Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Module 04: CS-1319-1: Programming Language Design and Implementation (PLDI)

Run-time Environments

Partha Pratim Das

Department of Computer Science
Ashoka University

ppd@ashoka.edu.in, partha.das@ashoka.edu.in, 9830030880

September 18, 19 & 25, 2023

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Module Objectives

- Understand the Run-Time Environment for Program Execution
- Understand Symbol Tables, Activation Records (Stack Frames) and interrelationships
- Understand Binding, Layout and Scopes

Module Outline

1. Objectives & Outline
2. Memory
3. Function Call
   - Symbol Table
   - Activation Record
   - x86 Assembly
   - Debug Build
   - Release Build
   - Decode ASM
4. Properties
   - Declaration, Definition, and Initialization
   - Scope & Binding
   - Storage Class
   - Address & Value
   - Object Lifetime
   - Param & Return
5. Array
6. Scopes
   - Nested Blocks
   - Global / Static
7. Summary

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Storage Organization

Dragon Book: Pages 427-430 (Storage Organization)

Storage Organization

Typical sub-division of run-time memory into code and data areas with the corresponding bindings

Memory Segment		**Bound Items**
Text		*Program Code*
Const		*Program Constants*
Static		*Global & Non-Local Static*
Heap		*Dynamic*
... Heap grows downwards here ... ... **Free Memory** ... Stack grows upwards here		
Stack		*Automatic*

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Function Call in Execution

Symbol Table ⇒ Activation Record

Dragon Book: Pages 430-438 (Stack Allocation of Space)
Dragon Book: Pages 363-369 (Three Address Codes)
Dragon Book: Pages 33-35 (Parameter Passing Mechanism)
Evaluation strategy, Wikipedia
Examples by PPD

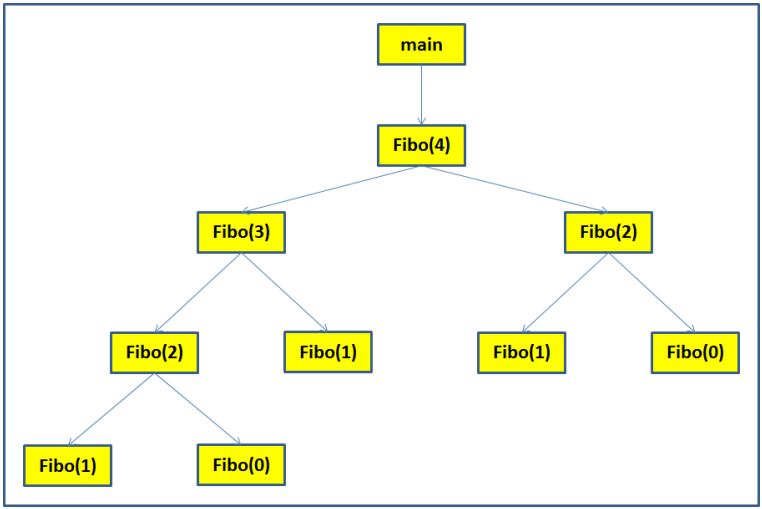
Fibo

```
int fibo(int n)
{
    if (n < 2)
        return n;
    else
        return
            fibo(n-1)+
            fibo(n-2);
}

int main()
{
    int m = 10;
    int f = 0;

    f = fibo(m);

    return 0;
}
```

```
fibo:   t1 = 2
        if (n < t1) goto L100
        goto L101
L100:   return n
        goto L102
L101:   t2 = 1
        t3 = n - t2
        param t3
        t4 = call fibo,1
        t5 = 2
        t6 = n - t5
        param t6
        t7 = call fibo, 1
        t8 = t4 + t7
        return t8
        goto L102
L102:   goto L102

main:   param m
        t1 = call fibo, 1;
        f = t1;
```

Activation Tree / Call Graph – Fibo

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary



Activation Records in Action on Stack – Fibo

AR of main()	
Prm	
RV	...
Lnk	crtmain()

AR of fibo(4)	
Prm	4
RV	...
Lnk	main()

AR of fibo(3)	
Prm	3
RV	...
Lnk	fibo(4)

AR of fibo(2)	
Prm	2
RV	...
Lnk	fibo(3)

AR of fibo(1)	
Prm	1
RV	...
Lnk	fibo(2)

AR of main()	
Prm	
RV	...
Lnk	crtmain()

AR of fibo(4)	
Prm	4
RV	...
Lnk	main()

AR of fibo(3)	
Prm	3
RV	...
Lnk	fibo(4)

AR of fibo(2)	
Prm	2
RV	...
Lnk	fibo(3)

AR of fibo(0)	
Prm	0
RV	...
Lnk	fibo(2)

AR of main()	
Prm	
RV	...
Lnk	crtmain()

AR of fibo(4)	
Prm	4
RV	...
Lnk	main()

AR of fibo(3)	
Prm	3
RV	...
Lnk	fibo(4)

AR of fibo(1)	
Prm	1
RV	...
Lnk	fibo(3)

AR of main()	
Prm	
RV	...
Lnk	crtmain()

AR of fibo(4)	
Prm	4
RV	...
Lnk	main()

AR of fibo(2)	
Prm	2
RV	...
Lnk	fibo(4)

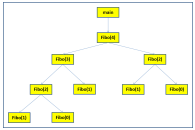
AR of fibo(1)	
Prm	1
RV	...
Lnk	fibo(2)

AR of main()	
Prm	
RV	...
Lnk	crtmain()

AR of fibo(4)	
Prm	4
RV	...
Lnk	main()

AR of fibo(2)	
Prm	2
RV	...
Lnk	fibo(4)

AR of fibo(0)	
Prm	0
RV	...
Lnk	fibo(2)



Example: main() & add(): Source & TAC

```
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main(int argc, char* argv[]) {
    int a, b, c;
    a = 2;
    b = 3;
    c = add(a, b);
    return;
}
```

```
add:    t1 = x + y
        z = t1
        return z
main:   t1 = 2
        a = t1
        t2 = 3
        b = t2
        param a
        param b
        c = call add, 2
        return
```

ST.glb				*Parent = None*
add	int × int → int			
		func	0	0
main	int × array(*, char*) → void			
		func	0	0

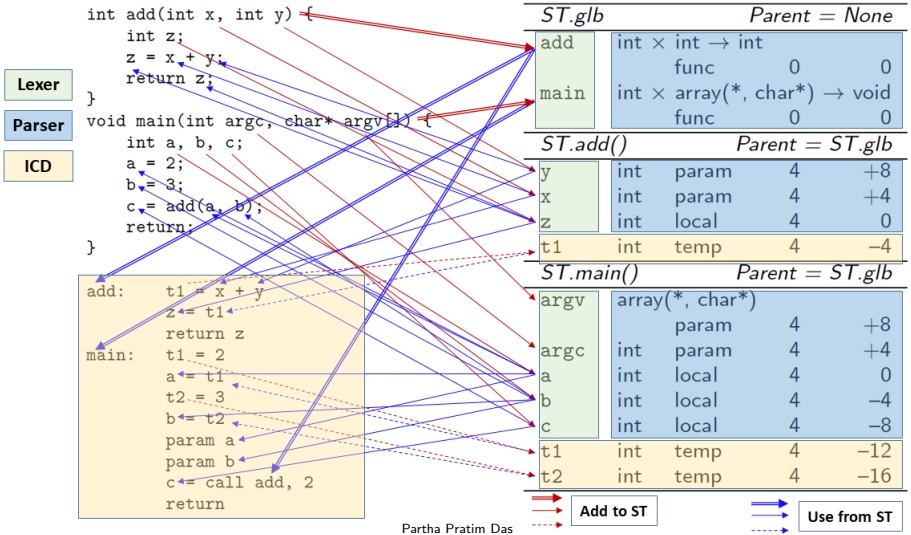
ST.add()				*Parent = ST.glb*
y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0
t1	int	temp	4	–4

ST.main()				*Parent = ST.glb*
argv	array(*, char*)			
		param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	–4
c	int	local	4	–8
t1	int	temp	4	–12
t2	int	temp	4	–16

Columns: Name, Type, Category, Size, & Offset

Example of Symbol Tables

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary



main() & add(): Peep-hole Optimized

```
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main(int argc, char* argv[]) {
    int a, b, c;
    a = 2;
    b = 3;
    c = add(a, b);
    return;
}
```

```
add:    z = x + y
        return z
main:   a = 2
        b = 3
        param a
        param b
        c = call add, 2
        return
```

ST.glb				*Parent = None*
add	int × int → int			
		func	0	0
main	int × array(*, char*) → void			
		func	0	0

ST.add()				*Parent = ST.glb*
y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0

ST.main()				*Parent = ST.glb*
argv	array(*, char*)			
		param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	–4
c	int	local	4	–8

Columns: Name, Type, Category, Size, & Offset

Activation Record

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

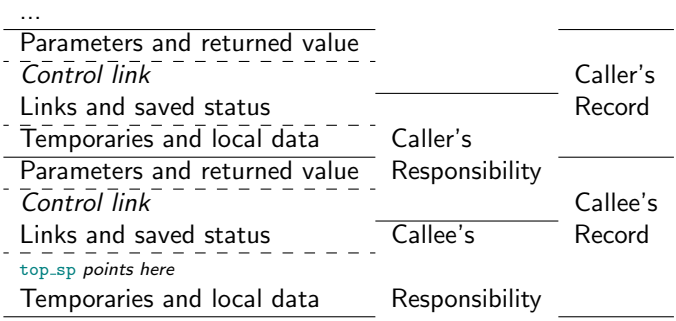
Actual Params	The actual parameters used by the calling procedure (often placed in registers for greater efficiency).
Returned Values	Space for the return value of the called function (often placed in a register for efficiency). Not needed for `void` type.
Return Address	The return address (value of the program counter, to which the called procedure must return).
Control Link	A control link, pointing to the activation record of the caller.
Access Link	An "access link" to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
Saved Machine Status	A saved machine status (state) just before the call to the procedure. This information typically includes the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
Local Data	Local data belonging to the procedure.
Temporary Variables	Temporary values arising from the evaluation of expressions (in cases where those temporaries cannot be held in registers).

Calling & Return Sequences

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Calling Sequences**:
  Consists of code that allocates an activation record on the stack and enters information into its fields.
  The code in a calling sequence is divided between
  - The calling procedure (the "caller") and
  - The procedure it calls (the "callee").
- **Return Sequence**:
  Restores the state of the machine so the calling procedure can continue its execution after the call.

Calling & Return Sequences



Calling & Return Sequences: Calling Sequences

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

The calling sequence and its division between caller and callee:

- **Caller's Responsibility**
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of `top_sp` into the callee's activation record. The caller then increments `top_sp` to the position shown – just past the caller's local data and temporaries and the callee's parameters and status fields.
- **Callee's Responsibility: Function Prologue**
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.

Calling & Return Sequences: Return Sequence

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

A suitable, corresponding return sequence is:

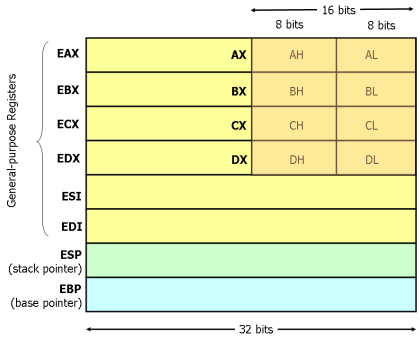
- **Callee's Responsibility: Function Epilogue**
  - The callee places the return value next to the parameters.
  - Using information in the machine-status field, the callee restores `top_sp` and other registers, and then branches to the return address that the caller placed in the status field.
- **Caller's Responsibility**
  - Although `top_sp` has been decremented, the caller knows where the return value is, relative to the current value of `top_sp`; the caller therefore may use that value.

Symbol Table to Activation Record: Functions

Symbol Table **3-Address Code** *Compile Time*	**Activation Record** **Target Code** *Run Time*

• Parameters
• Local Variables
• Temporary
• Nested Block
 	
Nested blocks are flattened out in the Symbol Table of the Function they are contained in so that all local and temporary variables of the nested blocks are allocated in the activation record of the function.
	• Variables
	  ◦ Parameters
	  ◦ Local Variables
	  ◦ Temporary
	  ◦ Non-Local References
	• Stack Management
	  ◦ Return Address
	  ◦ Return Value
	  ◦ Saved Machine Status
	• Call-Return Protocol

Understanding Assembly: Registers of x86



Register	Purpose	Remarks
EAX, EBX, ECX, EDX	General Purpose	Available in 32-, 16-, and 8-bits
ESI	Extended Source Index	General Purpose Index Register
EDI	Extended Destination Index	General Purpose Index Register
ESP	Extended Stack Pointer	Current Stack Pointer
EBP	Extended Base Pointer	Pointer to Stack Frame
EIP	Extended Instruction Pointer	Pointer to Instruction under Execution

Source: x86 instruction set and x86 Assembly Guide

Understanding Assembly: Notation and Conventions

- *Memory segment*: `SEGMENT` – `TEXT` / `DATA` / `CONST`: Text or Program / Data / Constant
- *External Symbols*: `EXTERN`: Symbols not defined in the current file
- *Data Declarations*: `DB`: Byte; `DW`: Word (2 bytes); `DD`: Double Word (4 bytes)
- *Pointer Declarations (Address of*: `BYTE PTR`: Byte; `WORD PTR`: Word; `DWORD PTR`: Double word
- *8 Registers (General Purpose)*: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` (base pointer), `esp` (stack pointer)
- *Address Computation* by `[]`: Content of an address – de-referencing. For example, if `ebp = 100`, then `[ebp+8]` is the value stored in location `108`. Also, `a[i]` = `i[a]` = `[a + i]`
- *Addressing Modes*: Implied (`EFLAGS` for `cmp` or `jle`) – Stack (`esp`); Immediate (`2`); Register (`eax`); Register Indirect (`-12[eax]`); Direct / Absolute / Memory (`023D0016H`); Indexed (`[esi + 08]`);
- *Arithmetic* & *Logical*: `mov a, b`: a = b; `add a, b`: a += b; `sub`; `imul`; `idiv`; `neg`; `and`; `or`; `xor`; `not`
- *Load Effective Address*: `lea a, [b + c * d]`: a = b + c * d
- `memset` *in assembly*: `rep stosd`: Stores the contents of `eax` for `ecx` number of times into where `edi` points to – increment (decrement) `edi` (depending on the direction flag) by 4 bytes each time
- *Jump* `<label>`: *Unconditional*: `jmp`. *Conditional*: `je` (`==`); `jne` (`!=`); `jz` (`==0`); `jg` (`>`); `jge` (`>=`); `jl` (`<`); `jle` (`<=`): Based on `EFLAGS` set by `cmp a, b` – `EFLAGS` (bit 6 for `0`; bit 7 for `<0`)
- *Hardwired Stack*: Managed by `esp` with `push` & `pop` operations. Stack grows from higher to lower memory address. Hence, `push` (`pop`) decrements (increments) `esp`
- *Assembly Functions*: `call <label>`: Jump to `<label>` storing *return address* on stack. `ret`: Return by indirect jump to *return address* when done.
- *Stack Frame*: A function has a frame for params / locals – variables are offset from `ebp` (base)

main(): x86 Assembly (MSVC++, 32-bit)

```
PUBLIC    _main
EXTRN     __RTC_CheckEsp:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT ; Symbol Offsets
_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_argc$ = 8      ; size = 4
_argv$ = 12     ; size = 4
_main     PROC ; Code Starts
; 6     : void main(int argc, char *argv[]) {
; Prologue Starts
    push    ebp
    mov     ebp, esp
    sub     esp, 12 ; 0000000cH
    mov     DWORD PTR [ebp-12], 0xccccccccH
    mov     DWORD PTR [ebp-8], 0xccccccccH
    mov     DWORD PTR [ebp-4], 0xccccccccH
; Prologue Ends - Function Body Starts
; 7     :       int a, b, c;
; 8     :       a = 2;
    mov     DWORD PTR _a$[ebp], 2
; 9     :       b = 3;
    mov     DWORD PTR _b$[ebp], 3
; 10    :       c = add(a, b);
    mov     eax, DWORD PTR _b$[ebp]
    push    eax
    mov     ecx, DWORD PTR _a$[ebp]
    push    ecx
    call    _add
    add     esp, 8 ; pop params
    mov     DWORD PTR _c$[ebp], eax
; 11    :       return;
; 12    : }
; Function Body Ends - Epilogue Starts
    xor     eax, eax
    add     esp, 12 ; 0000000cH
    cmp     ebp, esp
    call    __RTC_CheckEsp
    mov     esp, ebp
    pop     ebp
; Epilogue Ends
    ret     0
_main     ENDP ; Code Ends
_TEXT     ENDS
```

- No Edit + Continue
- No Run-time Check
- No Buffer Security Check

add(): x86 Assembly (MSVC++, 32-bit)

```
PUBLIC    _add
EXTRN     __RTC_Shutdown:PROC
EXTRN     __RTC_InitBase:PROC
; Function compile flags: /Odtp /RTCsu
rtc$IMZ    ENDS
_TEXT    SEGMENT ; Symbol Offsets
_z$ = -4      ; size = 4
_x$ = 8       ; size = 4
_y$ = 12      ; size = 4
_add    PROC ; Code Starts
; 1    : int add(int x, int y) {
; Prologue Starts
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR [ebp-4], 0xccccccccH
; Prologue Ends - Function Body Starts
; 2    :     int z;
; 3    :     z = x + y;
    mov    eax, DWORD PTR _x$[ebp]
    add    eax, DWORD PTR _y$[ebp]
    mov    DWORD PTR _z$[ebp], eax
; 4    :     return z;
    mov    eax, DWORD PTR _z$[ebp]

; 5    : }
; Function Body Ends - Epilogue Starts
    mov    esp, ebp
    pop    ebp
; Epilogue Ends
    ret    0
_add    ENDP ; Code Ends
_TEXT    ENDS
```

- No Edit + Continue
- No Run-time Check
- No Buffer Security Check

ARs of main() and add(): Compiled Code

ST & AR of main()

argv	array(*, char*)		
		prm	+8
argc	int(4)	prm	+4
a	int(4)	lcl	0
b	int(4)	lcl	−4
c	int(4)	lcl	−8

1012	−12	c
1016	−8	b=3
1020	−4	a=2
1024		ebp
1028		RA
1032	+8	argc
1036	+12	argv

ebp = 1024

ST & AR of add()

y	int(4)	prm	+8
x	int(4)	prm	+4
z	int(4)	lcl	0

992	−4	z=5
996		ebp=1024
1000		RA
1004	+8	ecx=2: x
1008	+12	eax=3: y

ebp=996

Code in Execution: main(): Start Address: 0x00

Loc.	Code	esp	ebp	eax	ecx	Stack / Reg.	Value
	; _a$=−4 ; _b$=−8 ; _c$=−12	1028	?	?	?		
0x00	push ebp	1024				[1024] =	ebp
0x01	mov ebp, esp		1024				
0x03	sub esp, 12 ; 0x0000000c	1012					
0x06	mov DWORD PTR [ebp−12], 0xcccccccc ;#fill					c = [1012] =	#fill
0x0d	mov DWORD PTR [ebp−8], 0xcccccccc ;#fill					b = [1016] =	#fill
0x14	mov DWORD PTR [ebp−4], 0xcccccccc ;#fill					a = [1020] =	#fill
0x1b	mov DWORD PTR _a$[ebp], 2					a = [1020] =	2
0x22	mov DWORD PTR _b$[ebp], 3					b = [1016] =	3
0x29	mov eax, DWORD PTR _b$[ebp]			3		eax =	[1016] = 3
0x2c	push eax	1008				y = [1008] =	eax = 3
0x2d	mov ecx, DWORD PTR _a$[ebp]				2	ecx =	[1020] = 2
0x30	push ecx	1004				x = [1004] =	ecx = 2
0x31	call _add	1000				RA = [1000] =	epi = 0x36
						epi = _add (0x50)	

Code in Execution: add(): Start Address: 0x50

Loc.	Code	esp	ebp	eax	ecx	Stack/Reg.	Value
	;_x$=8 ;_y$=12 ;_z$=–4	1000	1024	3	2		
0x50	push ebp	996				[996] =	ebp = 1024
0x51	mov ebp, esp		996				
0x53	push ecx	992					
0x54	mov DWORD PTR [ebp–4], 0xccccccccH ;#fill					z = [992] =	#fill
0x5b	mov eax, DWORD PTR _x$[ebp]			2		eax =	x = [1004] = 2
0x5e	add eax, DWORD PTR _y$[ebp]			5		eax =	eax+=y= ([1008]=3)
0x61	mov DWORD PTR _z$[ebp], eax					z = [992] =	eax = 5
0x64	mov eax, DWORD PTR _z$[ebp]			5		eax =	z = [992] = 5
0x67	mov esp, ebp	996					
0x69	pop ebp	1000	1024			ebp =	[1024]
0x6a	ret 0	1004				epi =	[1000] = 0x36

Code in Execution: main(): Start Address: 0x36

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Loc.	Code	esp	ebp	eax	ecx	Stack / Reg.	Value
	; On return	1004		5	2	epi =	[1000]
0x36	add esp, 8	1012					
0x39	mov DWORD PTR _c$[ebp], eax					c = [1012] =	eax = 5
0x3c	xor eax, eax			0		eax =	0
0x3e	add esp, 12 ; 0x0000000c	1024					
0x41	cmp ebp, esp					status = ?	
0x43	call __RTC_CheckEsp	1020				[1020] =	epi = 0x48
0x48	mov esp, ebp	1024					
0x4a	pop ebp	1028	?			ebp =	[1024]
0x4b	ret 0	1032					

Example: main() & add(): Using I/O

```
#include <stdio.h>

int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main() {
    int a, b, c;

    scanf("%d%d", &a, &b);
    c = add(a, b);
    printf("%d\n", c);
    return;
}
```

Let us build in Debug Mode

add(): Debug Build

```
PUBLIC    _add
EXTRN     __RTC_Shutdown:PROC
EXTRN     __RTC_InitBase:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT
_z$ = -4      ; size = 4
_x$ = 8       ; size = 4
_y$ = 12      ; size = 4
_add      PROC

; 3     : int add(int x, int y) {
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR [ebp-4], 0xccccccccH

; 4     :     int z;
; 5     :     z = x + y;
    mov     eax, DWORD PTR _x$[ebp]
    add     eax, DWORD PTR _y$[ebp]
    mov     DWORD PTR _z$[ebp], eax

; 6     :     return z;
    mov     eax, DWORD PTR _z$[ebp]

; 7     : }
    mov     esp, ebp
    pop     ebp
    ret     0
_add      ENDP
_TEXT     ENDS
```

- No change from earlier – as expected

main(): Debug Build

```
PUBLIC    _main
EXTRN     __imp__printf:PROC
EXTRN     __imp__scanf:PROC
EXTRN     @_RTC_CheckStackVars@8:PROC
EXTRN     __RTC_CheckEsp:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT
_c$ = -28      ; size = 4
_b$ = -20      ; size = 4
_a$ = -8       ; size = 4
_main     PROC

; 8    : void main() {
    push   ebp
    mov    ebp, esp
    sub    esp, 28  ; 0000001cH
    push   esi
    mov    eax, 0xccccccccH
    mov    DWORD PTR [ebp-28], eax
    mov    DWORD PTR [ebp-24], eax
    mov    DWORD PTR [ebp-20], eax
    mov    DWORD PTR [ebp-16], eax
    mov    DWORD PTR [ebp-12], eax
    mov    DWORD PTR [ebp-8], eax
    mov    DWORD PTR [ebp-4], eax
```

```
; 9    :     int a, b, c;
; 10   :
; 11   :     scanf("%d%d", &a, &b);
    mov    esi, esp
    lea    eax, DWORD PTR _b$[ebp]
    push   eax ; Address of b is passed
    lea    ecx, DWORD PTR _a$[ebp]
    push   ecx ; Address of a is passed
    push   OFFSET $SG2756
    call   DWORD PTR __imp__scanf
    add    esp, 12 ; 0000000cH
    cmp    esi, esp
    call   __RTC_CheckEsp

; 12   :     c = add(a, b);
    mov    edx, DWORD PTR _b$[ebp]
    push   edx ; Value of b is passed
    mov    eax, DWORD PTR _a$[ebp]
    push   eax ; Value of a is passed
    call   _add
    add    esp, 8 ; pop params
    mov    DWORD PTR _c$[ebp], eax
```

main(): Debug Build

```
; 13   :     printf("%d\n", c);
    mov    esi, esp
    mov    ecx, DWORD PTR _c$[ebp]
    push   ecx ; Value of c is passed
    push   OFFSET $SG2757
    call   DWORD PTR __imp__printf
    add    esp, 8
    cmp    esi, esp
    call   __RTC_CheckEsp
; 14   :     return;
; 15   : }
    xor    eax, eax
    push   edx
    mov    ecx, ebp
    push   eax
    lea    edx, DWORD PTR $LN6@main
    call   @_RTC_CheckStackVars@8
    pop    eax
    pop    edx
    pop    esi
    add    esp, 28 ; 0000001cH
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
```

```
$LN6@main:
    DD    2
    DD    $LN5@main
$LN5@main:
    DD    -8 ; fffffff8H
    DD    4
    DD    $LN3@main
    DD    -20 ; ffffffecH
    DD    4
    DD    $LN4@main
$LN4@main:
    DB    98 ; 00000062H
    DB    0
$LN3@main:
    DB    97 ; 00000061H
    DB    0
_main    ENDP
_TEXT    ENDS
```

- Run-time checks at the end

Example: main() & add(): Using I/O

```
#include <stdio.h>

int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main() {
    int a, b, c;

    scanf("%d%d", &a, &b);
    c = add(a, b);
    printf("%d\n", c);
    return;
}
```

Let us build in Release Mode

add(): Release Build

```
PUBLIC    _add
; Function compile flags: /Ogtp
_TEXT    SEGMENT
; _x$ = ecx
; _y$ = eax

; 4    :     int z;
; 5    :     z = x + y;
    add    eax, ecx

; 6    :     return z;
; 7    : }
    ret    0
_add    ENDP
_TEXT    ENDS
```

- Parameters passed through registers
- No save / restore of machine status
- No use of local (z)

main(): Release Build

```
PUBLIC    _main
; Function compile flags: /Ogtp
_TEXT    SEGMENT
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_main    PROC ; COMDAT

; 8    : void main() {
    push   ebp
    mov    ebp, esp
    sub    esp, 8

; 9    :     int a, b, c;
; 10   :
; 11   :     scanf("%d%d", &a, &b);
    lea    eax, DWORD PTR _b$[ebp]
    push   eax
    lea    ecx, DWORD PTR _a$[ebp]
    push   ecx
    push   OFFSET
        ??_C@_04LLKPOCGK@?$CFd?$CFd?$AA@
    call   DWORD PTR __imp__scanf
```

```
; 12   :     c = add(a, b);
    mov    edx, DWORD PTR _a$[ebp]
    add    edx, DWORD PTR _b$[ebp]

; 13   :     printf("%d\n", c);
    push   edx
    push   OFFSET
        ??_C@_03PMGGPEJJ@?$CFd?6?$AA@
    call   DWORD PTR __imp__printf
    add    esp, 20 ; 00000014H

; 14   :     return;
; 15   : }

    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

- No unnecessary save / restore of machine status
- Call to add() optimized out!

Decoding Assembly: Workout Example

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Consider a simple program:

```
#include <stdio.h>
int main() {
   int num1 = 2, num2 = 3, sum;
   sum = num1 + num2;
   printf("Sum = %d\n", sum);
   return 0;
}
```

Decoding Assembly: Workout Example

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
; ...

_DATA  SEGMENT
$SG3049        DB      'Sum = %d', 0aH, 00H ; string constant on Data Segment. 0aH = '\n'. 00H = NULL
_DATA  ENDS

PUBLIC _main
EXTRN  __imp__printf:PROC
EXTRN  __RTC_CheckEsp:PROC

; ...

_TEXT  SEGMENT  ; Program code

; DESCRIPTION OF STACK FRAME OF main

_sum$ = -12 ; size = 4 ; _sum$ = -12 is offset for sum. Address of sum is ebp-12
_num2$ = -8 ; size = 4 ; _num2$ = -8 is offset for num2. Address of num2 is ebp-8
_num1$ = -4 ; size = 4 ; _num1$ = -4 is offset for num1. Address of num1 is ebp-4
```

Decoding Assembly: Workout Example

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
_main  PROC ; Function main code start

; 4    : {

; PROLOGUE OF main STARTS

    push   ebp ; save ebp (base pointer) to remember the frame information for caller
               ; register and stack addressing
    mov    ebp, esp ; initialize epb to esp - frame of this function will be allocated from here
    sub    esp,  12 ; 0000000cH ; reserve 12 = 4 * 3 bytes for num1, num2, and sum of the frame of main
                    ; register and immediate addressing
    push   esi ; save esi - used as a temporary register
               ;  cccccccH = -858993460 is the uninitialized value / garbage marker
    mov    DWORD PTR [ ebp-12], -858993460 ; cccccccH ; init. sum to marker. Address of sum is ebp-12
    mov    DWORD PTR [ ebp-8], -858993460  ; cccccccH ; init. num2 to marker. Addr. of num2 is ebp-8
    mov    DWORD PTR [ ebp-4], -858993460  ; cccccccH ; init. num1 to marker. Addr. of num1 is ebp-4

; 5    :      int num1 = 2, num2 = 3, sum;

    mov    DWORD PTR _num1$[ebp], 2 ; _num1$[ebp] = ebp-4 is address of num1. Init. num1 with 2
    mov    DWORD PTR _num2$[ebp], 3 ; _num2$[ebp] = ebp-8 is address of num2. Init. num2 with 3
                                    ; register indirect and immediate addressing

; PROLOGUE OF main ENDS
```

Decoding Assembly: Workout Example

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
; 6    :
; 7    :      sum = num1 + num2;

       mov    eax, DWORD PTR _num1$[ebp] ; load num1 to eax (accumulator)
       add    eax, DWORD PTR _num2$[ebp] ; add num2 to eax. eax becomes num1 + num2 = 5
       mov    DWORD PTR _sum$[ebp], eax ; store eax to sum. sum becomes 5
                                        ; register and register indirect addressing

; 8    :
; 9    :      printf("Sum = %d\n", sum);

       mov    esi, esp ; save esp (stack pointer) to esi to prepare stack to pass parameters
       mov    ecx, DWORD PTR _sum$[ebp] ; load sum to ecx
       push   ecx ; push ecx to stack. Param 2 for printf. esp -= 4 as side-effect
       push   OFFSET $SG3049 ; push offset of ("Sum = %d\n") to stack. Param 1 for printf. esp -= 4
       call   DWORD PTR __imp__printf ; call external & imported printf - gets 2 params on top of stack
       add    esp, 8 ; esp += 8 to pop two params passed before call
       cmp    esi, esp ; compare esp with esi - the value of esp before call. EFLAGS to be set
       call   __RTC_CheckEsp ; check EFLAGS to confirm that esp matches its value before call

; 10   :
; 11   :      return 0;

       xor    eax, eax ; eax = eax ^ eax = 0. A one cycle instruction to clear eax

; 12   : }
```

Decoding Assembly: Workout Example

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
; EPILOGUE OF main STARTS

        pop     esi ; restore esi
        add     esp,  12 ; 0000000cH ; release 12 bytes of the frame of main
        cmp     ebp, esp ; compare esp with ebp - the value of esp before frame of main was reserved
        call    __RTC_CheckEsp ; check EFLAGS to confirm that esp matches its value before call
        mov     esp, ebp ; restore esp
        pop     ebp ; restore ebp { the frame of the parent (caller) function
        ret     0 ; Return 0. Control returns through indirect jump

; EPILOGUE OF main ENDS

_main   ENDP  ; Function main code end

_TEXT   ENDS
END
```

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Symbol Properties

Dragon Book: Pages 25-31 (Programming Language Basics)
Examples and Expansions by PPD

Properties of a Symbol

A symbol is an identifier in C.

- A symbol may be:
  - A variable
    - Simple identifier
    - Array
    - Structure or Union
    - Pointer
  - A function
  - A label
  - A type alias
- A symbol has multiple properties in *static* as well as *dynamic* contexts

Properties of a Symbol

A symbol has multiple properties in *static* as well as *dynamic* contexts including:

- **Static / Compile Time Properties**
  - *Declaration*: The (lexical) name of the symbol with its data type and qualifier/s, and the initial value if any it takes
  - *Definition*: The memory area (where the variable gets stored) and amount of storage to create for the variable. The allocation may be static or dynamic.
  - *Initialization*: The initial value of the symbol specified at a declaration
  - *Scope*: The regions of a program where the symbol may directly be accessible[1].
- **Static-Dynamic Bridge / Mixed Properties**
  - *Binding*: It is the association of entities (data and/or code) with symbols. An symbols bound to an entity / object is said to reference that object.
  - *Storage Class*: Every variable has a storage class that specifies the storage duration
- **Dynamic / Run Time Properties**
  - *Address*: The physical memory address of the symbol at rum-time
  - *Value*: The value of the symbol during execution. All symbols may not have a value
  - *Lifetime*: The time between the *definition* and *de-allocation* of a variable

[1] In C, symbols are statically scoped. Dynamic scoping, though rare, is supported in languages like LaTeX, bash

Properties of a Symbol: Declaration

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Declaration**: The (lexical) name of the symbol with its data type and qualifier/s
  - Variable

```
// Symbol Name = "sum", Symbol Type = "int"
int sum;

// Symbol Name = "array_size", Symbol Type = "const int"
const int array_size = 10;
```

  - Function

```
// Symbol Name = "info", Symbol Type = "int --> int"
int fibo(int);
```

- Declaration are maintained in the Symbol Table
- Declaration are processed at multiple phases
  - Lexcical Analyzer tokenizes the symbol (`sum` or `fibo`) and creates entry in Symbol Table
  - Syntax Analyzer adds the type information (`int` or `int` $\rightarrow$ `int`) on Symbol Table
  - The symbol's size information is also entered. This will be used to created the final offset of the symbol in the Activation Record

Properties of a Symbol: Definition

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Definition**: The static or dynamic allocation of the memory address of the symbol. It is the process of creating the binding
  - Variable: Address computation logic is available

```
// Symbol Name = "sum", Symbol Scope = Global
extern int sum; // Declaration without definition - definition created by linking
// Symbol Name = "sum", Symbol Scope = Global
int sum; // Definition at declaration
// Symbol Name = "temp", Symbol Scope = Block
 ... int temp; /* Definition at declaration */ ...
// Symbol Name = "*p", Symbol Scope = Dynamic
int *p = 0; // p is declared, def. and initialized - static
p = malloc(sizeof(int); // *p is defined - dynamic
```

  - Function: A function can have only one definition or function body

```
int fibo(int n)  if (0 == n) return 1; else return n*fibo(n-1);
```

- Definitions typically result in TAC during intermediate code generation that use various symbol information from the Symbol Table
- This process may involve compiler-defined (un-named) temporary variables that also go into the Symbol Table. For example `sum = sum + 1;` may be translated to:

```
t1 = sum + 1 // t1 is un-named temporary
sum = t1
```

Properties of a Symbol: Initialization

- **Initialization**: The initial value of the symbol specified at a declaration

```
// Symbol Name = "sum", Symbol Type = "int",
// Symbol Initialization = "0"
int sum = 0;

// Symbol Name = "p", Symbol Type = "int*",
// Symbol Initialization = "&sum"
int *p = &sum;
```

- Initialization is maintained in the Symbol Table along with the Declaration of the symbol
- Initialization (static time) is different from assignment (run-time)
- Initialization usually is optional
- Initialization is processed at multiple phases
  - Lexcical Analyzer tokenizes the initialization constant (`0`)
  - Syntax Analyzer adds the initialization information on Symbol Table
  - Semantic Analyzer evaluates the constant initialization expression (like `const double pi = 4.0*atan(1.0);` and updates Symbol Table
  - Note that `class Shape { ... virtual void Draw() = 0; ... };` is not an initialization, but semantic specifier for pure virtual functions

Properties of a Symbol: Scope

- **Scope**: The regions of a program where the symbol may directly be accessible. In C, all symbols are statically / lexically scoped[2]. C supports the following scopes[3]:
  - *Global Scope*: Scope of a symbol that is *declared outside of all functions*.
    - ▷ It spans *across all files* of a program.
    - ▷ A program has *only one global scope* and *all scopes are nested within it*.
  - *File Scope*: Any *source file in entirety* where a symbol is marked by `static` to belong to this scope.
  - *Block Scope*: Region within a pair of curly braces (`{ ...}`)
    - ▷ Symbols within block scope are *local to their block*.
    - ▷ A *block scope may be nested* within a function scope or some other block scope as a child.
    - ▷ All symbols from parent scopes are visible in the block and may be hidden by *re-declaration*.
  - *Function Scope*: The block scope associated with a *function definition*.
    - ▷ Every function scope is *contained within global or file scope* and may not be nested.
    - ▷ Function scope is *applicable to labels only*. A label declared is used as a target to `goto` statement and both `goto` and label statement must be in same function.
    - ▷ Symbols declared in function prototype are visible within the *Function Prototype Scope*
- Every scope is associated with a Symbol Table that contains its symbols
- A scope of a variable is often closely related to its lifetime, though they may be different[4].

[2] Some languages use dynamic scopes too

[3] Languages like C++ support more scopes like class scope, namespace scope

[4] For example, for dynamic allocation

Properties of a Symbol: Scope: Examples

```
/***** file1.c *****/
#include <stdio.h>
int i = 5, k;
extern int j;
int add(int, int);
int mul(int, int);

int main()
{
    printf("i = %d, j = %d, k = %d\n", i, j, k); // 1
    {
        int i = 7;
        j = 4;
        printf("i = %d, j = %d, k = %d\n", i, j, k); // 2
    }
    printf("i = %d, j = %d, k = %d\n", i, j, k); // 3
    int x = 3, y = 2;
    printf("Add = %d\nMul = %d\n", add(x, y), mul(x, y)); // 4
    return 0;
}
int add(int a, int b)
{ return a + trans(b); }

int trans(int a)
{ return a + 1; }
```

```
/***** file2.c *****/
int j = 6;
static int k = 1;


int mul(int a, int b)

{
    int t = a * trans(b);

    return t;
}

static int trans(int a)

{ return a - k; }


-----
What is the output?
```

Properties of a Symbol: Scope: Examples

```c
/***** file1.c *****/
#include <stdio.h>
int i = 5, k; // Global Scope defn for i & k. k init to 0
extern int j; // Global Scope decl for j
int add(int, int); // Global Scope prototype for add
int mul(int, int); // Global Scope prototype for mul

int main() // Global Scope defn for main
{ // Function Scope for main starts
    printf("i = %d, j = %d, k = %d\n", i, j, k); // 1
    { // Block Scope starts
        int i = 7; // Local i defined. Hides global i
        j = 4; // Global j set
        printf("i = %d, j = %d, k = %d\n", i, j, k); // 2
    } // Block Scope ends
    printf("i = %d, j = %d, k = %d\n", i, j, k); // 3
    int x = 3, y = 2; // Local x & y defined
    printf("Add = %d\nMul = %d\n", add(x, y), mul(x, y)); // 4
    return 0;
} // Function Scope for main ends
int add(int a, int b)  // Global Scope defn for add
{ return a + trans(b); } // Function Scope for add
                         // Uses trans in file1.c
int trans(int a) // Global Scope defn for trans
{ return a + 1; } // Function Scope for trans
```

```c
/***** file2.c *****/
int j = 6; // Global Scope defn for j
static int k = 1; // File Scope defn
    // for k - not visible in file1.c

int mul(int a, int b) // Global Scope
                      // defn for mul
{ // Function Scope for mul starts
    int t = a * trans(b); // Local t defn
        // Uses trans in file2.c
    return t;
} // Function Scope for mul ends

static int trans(int a) // File Scope
                        // defn for trans
{ return a - k; } // Function Scope
        // for trans. Uses k in file2.c
```

```
-----
i = 5, j = 6, k = 0 // 1
i = 7, j = 4, k = 0 // 2
i = 5, j = 4, k = 0 // 3
Sum = 6 // 4
Product = 3 // 4
```

Properties of a Symbol: Binding

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Binding / Name Binding**: It is the association of entities (data and/or code) with symbols. An symbols bound to an entity / object is said to reference that object.
- Machine languages have no built-in notion of identifiers, but name-object bindings as a service and notation for the programmer is implemented by programming languages.
- Binding is *intimately connected with scoping*, as scope determines which names bind to which objects:
  - at which locations in the program code (lexically) and
  - in which one of the possible execution paths (temporally).
- Binding may be one of two kinds:
  - *Static binding* (or *Early binding*) is name binding performed before the program is run. C & C++ use static binding.
  - *Dynamic binding* (or *Late binding* or *Virtual binding*) is name binding performed as the program is running. C++ support dynamic binding with virual functions.
- The name binding is created during Intermediate Code Generation phase by resolving the symbol definition for every symbol occurrence. The tree of Symbol Tables is used for this purpose.
- The result of name binding is an appropriate address expressions (like `[ebp] + offset`) that can automatically create the Activation Record at run-time, thereby achieving the binding in an elegant way

Properties of a Symbol: Binding: Examples

```
#include <stdio.h> /***** file1.c *****/
int i = 5, k;
extern int j;  // j
int add(int, int);
int mul(int, int);
int main() {
    printf("i = %d, j = %d, k = %d\n", i, j, k); // i, j, k
    {   int i = 7;
        j = 4; // j
        printf("i = %d, j = %d, k = %d\n", i, j, k); // i, j, k
    }
    printf("i = %d, j = %d, k = %d\n", i, j, k); // i, j, k
    int x = 3, y = 2;
    %d\n", add(x, y), mul(x, y)); // x, y, add, mul
    printf("Add = %d\nMul =
    return 0;
}
int add(int a, int b) { return a + trans(b); } // trans
int trans(int a) { return a + 1; }
/***** file2.c *****/
int j = 6;
static int k = 1;
int mul(int a, int b) { int t = a * trans(b); return t; } // trans
static int trans(int a) { return a - k; } // k
```

Properties of a Symbol: Binding: Examples

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
#include <stdio.h> /***** file1.c *****/
int i = 5, k;
extern int j;  // j: global (file2.c)
int add(int, int);
int mul(int, int);
int main() {
    printf("i = %d, j = %d, k = %d\n", i, j, k); // i: global, j: global, k: global
    {   int i = 7;
        j = 4; // j: global
        printf("i = %d, j = %d, k = %d\n", i, j, k); // i: local, j: global, k: global
    }
    printf("i = %d, j = %d, k = %d\n", i, j, k); // i: global, j: global, k: global
    int x = 3, y = 2;
    // x: local, y: local, add: global (file1.c), mul: global (file2.c)
    printf("Add = %d\nMul = %d\n", add(x, y), mul(x, y));
    return 0;
}
int add(int a, int b) { return a + trans(b); } // trans: global (file1.c)
int trans(int a) { return a + 1; }
/***** file2.c *****/
int j = 6;
static int k = 1;
int mul(int a, int b) { int t = a * trans(b); return t; } // trans: file (file2.c)
static int trans(int a) { return a - k; } // k: file (file2.c)
```

Properties of a Symbol: Storage Class

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Storage Class**: Every variable has a storage class that specifies the storage duration
- A storage duration, may be
  - static (default for global)
  - automatic (default for local)
  - dynamic (allocated), and
  - others like linkage and register hint
- Storage class tells us the following factors:
  - Where the variable is stored (in memory or cpu register)?
  - What will be the initial value of variable, if nothing is initialized?
  - What is the scope of variable (where it can be accessed)?
  - What is the life of a variable?
- Summary:

Specifiers	Lifetime	Scope	Default initializer
`auto` / *(none)*	Block (stack)	Block	Uninitialized
`register`	Block (stack or CPU register)	Block	Uninitialized
`static`	Program	Block or compilation unit	`0`
`extern`	Program	Global (entire program)	`0`
(none)[5]	Dynamic (heap)	Uninitialized	Initialized to `0` if using `calloc()`

[5] Allocated and deallocated using the `malloc()` and `free()` library functions

Properties of a Symbol: Address

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Address**: The physical memory address of the symbol at rum-time

```
// Symbol Name = "sum", Symbol Type = "int"
// Symbol Address = &sum // Address of sum
int sum;
// Symbol Name = "add", Symbol Type = "int x int -> int"
// Symbol Address = &add // Address of add function
int add(int, int);
```

- For example, consider the output of the following program:

```
#include <stdio.h>
int main() {
    int a = 10;
    printf("a = %d\n&a = %p\n", a, &a);
    printf("&printf = %p\n", &printf);
    return 0;
}
a = 10  // Value of variable 'a'
&a = 0x7ffe77e67274  // Address or binding of variable 'a'
&printf = 0x7f0894da2c90  // Address or binding of function 'printf'
```

- During Target Code Generation phase, the symbol offsets in the Symbol Table are converted into address expressions (like `[ebp] + offset`) that can automatically create the Activation Record at run-time, thereby achieving the binding in an elegant way

Properties of a Symbol: Value

- **Value**: The value of the symbol during execution. All symbols may not have a value
  - Variable: Access to the symbol gives its value (*Read*). Any assignment and / or direct / indirect *Write* to the symbol sets its value

```
// Symbol Name = "sum", Symbol Update = By direct assignment
sum = sum + 1;
// Symbol Name = "*p", Symbol Update = By indirect assignment
*p = *p + 1;
```

  - Function: A function invokation gives its value (for the parameters)

```
int fibo(int n) { if (0 == n) return 1; else return n*fibo(n-1); }

fibo(5); // Has value 120
```

- Assignments typically result in TAC during intermediate code generation that use various symbol information from the Symbol Table
- This process may involve compiler-defined (un-named) temporary variables that also go into the Symbol Table. For example `sum = sum + 1;` may be translated to:

```
t1 = sum + 1 // t1 is un-named temporary
sum = t1
```

Properties of a Symbol: Lifetime

- **Lifetime / Life Cycle**: The time between the *definition* and *de-allocation* of a variable[6]
- **Execution Stages**
  - Memory Allocation and Binding
  - Lifetime
    - Control passes the definition (and optional initialization) of the variable or dynamic allocation[7]
    - Object Use
    - Control exists the scope or dynamic de-allocation[8]
  - Memory De-Allocation and De-Binding
- Lifetime depends on scope and nature of variable
  - Automatic
  - Static
    - Global
    - Local
  - Dynamic

Sources: Module M13: Constructors, Destructors & Object Lifetime in Programming in Modern C++, NPTEL

[6] In an OOPL like C++, it is the time when control enters constructor body to its exit from destructor body
[7] Constructor Call and Execution in an OOPL like C++
[8] Destructor Call and Execution in an OOPL like C++

Properties of a Symbol: Lifetime: Examples

```
/* 01 */ #include <stdio.h>
/* 02 */ #include <stdlib.h> // E00: sum & accu allocated
/* 03 */ int sum; // E01: Static global
/* 04 */ int add(int n) { // E06, E15, E20: Automatic
/* 05 */     static int accu = 0; // E07: Static local
/* 06 */     int t = accu + n; // E08, E16, E21: Automatic
/* 07 */     accu = t;
/* 08 */     return t; // E09, E17, E22
/* 09 */ }
/* 10 */ int main() { // E02
/* 11 */     char fmt[] =
/* 12 */        "Sum = %d\n"; // E03: Automatic
/* 13 */     int a = 2; // E04: Automatic
/* 14 */     add(a); // E05
/* 15 */     int *p; // E10: Automatic
/* 16 */     p = (int *)
/* 17 */        malloc(sizeof(int)); // E11: Dynamic
/* 18 */     *p = 3; // E12
/* 19 */     int b = 5; // E13: Automatic
/* 20 */     add(*p); // E14
/* 21 */     free(p); // E18: Dynamic
/* 22 */     sum = add(b); // E19, E23
/* 23 */     printf(fmt, sum);
/* 24 */     return 0; // E24
/* 25 */ } // E25: main returns. sum & accu de-allocated
```

E01	
E02	
E03	
E04	
E05	
E06	
E07	
E08	
E09	
E10	
E11	
E12	
E13	
E14	
E15	
E16	
E17	
E18	
E19	
E20	
E21	
E22	
E23	
E24	

Properties of a Symbol: Lifetime: Examples

```
/* 01 */ #include <stdio.h>
/* 02 */ #include <stdlib.h> // E00: sum & accu allocated
/* 03 */ int sum; // E01: Static global
/* 04 */ int add(int n) { // E06, E15, E20: Automatic
/* 05 */     static int accu = 0; // E07: Static local
/* 06 */     int t = accu + n; // E08, E16, E21: Automatic
/* 07 */     accu = t;
/* 08 */     return t; // E09, E17, E22
/* 09 */ }
/* 10 */ int main() { // E02
/* 11 */     char fmt[] =
/* 12 */        "Sum = %d\n"; // E03: Automatic
/* 13 */     int a = 2; // E04: Automatic
/* 14 */     add(a); // E05
/* 15 */     int *p; // E10: Automatic
/* 16 */     p = (int *)
/* 17 */        malloc(sizeof(int)); // E11: Dynamic
/* 18 */     *p = 3; // E12
/* 19 */     int b = 5; // E13: Automatic
/* 20 */     add(*p); // E14
/* 21 */     free(p); // E18: Dynamic
/* 22 */     sum = add(b); // E19, E23
/* 23 */     printf(fmt, sum);
/* 24 */     return 0; // E24
/* 25 */ } // E25: main returns. sum & accu de-allocated
```

E01	sum def.: *Life starts*
E02	main called. fmt, a, p & b alloc.
E03	fmt def. (w/ init): *Life starts*
E04	a def. (w/ init): *Life starts*
E05	add called
E06	n & t alloc. n def. (w/ init): *Life starts*
E07	accu def. (w/ init): *Life starts*
E08	t def. (w/ init): *Life starts*
E09	n & t: *Life ends*. De-alloc @ line 09
E10	add returns. p def.: *Life starts*
E11	*p alloc.: *Life starts*
E12	*p assigned
E13	b def. (w/ init): *Life starts*
E14	add called
E15	n & t alloc. n def. (w/ init): *Life starts*
E16	t def. (w/ init): *Life starts*
E17	n & t: *Life ends*. De-alloc @ line 09
E18	add returns. *p de-alloc. *Life ends*
E19	add called
E20	n & t alloc. n def. (w/ init): *Life starts*
E21	t def. (w/ init): *Life starts*
E22	n & t: *Life ends*. De-alloc @ line 09
E23	add returns. sum assigned
E24	fmt, a, p & b: *Life ends*. De-alloc @ line 25

Parameter Passing and Return Value

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- **Call-by-Value (CBV)**
  - **C, C++, ALGOL, Scheme**: the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (by copying the value into a new memory region)
  - *Eager Evaluation*
  - **Return-by-Value** supported in **C, C++**
- **Call-by-Reference (CBR)**
  - **C++, C#**: a function receives an implicit reference to a variable used as argument, rather than a copy of its value. **Call-by-Reference-to-const (CBRc)** available in **C#** as well (array parameter)
  - **C, C++**: CBR may be simulated in languages that use CBV by making use of references, such as pointers (**Call-by-Address or CBA**)
  - **Return-by-Reference** supported in **C++**
- **Call-by-Copy-Restore (CBCR) / Value-Result**
  - **Fortran IV, Ada**: a special case of call by reference where the provided reference is unique to the caller (**Copy-in-Copy-out**)
- **Call-by-Name (CBN)**
  - **C / C++ Macro, ALGOL 60, Simula**: the arguments to a function are not evaluated before the function is called – rather, they are substituted directly into the function body
  - *Lazy Evaluation*
  - **Call-by-Need (Haskell, R)**: a memorized variant of CBN where, if the function argument is evaluated, that value is stored for subsequent uses
  - **Call-by-Push-Value (CBPV)**: inspired by monads, allows writing semantics for λ-calculus without writing two variants to deal with the difference between **CBN** and **CBV**

Source: *Evaluation strategy*, Wikipedia

Parameter Passing and Return Value: Example

```
#include <iostream>
using namespace std;

void f(int a, int b) { a++; b--; return; }              // CBV
void g(int& a, int& b) { a++; b--; return; }            // CBR
void h(int* pa, int* pb) { (*pa)++; (*pb)--; return; } // CBA
#define m_f(a, b) ( a * b )                             // CBN

int main() {
    int x = 3, y = 4, z = 5;
    f(x, y);
    cout << x << " " << y << endl;          // CBV = 3 4
    g(x, y);
    cout << x << " " << y << endl;          // CBR = 4 3
    h(&x, &y); // x = 4, y = 3
    cout << x << " " << y << endl;          // CBA = 5 2
    g(z, z);
    cout << z << endl;                      // CBR = 5
                                            // CBCR = 6 (z <- a) or 4 (z <- b)
    cout << m_f(x + 1, y + 1) << endl;      // CBN = x + 1 * y + 1 = x + y + 1 = 8
}
```

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Rest of the Module Self-Study

Array

Scope: Nested Blocks & Global / Static

Practice for Assignments & Quizzes

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Array Data Type

Examples by PPD

Example: main() & Sum(): Using Array & Nested Block

```
#include <stdio.h>

int Sum(int a[], int n) {
    int i, s = 0;
    for(i = 0; i < n; ++i) {
        int t;
        t = a[i];
        s += t;
    }
    return s;
}
void main() {
    int a[3];
    int i, s, n = 3;
    for(i = 0; i < n; ++i)
        a[i] = i;
    s = Sum(a, n);
    printf("%d\n", s);
}
```

```
Sum:    s = 0
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        t_1 = a[t1]
        s = s + t_1
        goto L1
L3:     return s
```

Block local variable `t` is named as `t_1` to qualify for the unnamed block within which it occurs.

```
main:   n = 3
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        a[t1] = i
        goto L1
L3:     param a
        param n
        s = call Sum, 2
        param "%d\n"
        param s
        call printf, 2
        return
```

Parameter `s` of `printf` is handled through varargs.

ST.glb: ST.glb.parent = null

Sum	array(*, int) × int → int	function	0	ST.Sum
main	void → void	function	0	ST.main

ST.main(): ST.main.parent = ST.glb

a	array(3, int)	local	12	0
i	int	local	4	12
s	int	local	4	16
n	int	local	4	20
t1	int	temp	4	24

ST.Sum(): ST.Sum.parent = ST.glb

a	int[]	param	4	0
n	int	param	4	4
i	int	local	4	8
s	int	local	4	12
t_1	int	local	4	16
t1	int	temp	4	20

Columns are: Name, Type, Category, Size, & Offset

main()

```
PUBLIC    _main
; Function compile flags: /Odtp /RTCsu
_TEXT    SEGMENT
_n$ = -32     ; size = 4
_s$ = -28     ; size = 4
_i$ = -24     ; size = 4
_a$ = -16     ; size = 12
_main    PROC

; 12   : void main() {

    push   ebp
    mov    ebp, esp
    sub    esp, 32 ; 00000020H
    push   esi
    mov    eax, -858993460 ; ccccccccH
    mov    DWORD PTR [ebp-32], eax
    mov    DWORD PTR [ebp-28], eax
    mov    DWORD PTR [ebp-24], eax
    mov    DWORD PTR [ebp-20], eax
    mov    DWORD PTR [ebp-16], eax
    mov    DWORD PTR [ebp-12], eax
    mov    DWORD PTR [ebp-8], eax
    mov    DWORD PTR [ebp-4], eax

; 13   :     int a[3];
; 14   :     int i, s, n = 3;

    mov    DWORD PTR _n$[ebp], 3

; 15   :     for(i = 0; i < n; ++i)

    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    mov    ecx, DWORD PTR _i$[ebp]
    cmp    ecx, DWORD PTR _n$[ebp]
    jge    SHORT $LN1@main

; 16   :         a[i] = i;

    // Index in edx
    mov    edx, DWORD PTR _i$[ebp]
    // Right-hand Expression in eax
    mov    eax, DWORD PTR _i$[ebp]
    // Index expression directly used
    mov    DWORD PTR _a$[ebp+edx*4], eax
    jmp    SHORT $LN2@main
$LN1@main:
```

- Array reference in a uses index expression in code – no temporary used
- `for` loop condition implemented as `cmp` and conditional jump `jge`

main()

```
; 17   :     s = Sum(a, n);

    mov    ecx, DWORD PTR _n$[ebp]
    push   ecx
    lea    edx, DWORD PTR _a$[ebp]
    push   edx
    call   _Sum
    add    esp, 8
    mov    DWORD PTR _s$[ebp], eax

; 18   :     printf("%d\n", s);

    mov    esi, esp
    mov    eax, DWORD PTR _s$[ebp]
    push   eax
    push   OFFSET $SG2765
    call   DWORD PTR __imp__printf
    add    esp, 8
    cmp    esi, esp
    call   __RTC_CheckEsp

; 19   : }

    xor    eax, eax
    push   edx
    mov    ecx, ebp
    push   eax
    lea    edx, DWORD PTR $LN8@main
    call   @_RTC_CheckStackVars@8
    pop    eax
    pop    edx
    pop    esi
    add    esp, 32 ; 00000020H
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
    npad   3
$LN8@main:
    DD     1
    DD     $LN7@main
$LN7@main:
    DD     -16 ; fffffff0H
    DD     12 ; 0000000cH
    DD     $LN6@main
$LN6@main:
    DB     97 ; 00000061H
    DB     0
_main    ENDP
_TEXT    ENDS
END
```

- `lea` used to pass parameter in a

Sum()

```
PUBLIC    _Sum
EXTRN     __RTC_Shutdown:PROC
EXTRN     __RTC_InitBase:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT
_t$2755 = -12; size = 4
_s$ = -8      ; size = 4
_i$ = -4      ; size = 4
_a$ = 8       ; size = 4
_n$ = 12      ; size = 4
_Sum      PROC
; 3    : int Sum(int a[], int n) {
    push   ebp
    mov    ebp, esp
    sub    esp, 12 ; 0000000cH
    mov    DWORD PTR [ebp-12], 0xcccccccch
    mov    DWORD PTR [ebp-8], 0xcccccccch
    mov    DWORD PTR [ebp-4], 0xcccccccch
; 4    :     int i, s = 0;
    mov    DWORD PTR _s$[ebp], 0
; 5    :     for(i = 0; i < n; ++i) {
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@Sum
$LN2@Sum:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@Sum:
    mov    ecx, DWORD PTR _i$[ebp]
    cmp    ecx, DWORD PTR _n$[ebp]
    jge    SHORT $LN1@Sum
```

```
; 6    :          int t;
; 7    :          t = a[i];
    mov    edx, DWORD PTR _i$[ebp]
    mov    eax, DWORD PTR _a$[ebp]
    mov    ecx, DWORD PTR [eax+edx*4]
    mov    DWORD PTR _t$2755[ebp], ecx
; 8    :          s += t;
    mov    edx, DWORD PTR _s$[ebp]
    add    edx, DWORD PTR _t$2755[ebp]
    mov    DWORD PTR _s$[ebp], edx
; 9    :     }
    jmp    SHORT $LN2@Sum
$LN1@Sum:
; 10   :     return s;
    mov    eax, DWORD PTR _s$[ebp]
; 11   : }
    mov    esp, ebp
    pop    ebp
    ret    0
_Sum      ENDP
_TEXT     ENDS
```

- a is reference parameter – &a[0]
- Local variable declaration `int t;` in block is renamed to _t$2755 instead of _t$ to track unnamed block

Activation Records of main() & Sum()

Offset	Addr.	Stack	Description
−12	960	t (_t$2755)	Local data w/ buffer
−8	964	s	
−4	968	i	
ebp →	972	ebp (of main())	Control link
	976	Return Address	
+8	980	a	Reference Param – &a[0]
+12	984	n	
	988	esi	Saved registers
−32	992	n	Local data w/ buffer
−28	996	s	
−24	1000	i	
	1004	0xcccccccc	
−16	1008	a [0]	
	1012	a [1]	
	1016	a [2]	
	1020	0xcccccccc	
ebp →	1024	ebp (of Caller of main())	Control link
	1028	Return Address	

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

Scopes

Examples by PPD

Example: Nested Blocks: Source & TAC

```
int a;
int f(int x) { // function scope f
    int t, u;
    t = x; // t in f, x in f
    { // un-named block scope f_1
        int p, q, t;
        p = a; // p in f_1, a in global
        t = 4; // t in f_1, hides t in f
        { // un-named block scope f_1_1
            int p;
            p = 5; // p in f_1_1, hides p in f_1
        }
        q = p; // q in f_1, p in f_1
    }
    return u = t; // u in f, t in f
}
```

```
f: // function scope f
   // t in f, x in f
   t = x
   // p in f_1, a in global
   p@f_1 = a@glb
   // t in f_1, hides t in f
   t@f_1 = 4
   // p in f_1_1, hides p in f_1
   p@f_1_1 = 5
   // q in f_1, p in f_1
   q@f_1 = p@f_1
   // u in f, t in f
   u = t
```

ST.glb: ST.glb.parent = null

a	int	global	4	0	null
f	int → int				
		func	0	0	ST.f

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f

p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1

p	int	local	4	0	null

Columns: Name, Type, Category, Size, Offset, & Symtab

Grammar and Parsing for this example is discussed with the Parse Tree in 3-Address Code Generation

Nested Blocks Flattened

```
f: // function scope f
   // t in f, x in f
   t = x
   // p in f_1, a in global
   p@f_1 = a@glb
   // t in f_1, hides t in f
   t@f_1 = 4
   // p in f_1_1, hides p in f_1
   p@f_1_1 = 5
   // q in f_1, p in f_1
   q@f_1 = p@f_1
   // u in f, t in f
   u = t
```

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f

p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1

p	int	local	4	0	null

Columns: Name, Type, Category, Size, Offset, & Symtab

```
f: // function scope f
   // t in f, x in f
   t = x
   // p in f_1, a in global
   p#1 = a@glb  // p@f_1
   // t in f_1, hides t in f
   t#3 = 4      // t@f_1
   // p in f_1_1, hides p in f_1
   p#4 = 5      // p@f_1_1
   // q in f_1, p in f_1
   q#2 = p#1    // q@f_1, p@f_1
   // u in f, t in f
   u = t
```

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
p#1	int	blk-local	4	0	null
q#2	int	blk-local	4	4	null
t#3	int	blk-local	4	8	null
p#4	int	blk-local	4	0	null

Example: Nested Blocks: main()

```
_DATA    SEGMENT
COMM     _a:DWORD
_DATA    ENDS
PUBLIC     _f
; Function compile flags: /Odtp /RTCsu
_TEXT    SEGMENT
_p$1 = -24 ; size = 4 // p#4
_t$2 = -20 ; size = 4 // t#3
_q$3 = -16 ; size = 4 // q#2
_p$4 = -12 ; size = 4 // p#1
_u$ = -8   ; size = 4
_t$ = -4   ; size = 4
_x$ = 8    ; size = 4
_f     PROC

; 2    : int f(int x) { // function scope f

    push   ebp
    mov    ebp, esp
    sub    esp, 24          ; 00000018H
    mov    eax, -858993460  ; ccccccccH
    mov    DWORD PTR [ebp-24], eax
    mov    DWORD PTR [ebp-20], eax
    mov    DWORD PTR [ebp-16], eax
    mov    DWORD PTR [ebp-12], eax
    mov    DWORD PTR [ebp-8], eax
    mov    DWORD PTR [ebp-4], eax
```

```
; 3    : int t, u;
; 4    : t = x; // t in f, x in f

    mov    eax, DWORD PTR _x$[ebp]
    mov    DWORD PTR _t$[ebp], eax

; 5    : { // un-named block scope f_1
; 6    :   int p, q, t;
; 7    :   p = a; // p in f_1, a in global

    mov    ecx, DWORD PTR _a
    mov    DWORD PTR _p$4[ebp], ecx

; 8    :   t = 4; // t in f_1, hides t in f

    mov    DWORD PTR _t$2[ebp], 4

; 9    :   { // un - named block scope f_1_1
; 10   :     int p;
; 11   :     p = 5; // p in f_1_1, hides p in f_1

    mov    DWORD PTR _p$1[ebp], 5

; 12   :   }
; 13   :   q = p; // q in f_1, p in f_1

    mov    edx, DWORD PTR _p$4[ebp]
    mov    DWORD PTR _q$3[ebp], edx
```

Nested Blocks: main()

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
; 14   :   }
; 15   :   return u = t; // u in f, t in f

    mov    eax, DWORD PTR _t$[ebp]
    mov    DWORD PTR _u$[ebp], eax
    mov    eax, DWORD PTR _u$[ebp]

; 16   : }

    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
_TEXT    ENDS
```

Example : Global & Function Scope: main() & add(): Source & TAC

```
int x, ar[2][3], y;
int add(int x, int y);
double a, b;
int add(int x, int y) {
    int t;
    t = x + y;
    return t;
}
void main() {
    int c;
    x = 1;
    y = ar[x][x];
    c = add(x, y);
    return;
}
```

```
add:    t#1 = x + y
        t = t#1
        return t

main:   t#1 = 1
        x = t#1
        t#2 = x * 12
        t#3 = x * 4
        t#4 = t#2 + t#3
        y = ar[t#4]
        param x
        param y
        c = call add, 2
        return
```

ST.glb: ST.glb.parent = null

x	int	global	4	0	null
ar	array(2, array(3, int))				
		global	24	4	null
y	int	global	4	28	null
add	int × int → int				
		func	0	32	ST.add()
a	double	global	8	32	null
b	double	global	8	40	null
main	void → void				
		func	0	48	ST.main()

ST.add(): ST.add.parent = ST.glb

x	int	param	4	0
y	int	param	4	4
t	int	local	4	8
t#1	int	temp	4	12

ST.main(): ST.main.parent = ST.glb

c	int	local	4	0
t#1	int	temp	4	4
t#2	int	temp	4	8
t#3	int	temp	4	12
t#4	int	temp	4	16

Columns: Name, Type, Category, Size, Offset, & Symtab

Grammar and Parsing for this example is discussed with the Parse Tree in 3-Address Code Generation

Example: Global & Function Scope: main()

```
_DATA    SEGMENT
COMM     _x:DWORD
COMM     _ar:DWORD:06H // 4 * 6 = 24
COMM     _y:DWORD
COMM     _a:QWORD
COMM     _b:QWORD
_DATA    ENDS
PUBLIC     _add
PUBLIC     _main
; Function compile flags: /Odtp /RTCsu
_TEXT    SEGMENT
_c$ = -4       ; size = 4
_main    PROC

; 9    : void main() {

    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR [ebp-4], -858993460
           ; ccccccccH

; 10   :     int c;
; 11   :     x = 1;

    mov    DWORD PTR _x, 1

; 12   :     y = ar[x][x];

    imul   eax, DWORD PTR _x, 12
    mov    ecx, DWORD PTR _x
    mov    edx, DWORD PTR _ar[eax+ecx*4]
    mov    DWORD PTR _y, edx

; 13   :     c = add(x, y);

    mov    eax, DWORD PTR _y
    push   eax
    mov    ecx, DWORD PTR _x
    push   ecx
    call   _add
    add    esp, 8
    mov    DWORD PTR _c$[ebp], eax

; 14   :     return;
; 15   : }

    xor    eax, eax
    add    esp, 4
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Example: Global & Function Scope: add()

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

```
; Function compile flags: /Odtp /RTCsu
_TEXT    SEGMENT
_t$ = -4      ; size = 4
_x$ = 8       ; size = 4
_y$ = 12      ; size = 4
_add    PROC

; 4    : int add(int x, int y) {

    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR [ebp-4], -858993460
           ; ccccccccH

; 5    :     int t;
; 6    :     t = x + y;

    mov    eax, DWORD PTR _x$[ebp]
    add    eax, DWORD PTR _y$[ebp]
    mov    DWORD PTR _t$[ebp], eax
```

```
; 7    :     return t;

    mov    eax, DWORD PTR _t$[ebp]

; 8    : }

    mov    esp, ebp
    pop    ebp
    ret    0
_add    ENDP
_TEXT    ENDS
```

Example:Global, Extern & Local Static Data

```
// File Main.c
extern int n;
int Sum(int x) {
    static int lclStcSum = 0;

    lclStcSum += x;
    return lclStcSum;
}
int sum = -1;
void main() {
    int a = n;

    Sum(a);
    a *= a;
    sum = Sum(a);
    return;
}

// File Global.c
int n = 5;
```

```
      lclStcSum = 0
Sum:  lclStcSum = lclStcSum + x
      return lclStcSum

        sum = -1
main:   a = glb_n
        param a
        call Sum, 1
        a = a * a
        param a
        sum = call Sum, 1
        return
```

ST.glb (Main.c)

n	int	extern	4	0
Sum	int → int	func	0	4
sum	int	global	4	0
main	void → void	func	0	8

ST.glb (Global.c)

n	int	global	4	0

ST.Sum()

x	int	param	4	0
lclStcSum	int	static	4	4

ST.main()

a	int	local	4	0

Columns are: Name, Type, Category, Size, & Offset

main()

```
PUBLIC    _sum // Global int sum;
_BSS    SEGMENT
?1clStcSum@?1??Sum@@9@9 DD 01H DUP (?)
    ; `Sum'::`2'::1clStcSum // int lclStcSum = 0;
_BSS    ENDS
_DATA    SEGMENT
_sum    DD    0ffffffffH // int sum = -1;
_DATA    ENDS
PUBLIC    _Sum
PUBLIC    _main
EXTRN    _n:DWORD // extern int n;
; Function compile flags: /Odtp /RTCsu
; File ..\main.c
_TEXT    SEGMENT
_a$ = -4      ; size = 4
_main    PROC

; 13   : void main() {
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR [ebp-4], -858993460
    ; ccccccccH

; 14   :     int a = n;
    mov    eax, DWORD PTR _n
    mov    DWORD PTR _a$[ebp], eax
; 15   :
```

```
; 16   :     Sum(a);
    mov    ecx, DWORD PTR _a$[ebp]
    push   ecx
    call   _Sum
    add    esp, 4

; 17   :     a *= a;
    mov    edx, DWORD PTR _a$[ebp]
    imul   edx, DWORD PTR _a$[ebp]
    mov    DWORD PTR _a$[ebp], edx

; 18   :     sum = Sum(a);
    mov    eax, DWORD PTR _a$[ebp]
    push   eax
    call   _Sum
    add    esp, 4
    mov    DWORD PTR _sum, eax

; 19   :     return;
; 20   : }
    xor    eax, eax
    add    esp, 4
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Sum()

```
; Function compile flags: /Odtp /RTCsu
; File ..\main.c
_TEXT    SEGMENT
_x$ = 8      ; size = 4
_Sum    PROC

; 4    : {

    push   ebp
    mov    ebp, esp

; 5    :     static int lclStcSum = 0;
; 6    :
; 7    :     lclStcSum += x;

    mov    eax, DWORD PTR ?lclStcSum@?1??Sum@@9@9
    add    eax, DWORD PTR _x$[ebp]
    mov    DWORD PTR ?lclStcSum@?1??Sum@@9@9, eax

; 8    :     return lclStcSum;

    mov    eax, DWORD PTR ?lclStcSum@?1??Sum@@9@9

; 9    : }

    pop    ebp
    ret    0
_Sum    ENDP
_TEXT    ENDS
```

```
    TITLE    $HOME\Global.c
PUBLIC    _n // int n;
_DATA    SEGMENT
_n    DD    05H // int n = 5;
_DATA    ENDS
END
```

Example: Binary Search

```
int bs(int a[], int l,
       int r, int v) {
    while (l <= r) {
        int m = (l + r) / 2;
        if (a[m] == v)
            return m;
        else
            if (a[m] > v)
                r = m - 1;
            else
                l = m + 1;
    }
    return -1;
}
```

```
100: if l <= r goto 102
101: goto 121
102: t1 = l + r
103: t2 = t1 / 2
104: m = t2
105: t3 = m * 4
106: t4 = a[t3]
107: if t4 == v goto 109
108: goto 111
109: return m
110: goto 100
```

```
111: t5 = m * 4
112: t6 = a[t5]
113: if t6 > v goto 115
114: goto 118
115: t7 = m - 1
116: r = t7
117: goto 100
118: t8 = m + 1
119: l = t8
120: goto 100
121: t9 = -1
122: return t9
```

ST.glb

bs	array(*, int) × int × int × int → int	
	func 0	0

Columns: Name, Type, Category, Size, & Offset

Temporary variables are numbered in the function scope – the effect of the respective block scope in the numbering is not considered. Hence, we show only a flattened symbol table

ST.bs()

a	array(*, int)	param	4	+16
l	int	param	4	+12
r	int	param	4	+8
r	int	param	4	+4
m	int	local	4	0
t1	int	temp	4	−4
t2	int	temp	4	−8
t3	int	temp	4	−12
t4	int	temp	4	−16
t5	int	temp	4	−20
t6	int	temp	4	−24
t7	int	temp	4	−28
t8	int	temp	4	−32
t9	int	temp	4	−36

Example: Transpose

```
int main() {
    int a[3][3];
    int i, j;
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < i; ++j) {
            int t;
            t = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = t;
        }
    }
    return;
}
```

ST.glb

main	void → void	func

ST.main()

a	array(3, array(3, int))			
		param	4	0
i	int	local	4	–4
j	int	local	4	–8
t01	int	temp	4	–12
t02	int	temp	4	–16
t03	int	temp	4	–20
t04	int	temp	4	–24
t05	int	temp	4	–28
t06	int	temp	4	–32
t07	int	temp	4	–36

```
100: t01 = 0
101: i = t01
102: t02 = 3
103: if i < t02 goto 108
104: goto 134
105: t03 = i + 1
106: i = t03
107: goto 103
108: t04 = 0
109: j = t04
110: if j < i goto 115
111: goto 105
112: t05 = j + 1
113: j = t05
114: goto 110
115: t06 = 12 * i
116: t07 = 4 * j
117: t08 = t06 + t07
```

ST.main()

t08	int	temp	4	–40
t09	int	temp	4	–44
t10	int	temp	4	–48
t11	int	temp	4	–52
t12	int	temp	4	–56
t13	int	temp	4	–60
t14	int	temp	4	–64
t15	int	temp	4	–68
t16	int	temp	4	–72
t17	int	temp	4	–76
t18	int	temp	4	–80
t19	int	temp	4	–84

```
118: t09 = a[t08]
119: t = t09
120: t10 = 12 * i
121: t11 = 4 * j
122: t12 = t10 + t11
123: t13 = 12 * j
124: t14 = 4 * i
125: t15 = t13 + t14
126: t16 = a[t15]
127: a[t12] = t16
128: t17 = 12 * j
129: t18 = 4 * i
130: t19 = t17 + t18
131: a[t19] = t
132: goto 112
133: goto 105
134: return
```

Module Summary

Module 04
Das
Obj. & Outline
Memory
Function Call
Symbol Table
Activation Record
x86 Assembly
Debug Build
Release Build
Decode ASM
Properties
Decl., Defn. & Init.
Scope & Binding
Storage Class
Address & Value
Lifetime
Param & Return
Array
Scopes
Nested Blocks
Global / Static
Summary

- Understood the Run-Time Environment for Program Execution comprising Storage Organization and Properties of Symbols
- Understood Symbol Tables, Activation Records (Stack Frames), their interrelationships in the context of Parameter passing and return value
- Understood Binding, Layout and Scopes
- Understood the translation of `int` data type, functions, arrays, and various types of scopes for run-time management