link of AR_T to the base of AR_R

$$\begin{aligned} l_{callee} &\leq l_{caller} + 1 \\ l_{callee} + n &= l_{caller} + 1 \\ n &= l_{caller} - l_{callee} + 1 \end{aligned}$$

- Traverse $n = l_{caller} - l_{callee} + 1$ access links from the caller to reach the base of activation record to which the callee's access link should point to



Parameter Passing Mechanisms

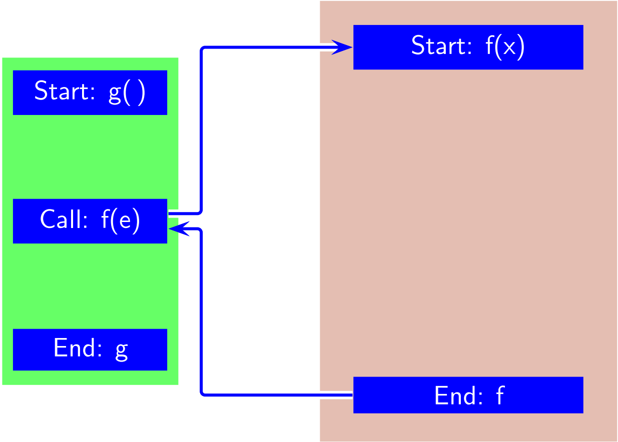
Parameter Passing Mechanisms

1. Call by value. Copy the value of the actual parameter into the formal parameter
2. Call by reference. Copy the address of the actual parameter into the formal parameter
3. Call by value-result (copy-restore). Copy the value of the actual parameter and copy the final value of the formal parameter into the actual parameter
4. Call by name. Textual substitution of formal parameter by the actual parameter

   Rename the local variables to avoid conflict between names
5. Call by need. Textual substitution of formal parameter by the actual parameter but evaluation only when needed and only once

   Rename the local variables to avoid conflict between names

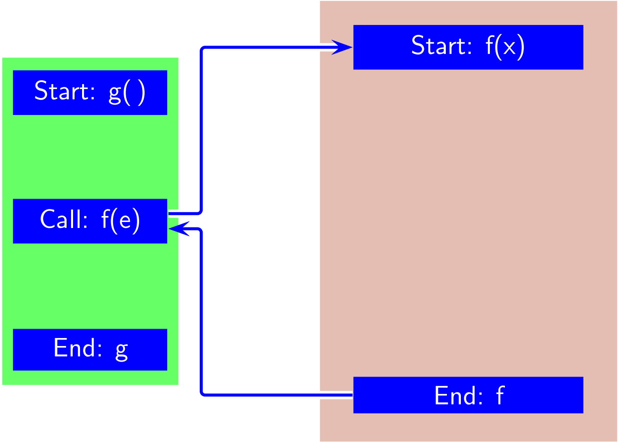
Parameter Passing Mechanisms for Variables as Parameters



- Function g is the caller and function f is the callee
- Variable x is the formal parameter of f
- Expression e is the actual parameter (e.g., $a + b$)

  In the simplest case, it could be a variable or a constant

Parameter Passing Mechanisms for Variables as Parameters



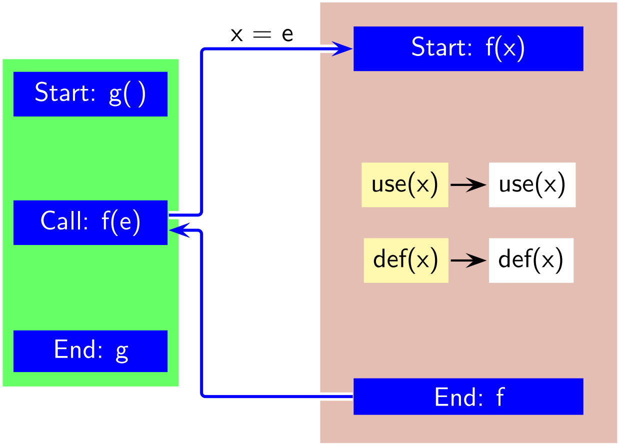
Notation

- $*x$ denotes a dereference of x where x is a pointer
- &x denotes the address of x
- &e denotes the address of the temporary variable in which e is evaluated before the call

  If e is a variable, say y, then &e is &y
- Reads and writes of x are denoted by use(x) and def(x), respectively

Parameter Passing Mechanisms for Variables as Parameters

Parameter passing by copy or call by value



- Expression e is evaluated just before the call and the result is copied into the location of x

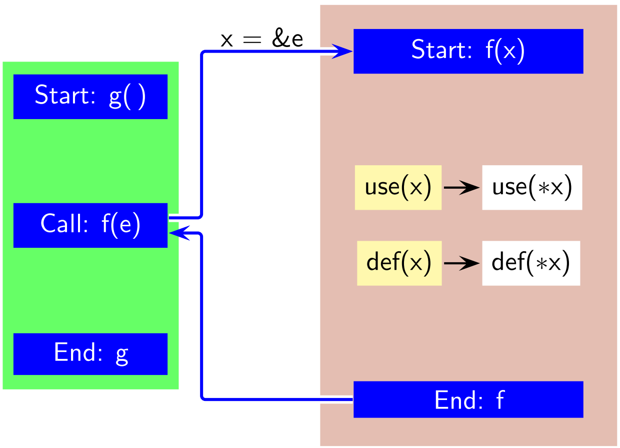
  Eager evaluation

- The reads and writes of x are performed on the location of x

  Reads are denoted by use(x) and writes are denoted by def(x)

Parameter Passing Mechanisms for Variables as Parameters

Parameter passing by reference or call by reference



- Expression e is evaluated just before the call and the address of e is copied into the location of x

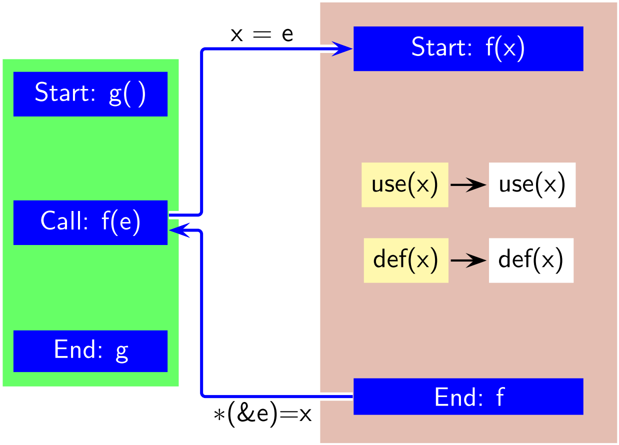
  Eager evaluation

  When e is a variable, the address of e is the address of the variable
- The reads and writes of x are performed by dereferencing x

  Reads are denoted by use(x) and writes are denoted by def(x)

Parameter Passing Mechanisms for Variables as Parameters

Parameter passing by value-result or call by copy-restore



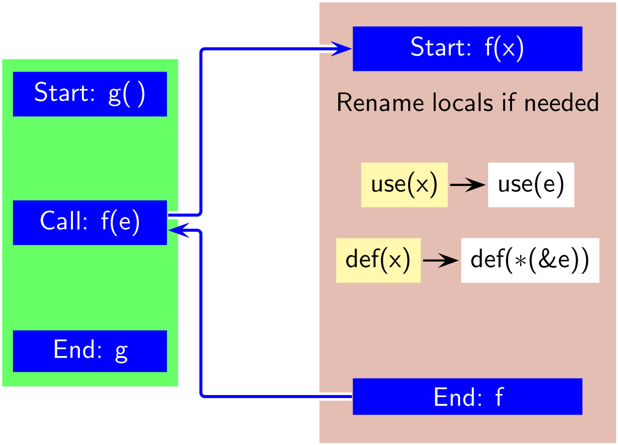
- Expression e is evaluated just before the call and the result is copied into the location of x

  Eager evaluation

- The reads and writes of x are performed on the location of x
- At the end of the call, the value in x is copied back into the location of expression e

Parameter Passing Mechanisms for Variables as Parameters

Parameter passing by name or call by name

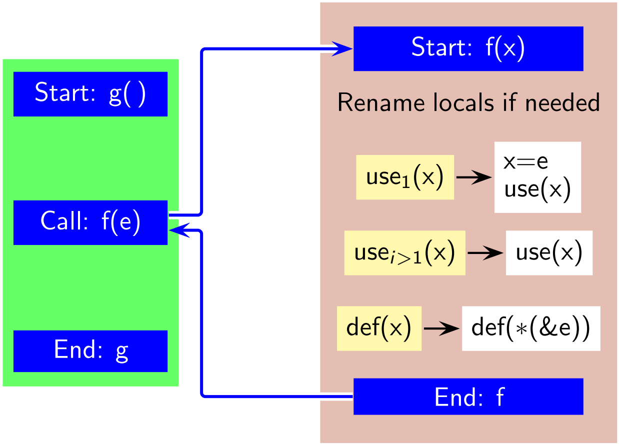


- Evaluation of expression e is delayed until the value of x is needed
- Expression e is evaluated afresh every time the value of x is needed

  Effectively, textual substitution like a macro
- A write to x writes into the address of e
- Implemented by a *thunk* which is a parameterless procedure per actual argument to evaluate the expression and return the address of the evaluation

Parameter Passing Mechanisms for Variables as Parameters

Parameter passing by need or call by need



- Evaluation of expression e is delayed until the value of x is needed
- Expression e is evaluated only once and the value is assigned to x

  Subsequent uses of x only read the value of x

  Lazy evaluation
- A write to x writes into the address of e

  Functional languages do not modify x

Special Cases

- Ada allows the following annotations for parameters
  - `in` for call by (const) value; this is the default
  - `out` for returning result
  - `in out` for call by value result
- C++ supports before call by value and call by references
  - Annotation & before a formal parameter indicates call by reference
    The default is call by value
  - Actual parameter for call by reference cannot be an expression
- C passes arrays by reference but structs and all other data types (including pointers) by value
  - Call by reference for other variables can be simulated by declaring formal parameters as pointer and passing addresses as actual parameters
- FORTRAN uses call by reference
- Java uses call by value

Distinguishing Between Different Parameter Passing Mechanisms

```
int a;
int main()
{
  a = 5;
  f(a,a+1);
  cout << a;
  return 0;
}

void f(x,y)
{
  a = a+10;
  a = a+y;
  x = x+y+100;
}
```

The value of variable a printed in `main` under different parameter passing mechanisms is

- For call by value: 21
- For call by reference: 127
- For call by value-result (copy-restore): 111
- For call by name: 163
- For call by need: 147

Distinguishing Between Different Parameter Passing Mechanisms

```
int a;
int main()
{
  a = 5;
  f(a,a+1);
  cout << a;
  return 0;
}

void f(x,y)
{
  a = a+10;
  a = a+y;
  x = x+y+100;
}
```

Program trace for call by value

```
a=5           ;a=5
t0=a+1        ;t0=6
x=a           ;x=5
y=t0          ;y=6
a=a+10        ;a=15
a=a+y         ;a=21
x=x+y+100     ;x=111
cout << a     ;21
```

Program trace for call by reference

```
a=5                   ;a=5
t0=a+1                ;t0=6
x=&a
y=&t0
a=a+10                ;a=15
a=a+(*y)=a+t0         ;a=21
*x=(*x)+(*y)+100      ;a=a+t0+100
                      ;a=127
cout << a             ;127
```

Distinguishing Between Different Parameter Passing Mechanisms


```
int a;
int main()
{
   a = 5;
   f(a,a+1);
   cout << a;
   return 0;
}

void f(x,y)
{
   a = a+10;
   a = a+y;
   x = x+y+100;
}
```

Program trace for call by value-result

```
a=5          ;a=5
t0=a+1       ;t0=6
x=a          ;x=5
y=t0         ;y=6
a=a+10       ;a=15
a=a+y        ;a=21
x=x+y+100    ;x=111
a=x          ;a=111
t0=y         ;t0=6
cout << a    ;111
```

Program trace for call by name

```
a=5            ;a=5
x≡a            ;replace every
               ;occurrence of
               ;x by a
y≡a+1          ;replace every
               ;occurrence of
               ;y by a+1
a=a+10         ;a=15
a=a+y=a+a+1    ;a=31
x=x+y+100      ;a=a+a+1+100
               ;a=163
cout << a      ;163
```

Distinguishing Between Different Parameter Passing Mechanisms

```
int a;
int main()
{
  a = 5;
  f(a,a+1);
  cout << a;
  return 0;
}

void f(x,y)
{
  a = a+10;
  a = a+y;
  x = x+y+100;
}
```

Program trace for call by need

```
a=5           ;a=5
x≡a           ;Evaluate a and copy in x
              ;on the first occurrence of x
y≡a+1         ;Evaluate a+1 and copy in y
              ;on the first occurrence of y
a=a+10        ;a=15
a=a+y         ;evaluate y=a+1=16
              ;a=a+y=31
x=x+y+100     ;evaluate x=a=31
              ;use y=16
              ;a=x+y+100=147
cout << a     ;147
```

Observations

- Call by value and call by reference may differ when the actual parameter is modified in the callee
- Call by reference and call by value result may differ when
  - a global variable is passed as an actual parameter, and
  - the global variable and the formal parameter are both modified in the callee
- Call by reference and call by name may differ when expressions are passed as actual parameters
- Call by name and call by need may differ when there are multiple uses of formal parameters

Parameter Passing Mechanisms for Procedures as Parameters

- Pass a closure of the procedure to be passed as a parameter

  A data structure containing a pair consisting of

  - A pointer to the procedure body
  - A pointer to the external environment (i.e. the declarations of the non-local variables visible in the procedure)

    Depends on the scope rules (i.e., static or dynamic scope)

Parameter Passing Mechanisms for Procedures as Parameters

- Pass a closure of the procedure to be passed as a parameter

  A data structure containing a pair consisting of

  - A pointer to the procedure body
  - A pointer to the external environment (i.e. the declarations of the non-local variables visible in the procedure)

    Depends on the scope rules (i.e., static or dynamic scope)

- For C, there are no nested procedures so the environment is trivially global

  A closure is represented trivially by a function pointer

Parameter Passing Mechanisms for Procedures as Parameters

- Pass a closure of the procedure to be passed as a parameter

  A data structure containing a pair consisting of

  - A pointer to the procedure body
  - A pointer to the external environment (i.e. the declarations of the non-local variables visible in the procedure)

    Depends on the scope rules (i.e., static or dynamic scope)

- For C, there are no nested procedures so the environment is trivially global

  A closure is represented trivially by a function pointer

- In C++, the environment of a class method consists of global declarations and the data members of the class

  The environment can be identified from the class name of the receiver object of the method call

Compiling Virtual Function Calls

Outline

- Internal representation of a class
- Translating virtual function calls
- Possible optimization

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};
```

Data Memory

Code Memory

Internal Representation of a Class

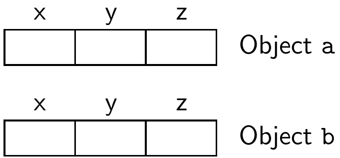
```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;
```

Data Memory

Code Memory

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;
```



Data Memory

Code Memory

```
void A::f1(struct A * const this , int i)
{ this->x = A::f2(this, i)*2;}

int A                              s, int i)
{ re

int m
```

There is no distinction between the public and private data in memory

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;
```

Every function with n parameters is converted to a function of $n+1$ parameter with the first parameter being the address of the object

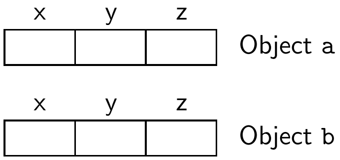
Data Memory

Code Memory

```
void A::f1(struct A * const this , int i)
{ this-> x  = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

 a.f1(5);
 b.f1(10);
```



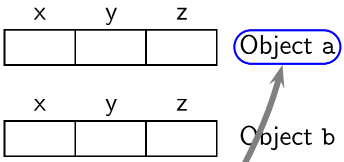
Data Memory

Code Memory

```
void A::f1(struct A * const this , int i)
{ this-> x  = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
  A::f1 ( &a , 5); A::f1 ( &b , 10);
}
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

a.f1(5);
b.f1(10);
```



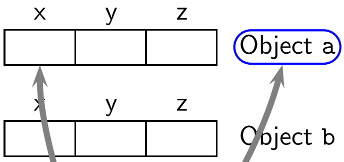
Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this-> x  = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
   A::f1 (&a, 5); A::f1 ( &b , 10);
}
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

a.f1(5);
b.f1(10);
```



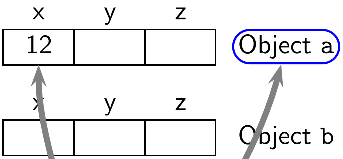
Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this->x = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
  A::f1 (&a, 5); A::f1 ( &b , 10);
}
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

a.f1(5);
b.f1(10);
```



Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this->x = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
   A::f1 (&a, 5); A::f1 ( &b , 10);
}
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

 a.f1(5);
 b.f1(10);
```

x	y	z	
12			Object a

x	y	z	
			Object b

Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this-> x  = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
  A::f1 (&a , 5); A::f1 (&b, 10);
}
```

Internal Representation of a Class


```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

a.f1(5);
b.f1(10);
```

x	y	z	
12			Object a

x	y	z	
			Object b

Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this->x = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
  A::f1 (&a , 5); A::f1 (&b, 10);
}
```

Internal Representation of a Class

```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

A a;
A b;

 a.f1(5);
 b.f1(10);
```

x	y	z	
12			Object a

x	y	z	
22			Object b

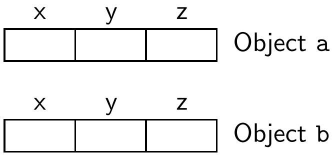
Data Memory

Code Memory

```
void A::f1(struct A * const this, int i)
{ this->x = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a,b;
  A::f1 (&a , 5); A::f1 (&b, 10);
}
```

Internal Representation of a Class

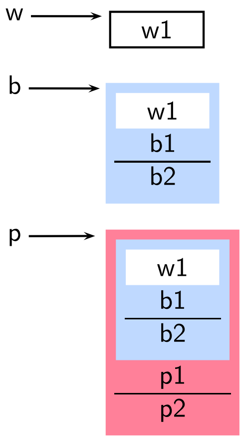
```
class A
{
  public:
    int y,z;
    void f1(int i)
      { x = f2(i)*2;}
  private:
    int x;
    int f2(int i)
      { return i+1;}
};

int main()
{ A a, b, *p;
  p=&a; p->f1(5);
  p=&b; p->f1(10);
}
```




```
void A::f1(struct A * const this, int i)
{ this->x = A::f2(this, i)*2;}

int A::f2(struct A * const this, int i)
{ return i+1;}

int main()
{ struct A a, b, *p;
  p=&a; A::f1(p, 5);
  p=&b; A::f1(p, 10);
}
```

Representing Inheritance of Data

```
class White
{ public:
     int  w1;
};
class Blue : public White
{ public:
      int b1;
      int b2;
};
class Pink : public Blue
{ public:
      int p1;
      int p2;
};
White w;
Blue b;
Pink p;
```



Representing Inheritance of Functions

- Non-virtual functions are inherited much like data members

  However, there is a single class-wide copy of the code and the address of the object is the first parameter

- Virtual functions create interesting possibilities based on the object to which a pointer points to

  A pointer to a base class object may point to an object of any derived class in the class hierarchy

An Example with Virtual Functions

```
class A
{  public:
     virtual void f()
     virtual void f(string i)
     virtual void g()
};

class B : public A
{  public:
     virtual void g()
             void f()
};

class C : public B
{  public:
             void f()
};
```

An Example with Virtual Functions

```
class A
{  public:
       virtual void f() {cout << "\tA:f" << endl;}
       virtual void f(string i) {cout << "\tA:f." << i << endl;}
       virtual void g() {cout << "\tA:g" << endl;}
};

class B : public A
{  public:
       virtual void g() {cout << "\tB:g" << endl;}
               void f() {cout << "\tB:f" << endl;}
};

class C : public B
{  public:
               void f() {cout<< "\tC:f" << endl;}
};
```

An Example with Virtual Functions

```
class A
{  public:
      virtual void f() {cout << "\tA:f" << endl;}
      virtual void f(string i) {cout << "\tA:f." << i << endl;}
      virtual void g() {cout << "\tA:g" << endl;}
};

class B : public A
{  public:
      virtual void g() {cout << "\tB:g" << endl;}
              void f() {cout << "\tB:f" << endl;}
};

class C : public B
{  public:
              void f() {cout<< "\tC:f" << endl;}
};
```

An Example with Virtual Functions

```
class A
{  public:
      vi
      vi                                    i << endl;}
      vi
};

class B
{  publ
      vi

};

class C
{  publ
          void f() {cout<< "\tC:f" << endl;}
};
```

If a function is declared as virtual in a class, it is considered virtual in the all its derived classes even if it not declared as virtual

If a function is declared as virtual in a derived class, it is not a virtual function in the base class unless it is explicitly declared to be virtual

Although `f()` is not declared virtual in class `B` and class `C`, it becomes virtual because it is virtual in class `A`

An Example with Virtual Functions

```
class A
{  public:
     virtual void f() {cout << "\tA:f" << endl;}
     virtual void f(string i) {cout << "\tA:f." << i << endl;}
     virtual void g() {cout << "\tA:g" << endl;}
};

class B : public A
{  public:
     virtual void g() {cout << "\tB:g" << endl;}
             void f() {cout << "\tB:f" << endl;}
};

class C : public B
{  public:
             void f() {cout<< "\tC:f" << endl;}
};
```

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output

`class A`
`{  public:`
`      virtual void f()`
`      virtual void f(string i)`
`      virtual void g()`
`};`
`class B : public A`
`{  public:`
`      virtual void g()`
`      void f()`
`};`
`class C : public B`
`{  public:`
`      void f()`
`};`
	`A a; B b; C c;`
	`A *p;`	

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output
	`A a; B b; C c;`	
`class A`	`A *p;`	
`{  public:`		
`virtual void f()`	`p = &a;`	
`virtual void f(string i)`	`p->f("classA");`	`A:f.classA`
`virtual void g()`	`p->f();`	`A:f`
`};`	`p->g();`	`A:g`
`class B : public A`		
`{  public:`		
`virtual void g()`		
`void f()`		
`};`		
`class C : public B`		
`{  public:`		
`void f()`		
`};`		

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output
	`A a; B b; C c;`	
`class A`	`A *p;`	
`{ public:`		
`virtual void f()`	`p = &a;`	
`virtual void f(string i)`	`p->f("classA");`	`A:f.classA`
`virtual void g()`	`p->f();`	`A:f`
`};`	`p->g();`	`A:g`
`class B : public A`		
`{ public:`	`p = &b;`	
`virtual void g()`	`p->f("classB");`	
`void f()`	`p->f();`	
`};`	`p->g();`	
`class C : public B`		
`{ public:`		
`void f()`		
`};`		

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output
	`A a; B b; C c;`	
`class A`	`A *p;`	
`{ public:`		
`virtual void f()`	`p = &a;`	
`virtual void f(string i)`	`p->f("classA");`	`A:f.classA`
`virtual void g()`	`p->f();`	`A:f`
`};`	`p->g();`	`A:g`
`class B : public A`		`A:f.classB`
`{ public:`	`p = &b;`	`B:f`
`virtual void g()`	`p->f("classB");`	`B:g`
`void f()`	`p->f();`	
`};`	`p->g();`	
`class C : public B`		
`{ public:`		
`void f()`		
`};`		

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output
	`A a; B b; C c;`	
`class A`	`A *p;`	
`{   public:`		
`        virtual void f()`	`p = &a;`	
`        virtual void f(string i)`	`p->f("classA");`	`A:f.classA`
`        virtual void g()`	`p->f();`	`A:f`
`};`	`p->g();`	`A:g`
`class B : public A`		`A:f.classB`
`{   public:`	`p = &b;`	`B:f`
`        virtual void g()`	`p->f("classB");`	`B:g`
`        void f()`	`p->f();`	
`};`	`p->g();`	
`class C : public B`		
`{   public:`	`p = &c;`	
`        void f()`	`p->f("classC");`	
`};`	`p->f();`	
	`p->g();`	

Examining the Behaviour of Virtual Functions (1)

Class Declarations	Calls	Output
	`A a; B b; C c;`	
`class A`	`A *p;`	
`{ public:`		
`virtual void f()`	`p = &a;`	
`virtual void f(string i)`	`p->f("classA");`	`A:f.classA`
`virtual void g()`	`p->f();`	`A:f`
`};`	`p->g();`	`A:g`
`class B : public A`		`A:f.classB`
`{ public:`	`p = &b;`	`B:f`
`virtual void g()`	`p->f("classB");`	`B:g`
`void f()`	`p->f();`	`A:f.classC`
`};`	`p->g();`	`C:f`
`class C : public B`		`B:g`
`{ public:`	`p = &c;`	
`void f()`	`p->f("classC");`	
`};`	`p->f();`	
	`p->g();`	

Examining the Behaviour of Virtual Functions (2)

Class declarations	Output with virtual functions	Output with no virtual functions

`class A`
`{ public:`
`virtual void f()`
`virtual void f(string i)`
`virtual void g()`
`};`
`class B : public A`
`{ public:`
`virtual void g()`
`void f()`
`};`
`class C : public B`
`{ public:`
`void f()`
`};`
`A a; B b; C c; A *p;`
	A:f.classA
	A:f
	A:g
	A:f.classB
	B:f
	B:g
	A:f.classC
	C:f
	B:g
		A:f.classA
		A:f
		A:g
		A:f.classB
		A:f
		A:g
		A:f.classC
		A:f
		A:g

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Runtime Support

Section:

Introduction

Compiling Procedure Calls

Parameter Passing Mechanisms

Compiling Virtual Function Calls

Examining the Behaviour of Virtual Functions (3)

Class declarations	Using the classes

`class A`
`{ public:`
`    virtual void f()`
`    virtual void f(string i)`
`    virtual void g()`
`};`
`class B : public A`
`{ public:`
`    virtual void g()`
`    void f()`
`    virtual void h()`
`};`
	`A a;`
	`B b;`
	`A *p;`
	`...`
	`p->h();`
	
	The compiler gives the following error regardless of the pointee of p
	`'class A' has no member named 'h'`
	
	Declaration `A *p;` is sufficient to conclude this.

Virtual Function Resolution

- Partially static and partially dynamic activity
- At compile time, a compiler creates a virtual function table for each class
- Every object contains a pointer to the virtual function table of the class of the object
- At run time, a pointer may point to an object of any derived class
- Compiler generates code to pick up the appropriate function by
  - accessing the virtual function table of the class of the object
  - accessing the function pointer stored in the entry for the function in the table

  The function is then invoked using the function pointer

Generating Code for Virtual Function Calls

Class declarations	Code	Translated Code
`class A`	`A a, *p;`	`A a, *p;`
`{ public:`	`B b; C c;`	`B b; C c;`
`virtual void f()`		
`virtual void f(string i)`	`p = &a;`	`p = &a;`
`virtual void g()`	`p->f("AA");`	`(*((p->_vptr)[1]))(p,"AA");`
`};`	`p->f();`	`(*((p->_vptr)[0]))(p);`
`class B : public A`	`p->g();`	`(*((p->_vptr)[2]))(p);`
`{ public:`	`p = &b;`	`p = &b;`
`virtual void g()`	`p->f("BB");`	`(*((p->_vptr)[1]))(p,"BB");`
`void f()`	`p->f();`	`(*((p->_vptr)[0]))(p);`
`};`	`p->g();`	`(*((p->_vptr)[2]))(p);`
`class C : public B`	`p = &c;`	`p = &c;`
`{ public:`	`p->f("CC");`	`(*((p->_vptr)[1]))(p,"CC");`
`void f()`	`p->f();`	`(*((p->_vptr)[0]))(p);`
`};`	`p->g();`	`(*((p->_vptr)[2]))(p);`

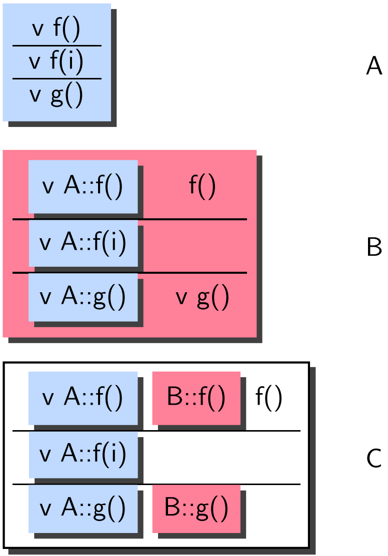
Generating Code for Virtual Function Calls

Class declarations	Code	Translated Code
`class A`	`A a, *p;`	`A a, *p;`
`{ public:`	`B b; C c;`	`B b; C c;`
`virtual void f()`		
`virtual void f(string i)`	`p = &a;`	`p = &a;`
`virtual void g()`	`p->f("AA");`	`(*((p->_vptr)[1]))(p,"AA");`
`};`	`p->f();`	`(*((p->_vptr)[0]))(p);`
`class B : public A`	`p->g();`	`(*((p->_vptr)[2]))(p);`
`{ publi`		
`virt`		`p,"BB");`
`void`		`p);`
`};`		`p);`
`class C`		
`{ publi`		`p,"CC");`
`void`		`p);`
`};`		`p);`

- Offset 0 in the virtual table contains a pointer to function `f()` because it is the first function in the class
- Offset 1 contains a pointer to function `f(string i)` because it is the second function in the class
- Offset 2 contains a pointer to function `g()` because it is the third function in the class

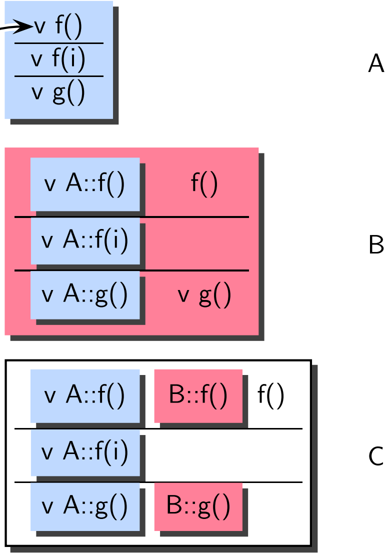
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();
```



Virtual Function Resolution Requires Dynamic Information

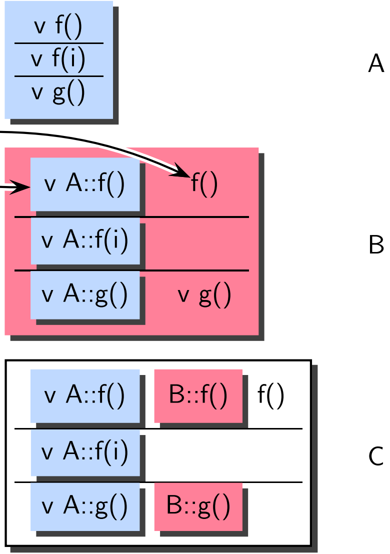
```
A *p;

p = ...
p->f();
```



In general, at compile time we do not know the class of the pointee of p which may be assigned in some other procedure

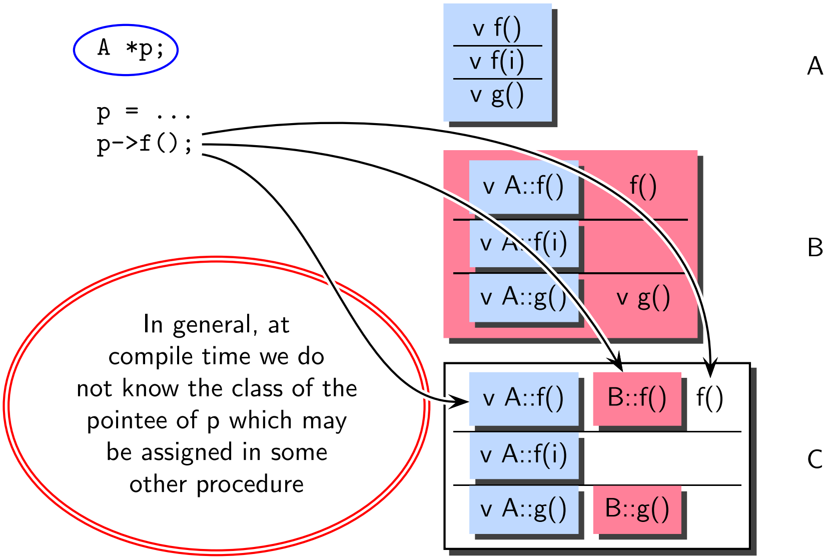
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();
```



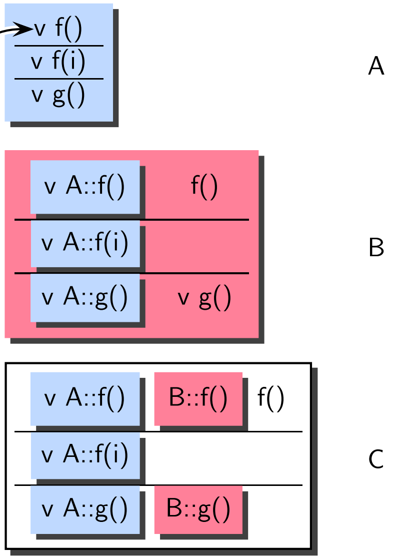
In general, at compile time we do not know the class of the pointee of p which may be assigned in some other procedure

Virtual Function Resolution Requires Dynamic Information



Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();
```



Virtual Function Resolution Requires Dynamic Information

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Runtime Support

Section:

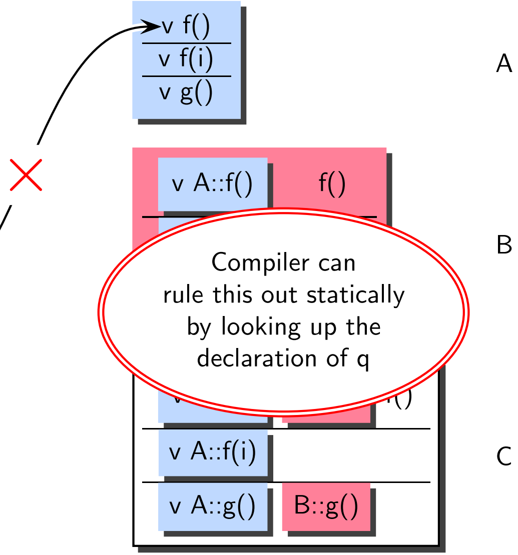
Introduction

Compiling Procedure Calls

Parameter Passing Mechanisms

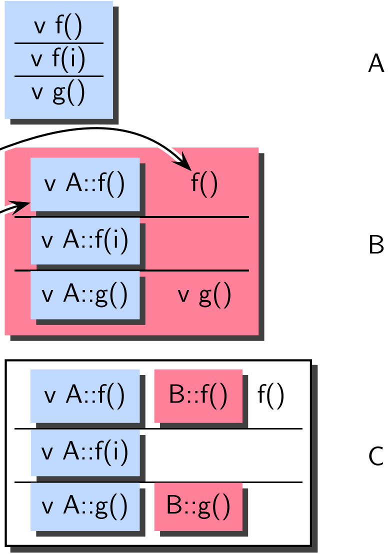
Compiling Virtual Function Calls

```
A *p;

p = ...
p->f();

B *q;

q = ...
q->f();
```



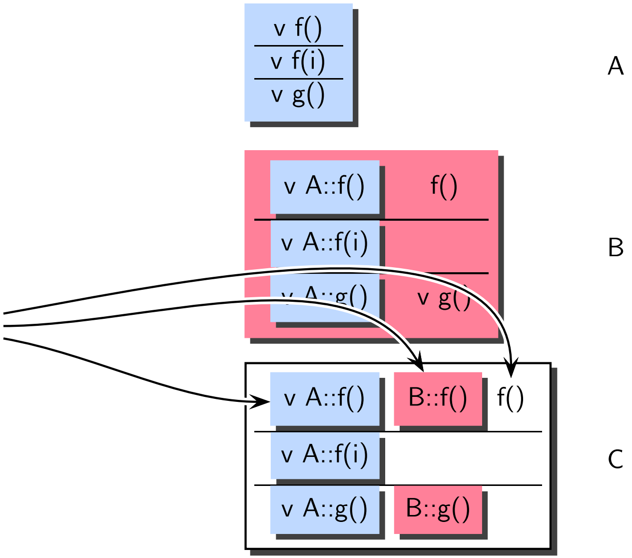
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();

B *q;

q = ...
q->f();
```



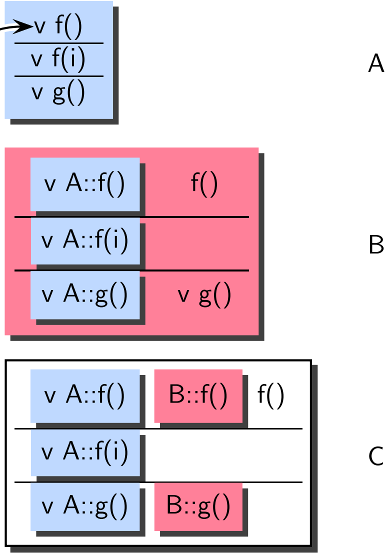
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();
```



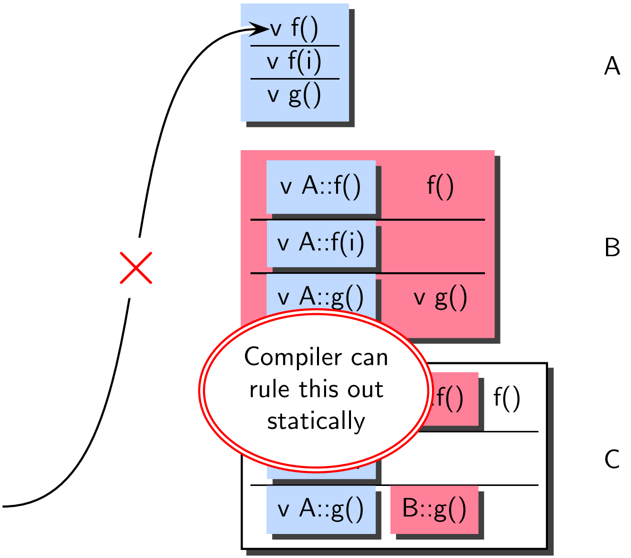
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();


C *r;

r = ...
r->f();
```



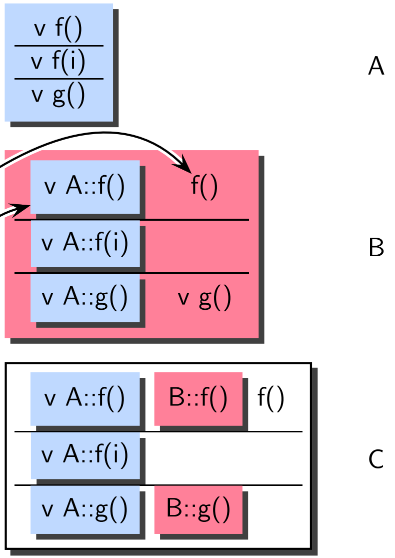
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();


C *r;

r = ...
r->f();
```



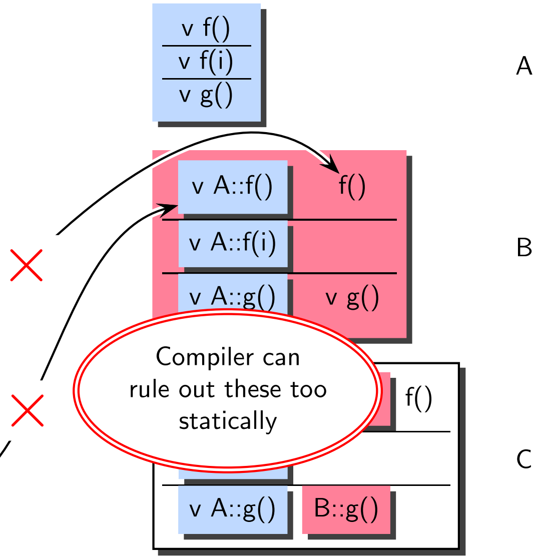
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();


C *r;

r = ...
r->f();
```



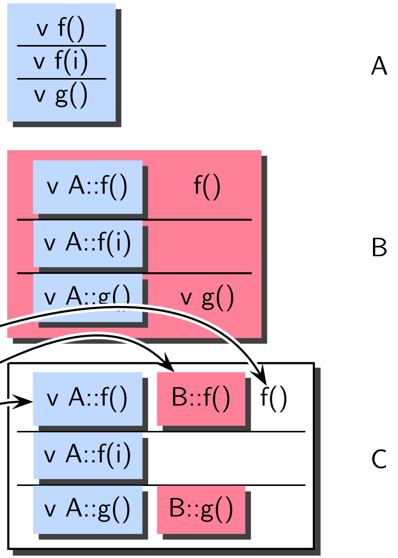
Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();


C *r;

r = ...
r->f();
```



Virtual Function Resolution Requires Dynamic Information

```
A *p;

p = ...
p->f();


B *q;

q = ...
q->f();


C *r;

r = ...
r->f();
```



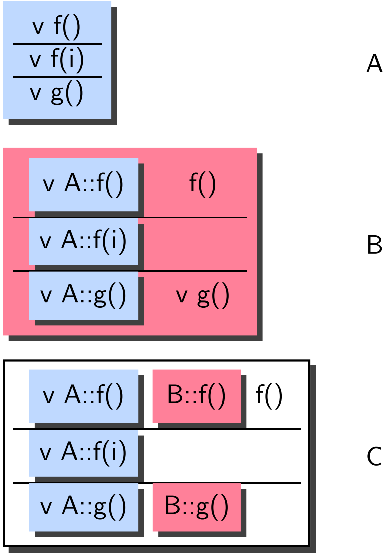
Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```



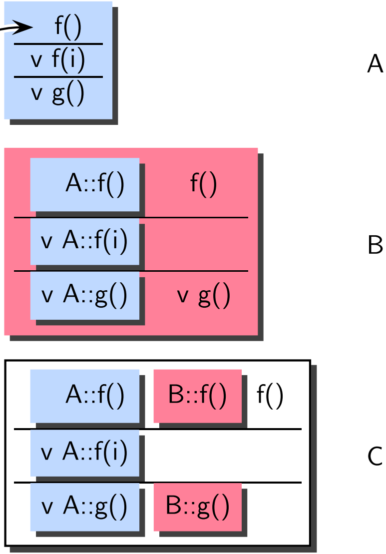
Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...

r->f();
```

In general, at compile time we do not know the class of the pointee of p



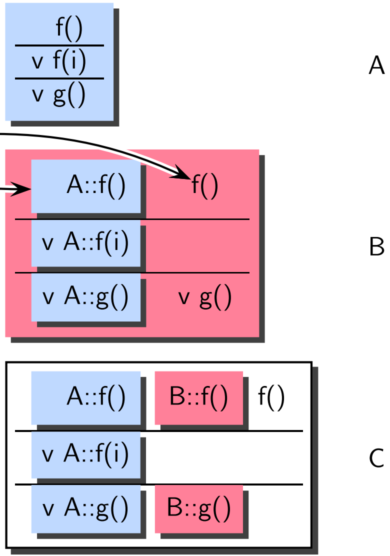
Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```

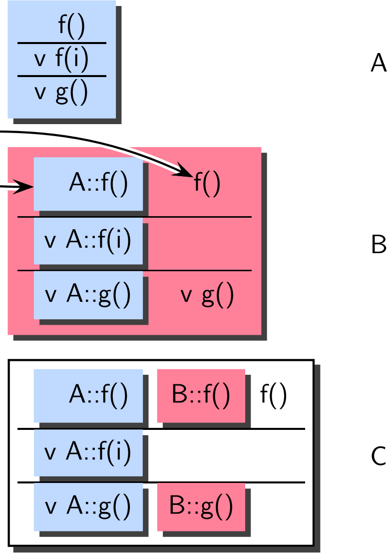


Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q =
```



Even if the pointee is an object of class B, `p->f()` must be `A::f()` because p is a pointer to class A

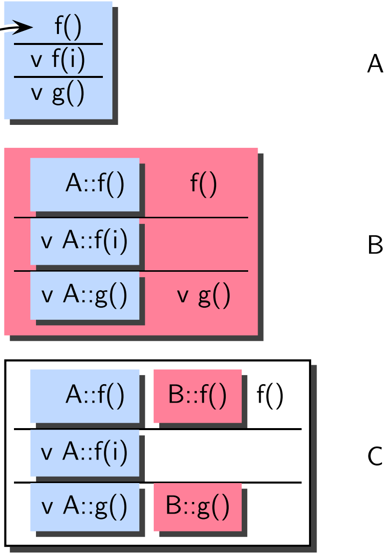
Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q =
```

Even if the pointee is an object of class B, `p->f()` must be `A::f()` because p is a pointer to class A



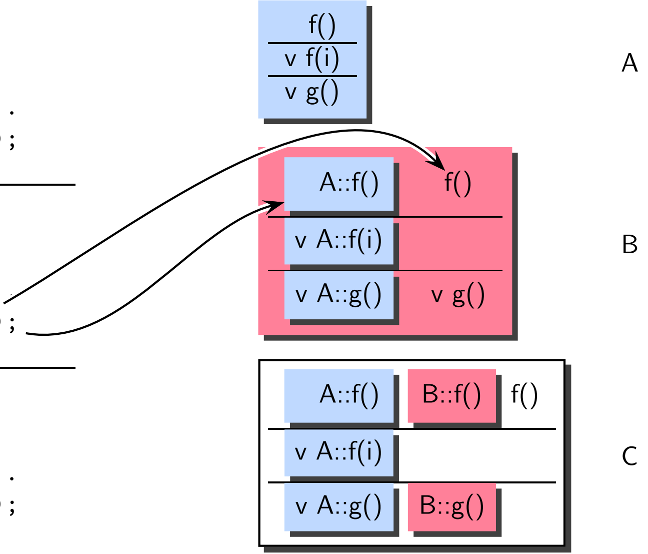
Non-Virtual Functions Do Not Require Dynamic Information

Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```



Non-Virtual Functions Do Not Require Dynamic Information

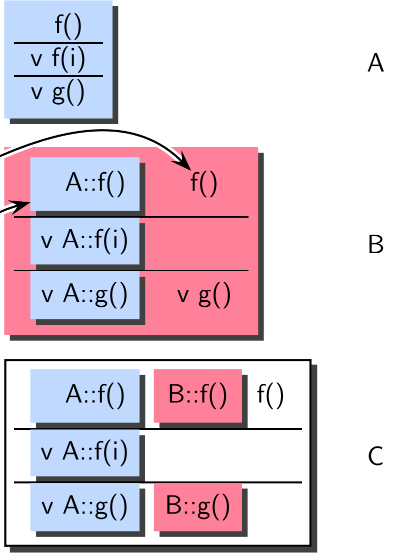
Non-virtual function = a function which is not virtual in *any* class in a hierarchy

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C
r
r->f();
```

q is a pointer to an object of class B



A Summary of Function Call Resolution

- Resolution of virtual functions depends on the class of the pointee object
  $\Rightarrow$ Needs dynamic information
- Resolution of non-virtual functions depends on the class of the pointer
  $\Rightarrow$ Compile time information is sufficient
- In either case, a pointee cannot belong to a "higher" class in the hierarchy
  ("higher" class is a class from which the class of the pointer is derived)

Constructing Virtual Function Table (1)

```
class A
{ public:
   virtual void f()
   virtual void f(string i)
   virtual void g()
};

class B : public A
{ public:
   virtual void g()
           void f()
};

class C : public B
{ public:
           void f()
};
```

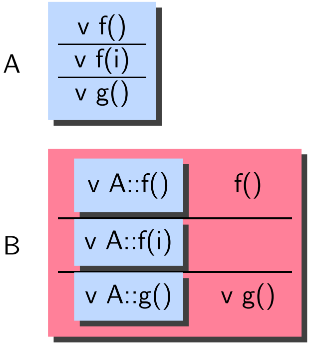
Constructing Virtual Function Table (1)

```
class A
{ public:
   virtual void f()
   virtual void f(string i)
   virtual void g()
};

class B : public A
{ public:
   virtual void g()
           void f()
};

class C : public B
{ public:
           void f()
};
```

A	
	v f()
	v f(i)
	v g()

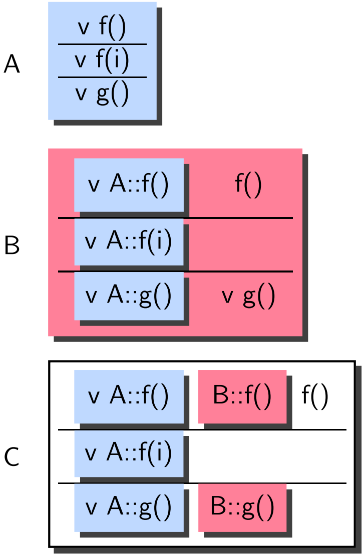
Constructing Virtual Function Table (1)

```
class A
{ public:
    virtual void f()
    virtual void f(string i)
    virtual void g()
};

class B : public A
{ public:
    virtual void g()
            void f()
};

class C : public B
{ public:
            void f()
};
```

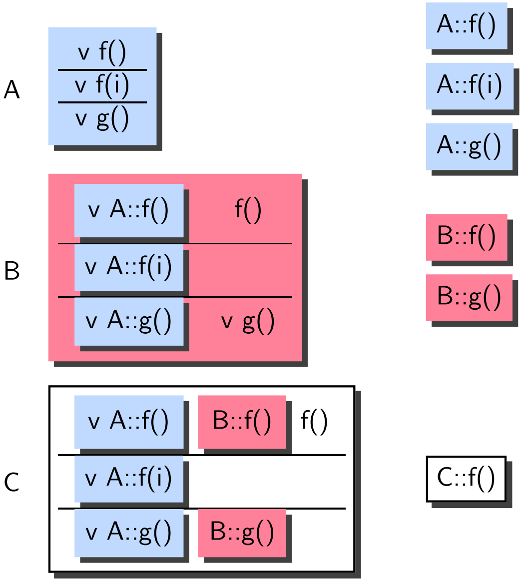


Constructing Virtual Function Table (1)

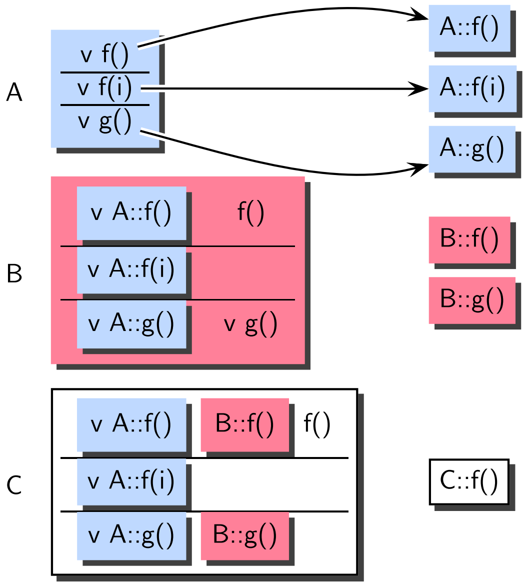
```
class A
{ public:
    virtual void f()
    virtual void f(string i)
    virtual void g()
};

class B : public A
{ public:
    virtual void g()
            void f()
};

class C : public B
{ public:
            void f()
};
```



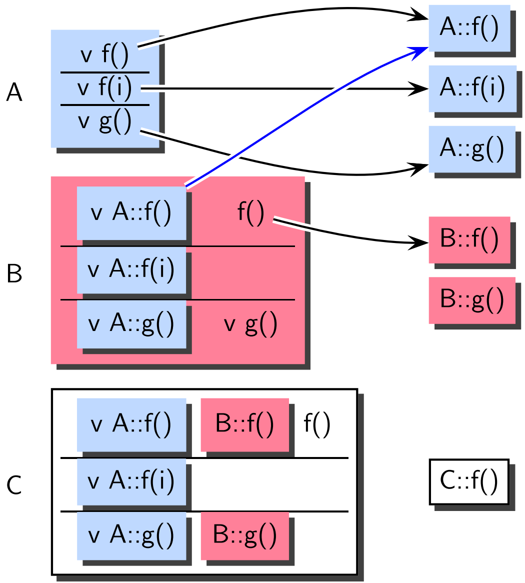
Constructing Virtual Function Table (2)



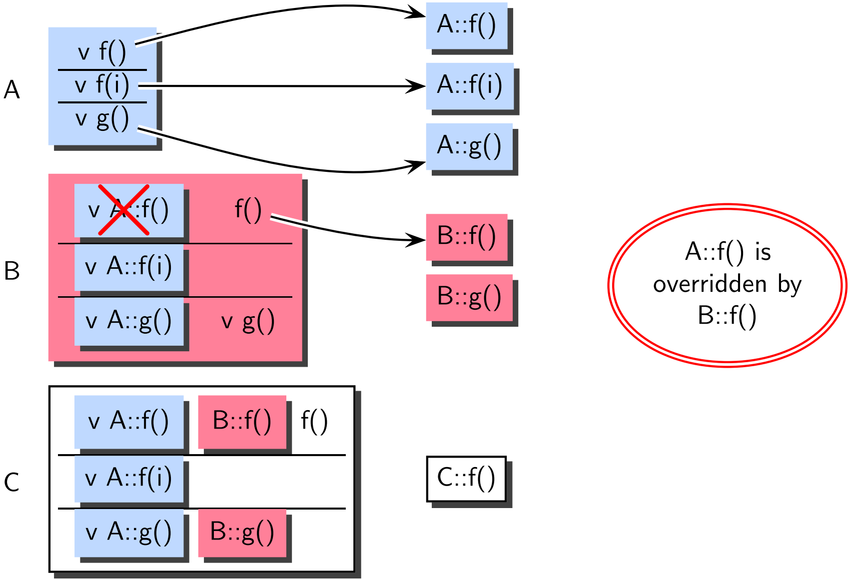
Constructing Virtual Function Table (2)



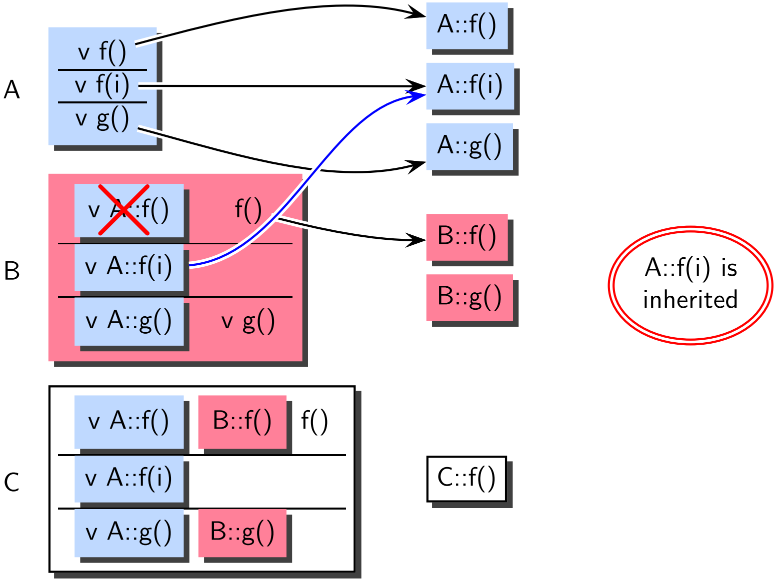
Constructing Virtual Function Table (2)



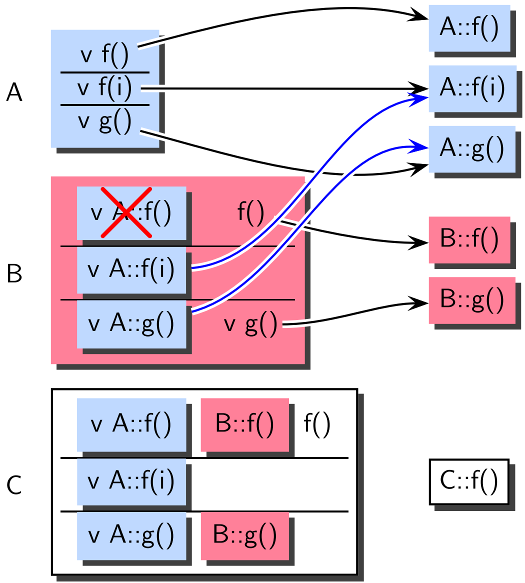
Constructing Virtual Function Table (2)



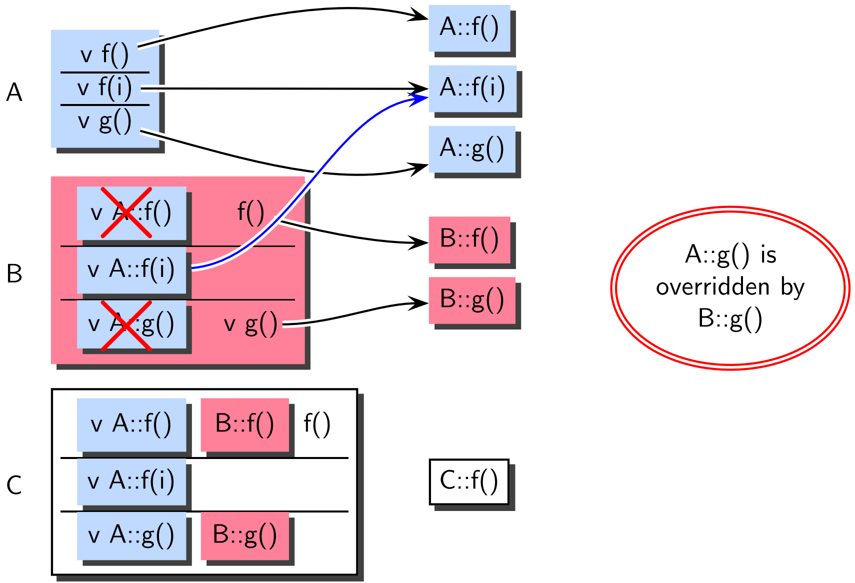
Constructing Virtual Function Table (2)



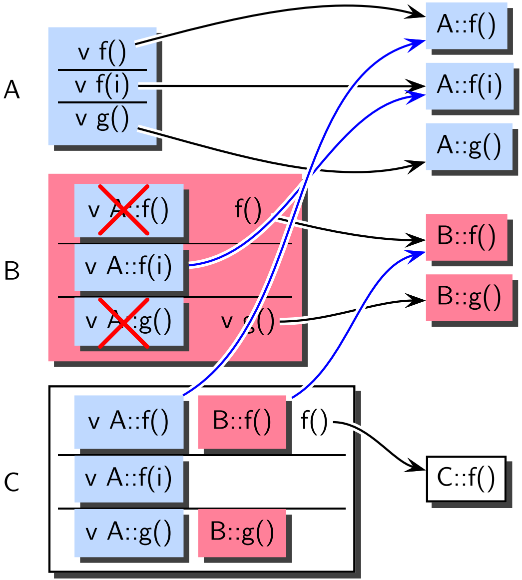
Constructing Virtual Function Table (2)



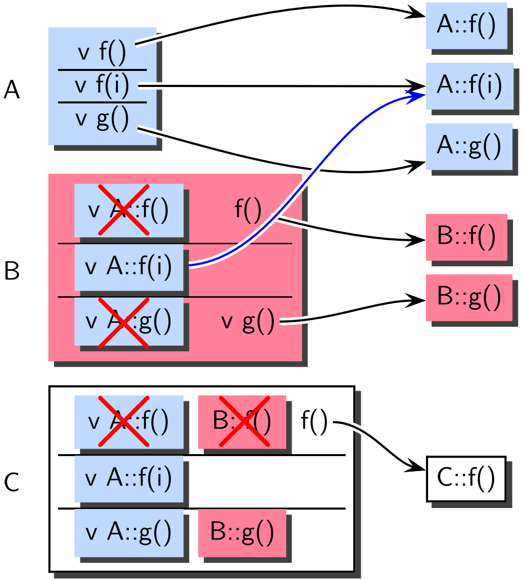
Constructing Virtual Function Table (2)



Constructing Virtual Function Table (2)

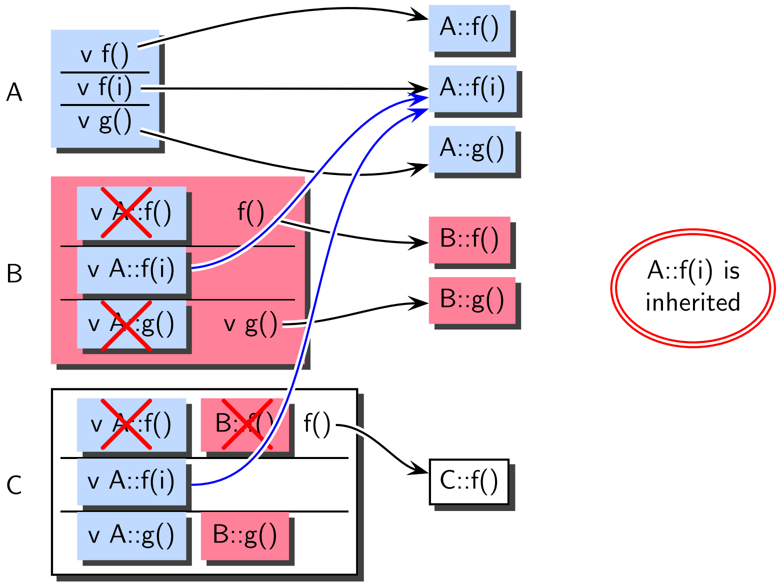


Constructing Virtual Function Table (2)

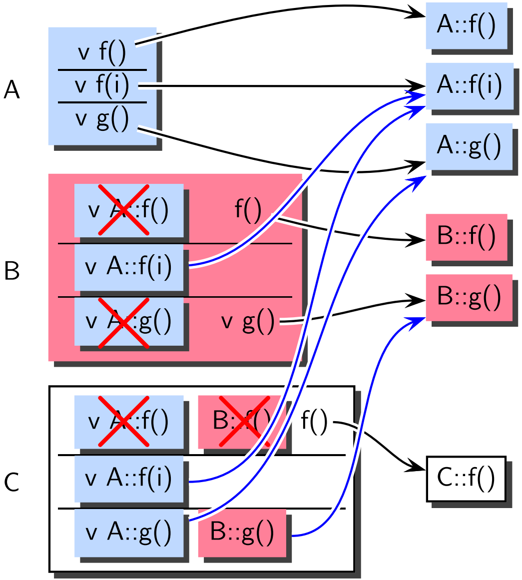


Both A::f() and B::f() are overridden by C::f()

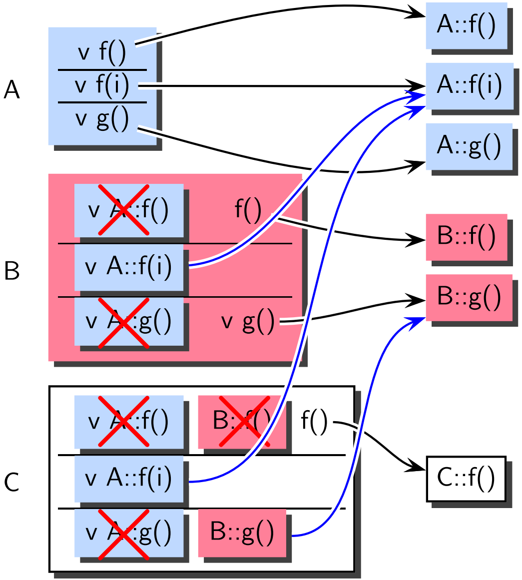
Constructing Virtual Function Table (2)



Constructing Virtual Function Table (2)

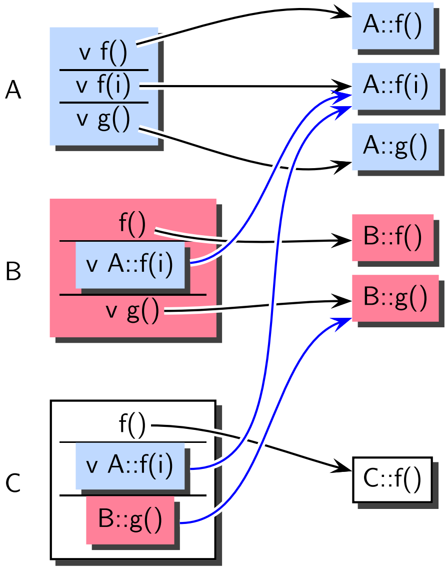


Constructing Virtual Function Table (2)



A::g() is overridden by B::g() which is inherited

Constructing Virtual Function Table (2)



IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Runtime Support

Section:

Introduction

Compiling Procedure Calls

Parameter Passing Mechanisms

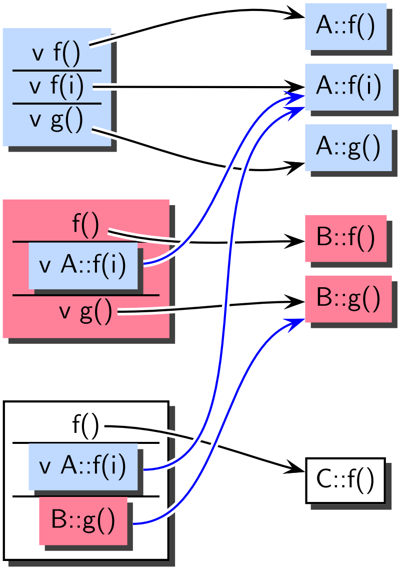
Compiling Virtual Function Calls

Using the Constructed Virtual Function Table

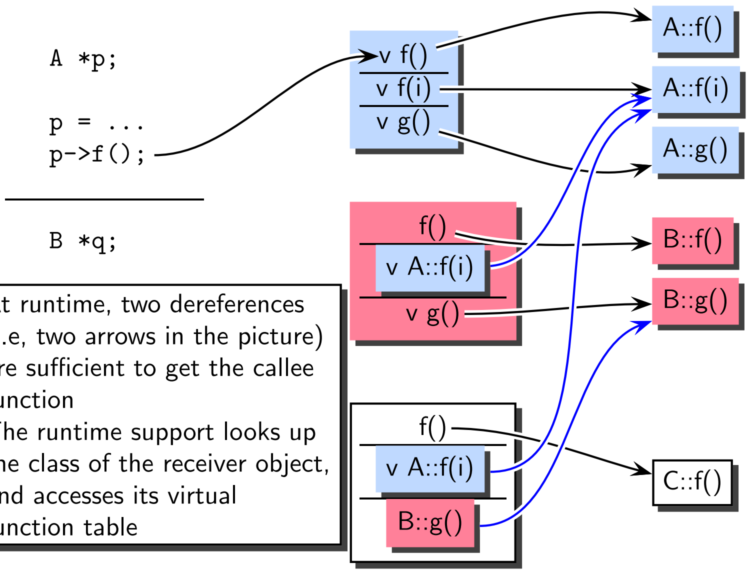
```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```



Using the Constructed Virtual Function Table

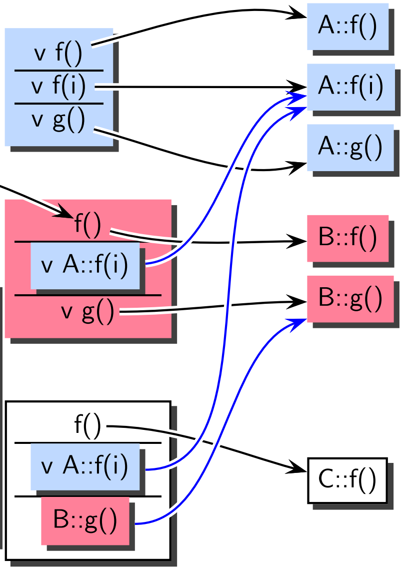


At runtime, two dereferences (i.e, two arrows in the picture) are sufficient to get the callee function
The runtime support looks up the class of the receiver object, and accesses its virtual function table

Using the Constructed Virtual Function Table

```
A *p;

p = ...
p->f();
```

```
B *q;
```



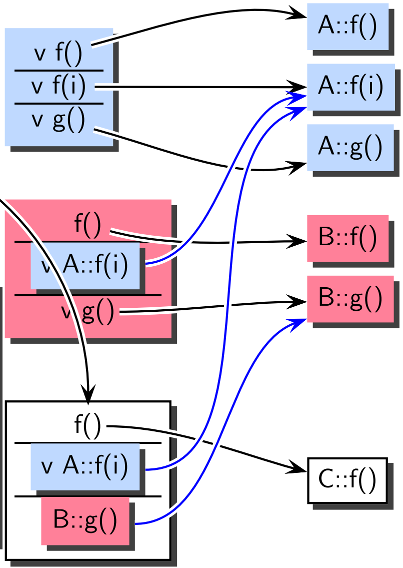
At runtime, two dereferences (i.e, two arrows in the picture) are sufficient to get the callee function
The runtime support looks up the class of the receiver object, and accesses its virtual function table

Using the Constructed Virtual Function Table

```
A *p;

p = ...
p->f();
```

```
B *q;
```

At runtime, two dereferences (i.e, two arrows in the picture) are sufficient to get the callee function
The runtime support looks up the class of the receiver object, and accesses its virtual function table

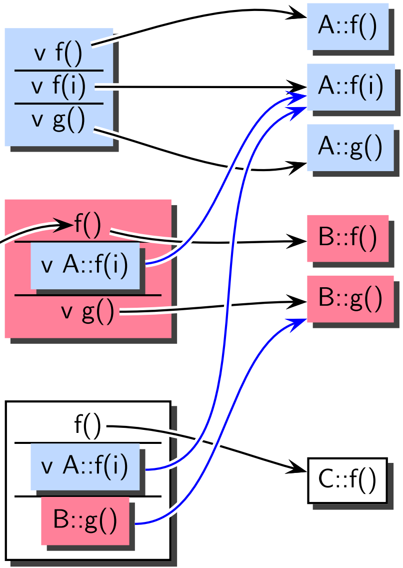


Using the Constructed Virtual Function Table

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```

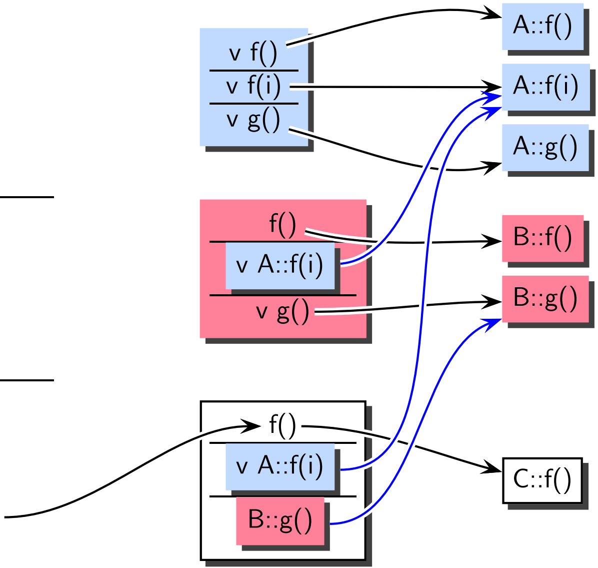


Using the Constructed Virtual Function Table

```
A *p;

p = ...
p->f();
```

```
B *q;

q = ...
q->f();
```

```
C *r;

r = ...
r->f();
```



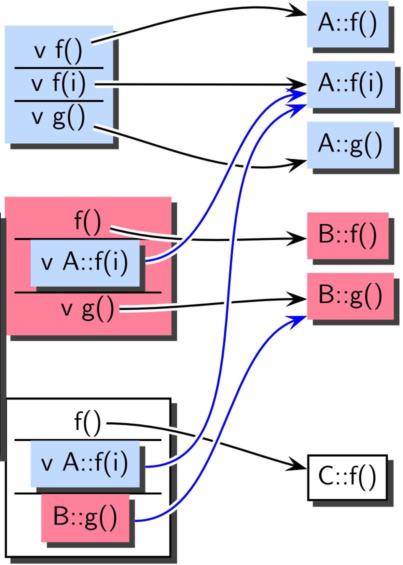
Using the Constructed Virtual Function Table

```
A *p;

p = ...
p->f();
```

This is possible because the size of the virtual function table is made same for all classes in the hierarchy

This is enabled by the fact that there can be only one function with a given name and permutation of types of the arguments

```
r = ...
r->f();
```



A Summary of Virtual Function Implementation

- Compile time activity
  - Collect all virtual functions across a class hierarchy
  - Ignore non-virtual functions
  - Analyze the class hierarchy to locate the appropriate function with a given permutation of argument types

A Summary of Virtual Function Implementation

- Compile time activity
  - Collect all virtual functions across a class hierarchy
  - Ignore non-virtual functions
  - Analyze the class hierarchy to locate the appropriate function with a given permutation of argument types
- Execution time activity
  - Dereference object pointer to know the class and access the virtual function table
  - Add offset to the base of the table to access the function pointer
  - Dereference the function pointer to make the call

Optimizing Virtual Function Calls

Source Code	Translated Code	Optimized Code
`A a; B b; C c;`	`A a; B b; C c;`	
`A*p;`	`A*p;`	
`p = &a;`	`p = &a;`	
`p->f("AA");`	`(*((p->_vptr)[1]))(p, "AA");`	
`p->f();`	`(*((p->_vptr)[0]))(p);`	
`p->g();`	`(*((p->_vptr)[2]))(p);`	
`p = &b;`	`p = &b;`	
`p->f("BB");`	`(*((p->_vptr)[1]))(p,"BB");`	
`p->f();`	`(*((p->_vptr)[0]))(p);`	
`p->g();`	`(*((p->_vptr)[2]))(p);`	
`p = &c;`	`p = &c;`	
`p->f("CC");`	`(*((p->_vptr)[1]))(p,"CC");`	
`p->f();`	`(*((p->_vptr)[0]))(p);`	
`p->g();`	`(*((p->_vptr)[2]))(p);`	

Optimizing Virtual Function Calls

Source Code	Translated Code	Optimized Code
`A a; B b; C c;` `A*p;`	`A a; B b; C c;` `A*p;`	

`p = &a;`
`p->f("A`
`p->f();`
`p->g();`		

`p = &b;`
`p->f("B`
`p->f();`
`p->g();`		

`p = &c;`
`p->f("CC");`
`p->f();`
`p->g();`
	`(*((p->_vptr)[1]))(p,"CC");`
	`(*((p->_vptr)[0]))(p);`
	`(*((p->_vptr)[2]))(p);`	

- Each function call involves a pointer indirection and an array indirection

  Runtime overheads

- No function call can be resolved at link time

  Call graph is not known and hence interprocedural optimizations are prohibited

Optimizing Virtual Function Calls

Source Code	Translated Code	Optimized Code
`A a; B b; C c;`	`A a; B b; C c;`	`A a; B b; C c;`
`A*p;`	`A*p;`	`A*p;`
`p = &a;`	`p = &a;`	`p = &a;`
`p->f("AA");`	`(*((p->_vptr)[1]))(p, "AA");`	`A::f(&a, "AA");`
`p->f();`	`(*((p->_vptr)[0]))(p);`	`A::f(&a);`
`p->g();`	`(*((p->_vptr)[2]))(p);`	`A::g(&a);`
`p = &b;`	`p = &b;`	`p = &b;`
`p->f("BB");`	`(*((p->_vptr)[1]))(p,"BB");`	`A::f(&b,"BB");`
`p->f();`	`(*((p->_vptr)[0]))(p);`	`B::f(&b);`
`p->g();`	`(*((p->_vptr)[2]))(p);`	`B::g(&b);`
`p = &c;`	`p = &c;`	`p = &c;`
`p->f("CC");`	`(*((p->_vptr)[1]))(p,"CC");`	`A::f(&b,"CC");`
`p->f();`	`(*((p->_vptr)[0]))(p);`	`C::f(&c);`
`p->g();`	`(*((p->_vptr)[2]))(p);`	`B::g(&c);`