Compilers (CS31003)

Lecture 26-27

Properties of a Symbol

- A symbol has multiple Properties based on its context
  - Binding: The physical memory address of the symbol

```
// Symbol Name = "sum", Symbol Type = "int"
// Symbol Binding = &sum // Address of sum
int sum;
```

- 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);
    return 0;
}
a = 10 // Value of 'a'
&a = 0x7ffe7be8ad9c // Address or binding of 'a'
```

- 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

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

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*

Activation Record

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).

Quick Sort – an example

* m
* r, q(1,9)
* p(1,9), q(1,3), q(5,9)
* p(1,3), q(1,0), q(2,3), p(5,9), q(5,5), q(7,9)
* …..

* Operation on stack

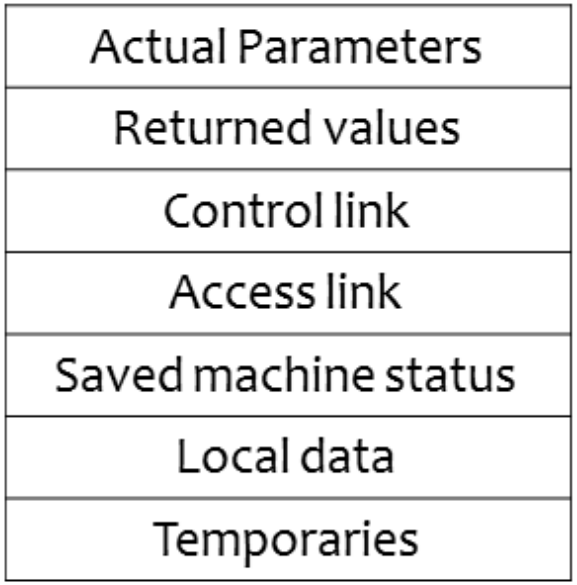
A general activation record

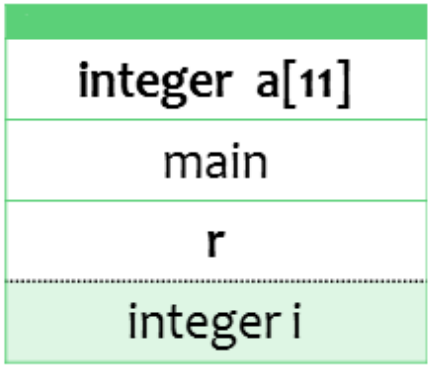
Actual Parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

integer a[11]
main

Quick sort

A general activation record





Quick sort

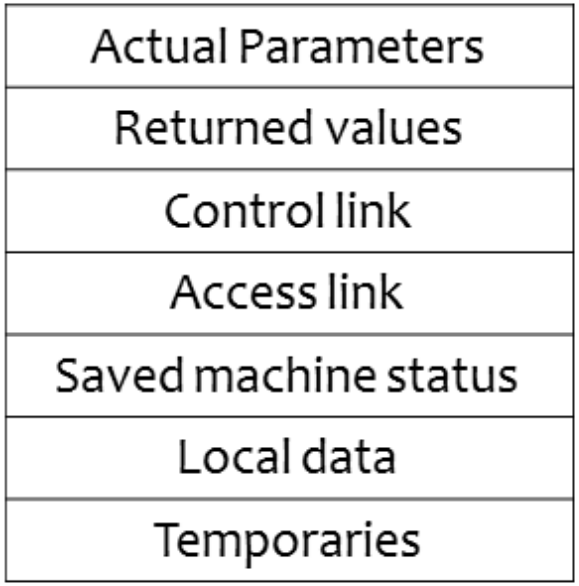
A general activation record

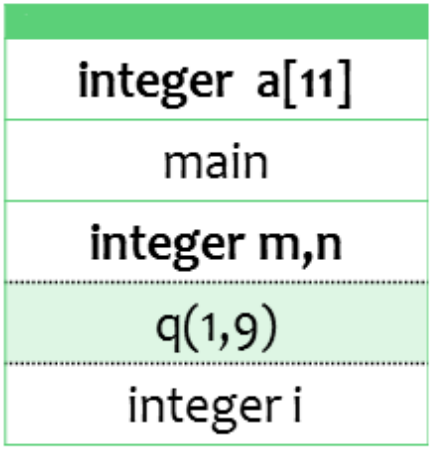
Actual Parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

integer a[11]
main

Quick sort

A general activation record





Quick sort

A general activation record

Actual Parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

integer a[11]
main
integer m,n
q(1,9)
integer i
integer m,n
q(1,3)
integer i

Quick sort

Calling & Return Sequences

- **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.

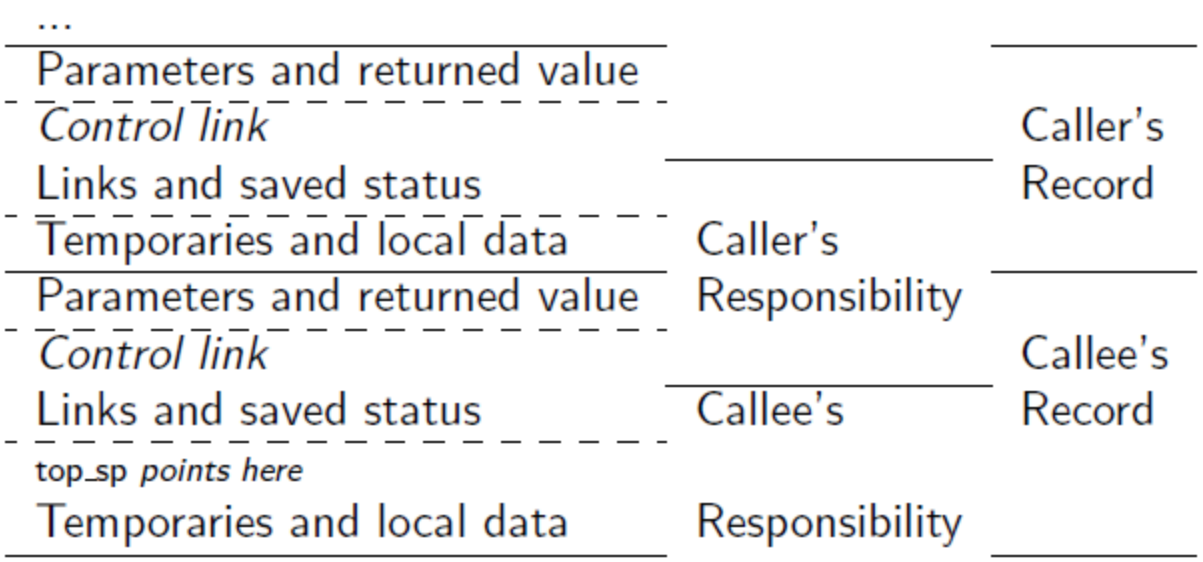
Calling Sequence

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to move past the caller's local data, temporaries and the callee's parameters and status fields.
3. The callee saves the register values and the other status information.
4. The callee initializes its local data and begins execution.

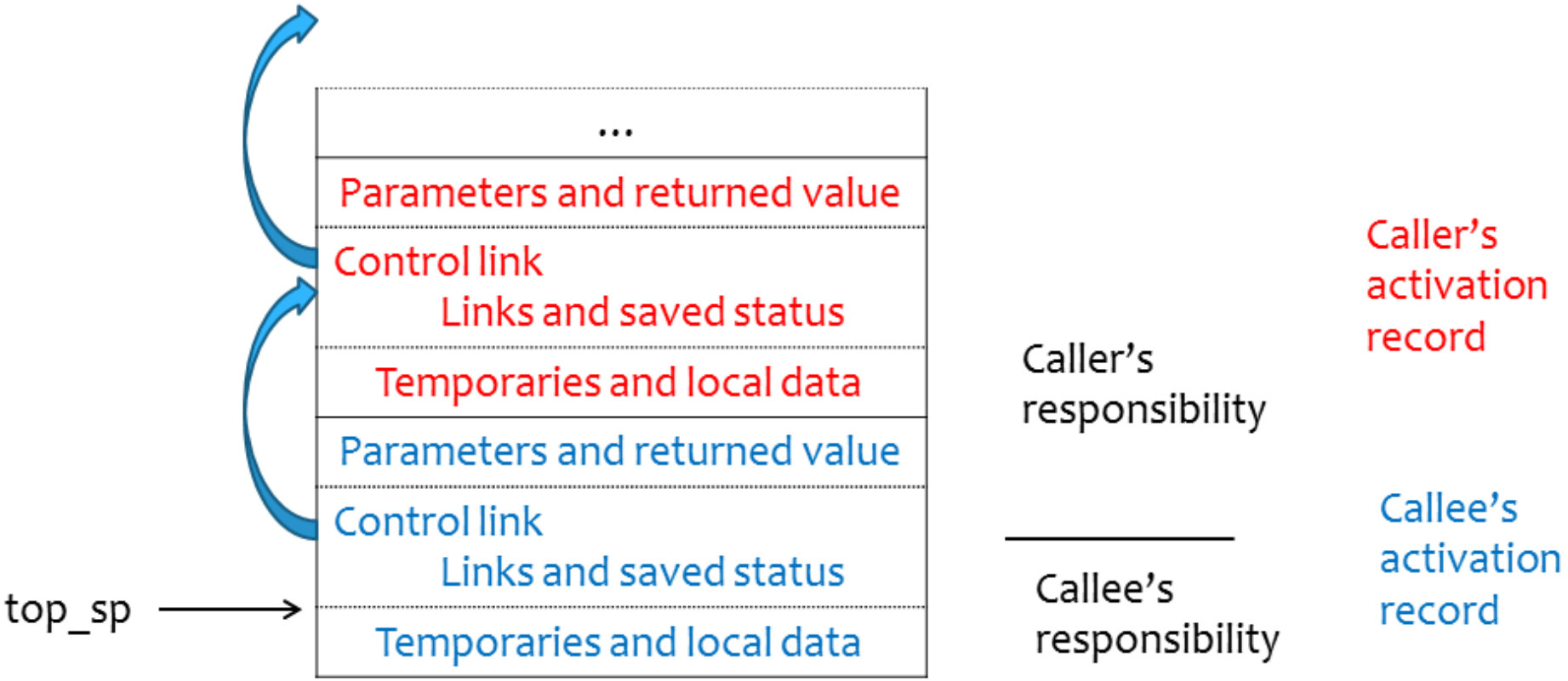
Return Sequence

1. The callee places the return value next to the parameters.
2. Using machine status field callee restores *top_sp*, registers and then branches to the return address that the caller placed in the status field.
3. Caller uses the return value as it knows where it is relative to the *top_sp*.

Calling & Return Sequences



Calling Sequence



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				
add	int × int → int	func	0	0
main	int × array(*, char*) → void	func	0	0

ST.add()				
y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0

ST.main()				
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

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

AR of main()

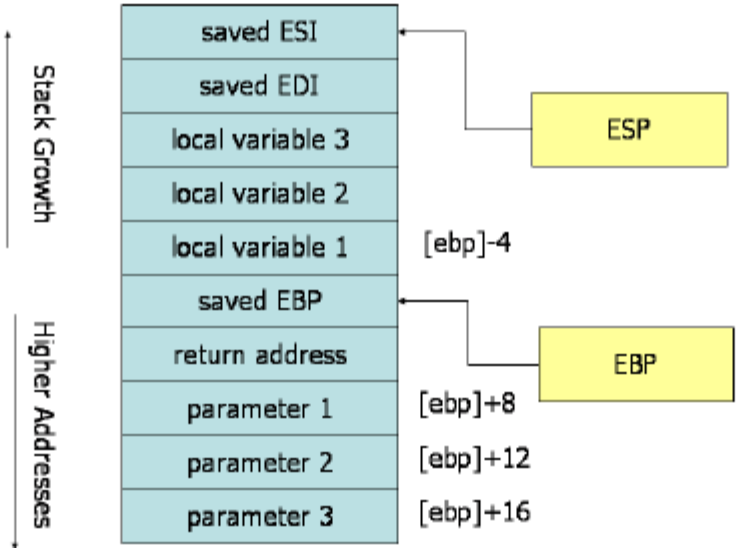
1012	−12	c
1016	−8	b = 3
1020	−4	a = 2
1024		ebp
1028		RA
1032	+8	argc
1036	+12	argv

ebp = 1024

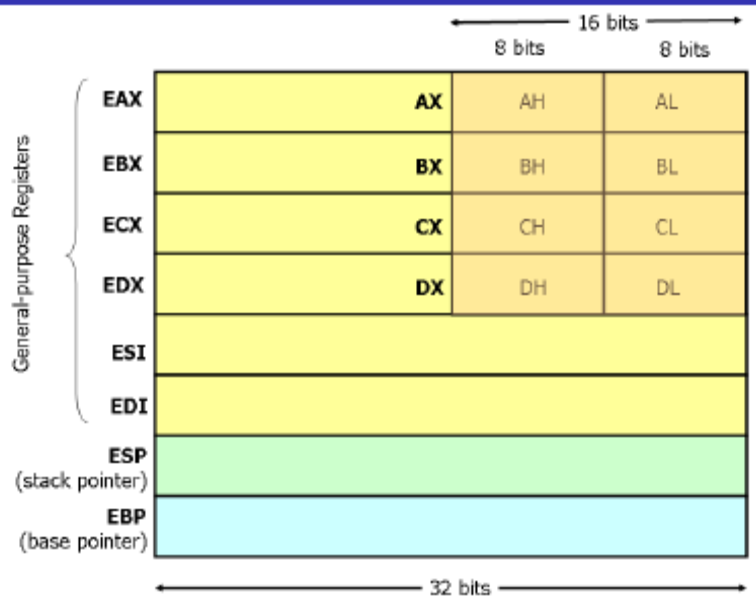
AR of add()

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

ebp = 996



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: http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

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$[ebp]				2	ecx =	[1020] = 2
0x30	push ecx	1004				x = [1004] =	ecx = 2
0x31	call _add	1000				RA = [1000] = epi = _add (0x50)	epi = 0x36
	; On return	1004		5	2	epi =	[1000]
0x36	add esp, 8	1012					
0x39	mov DWORD PTR _c$[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					

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

Activation Record of main()

Offset	Addr.	Stack	Description
	784	`edi`	Saved registers
	788	`esi`	
	792	`ebx`	
	796	`0xcccccccc`	Buffer for Edit & Continue (192 bytes)
	...	`0xcccccccc`	
	...	`0xcccccccc`	
	988	`0xcccccccc`	Local data w/ buffer
−32	992	`c`	
	996	`0xcccccccc`	
	1000	`0xcccccccc`	
−20	1004	`b = 3`	
	1008	`0xcccccccc`	
	1012	`0xcccccccc`	
−8	1016	`a = 2`	
	1020	`0xcccccccc`	
ebp →	1024	`ebp (of Caller of main())`	Control link
	1028	`Return Address`	RA (Caller saved)
+8	1032	`argc`	Params (Caller saved)
+12	1036	`argv`	

Activation Record of add()

Offset	Addr.	Stack	Description
	552	`edi`	Saved registers
	556	`esi`	
	560	`ebx`	
	564	`0xcccccccc`	Buffer for Edit & Continue (192 bytes)
	...	`0xcccccccc`	
	...	`0xcccccccc`	
	756	`0xcccccccc`	Local data w/ buffer
−8	760	`z = 5`	
	764	`0xcccccccc`	
ebp →	768	`ebp (of main()) = 1024`	Control link
	772	`Return Address`	RA (Caller saved)
+8	776	`ecx = 2: x`	Params (Caller saved)
+12	780	`eax = 3: y`	

Example: main() & d_add(): double type

```
double d_add(double x, double y) {
    double z;
    z = x + y;
    return z;
}
void main() {
    double a, b, c;
    a = 2.5;
    b = 3.4;
    c = d_add(a, b);
    return;
}
```

```
d_add:  z = x + y
        return z
main:   a = 2.5
        b = 3.4
        param a
        param b
        c = call d_add, 2
        return
```

ST.glb				
d_add	dbl × dbl → dbl	function	0	0
main	void → void	function	0	0
ST.d_add()				
x	dbl	param	8	0
y	dbl	param	8	16
z	dbl	local	8	24

ST.main()				
a	dbl	local	8	0
b	dbl	local	8	8
c	dbl	local	8	16

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

d_add(): double type

```
PUBLIC    _d_add
EXTRN     __fltused:DWORD
EXTRN     __RTC_Shutdown:PROC
EXTRN     __RTC_InitBase:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT
_z$ = -8 ; size = 8
_x$ = 8  ; size = 8
_y$ = 16 ; size = 8

; 1     : double d_add(double x, double y) {

    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR [ebp-8], 0xcccccccH
    mov    DWORD PTR [ebp-4], 0xcccccccH

; 2     :     double z;
; 3     :     z = x + y;

    fld    QWORD PTR _x$[ebp]
    fadd   QWORD PTR _y$[ebp]
    fstp   QWORD PTR _z$[ebp]

; 4     :     return z;

    fld    QWORD PTR _z$[ebp]
```

```
; 5     : }

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

- `QWORD PTR`: Quad Word Pointer – Refers to 8 consecutive bytes
- Uses FPU register stack for operations
- `fld`: Load Floating Point Value
- `fadd`: Adds the destination and source operands and stores the sum in the destination location
- `fstp`: Store Floating Point Value
- Return value (local variable `z`) passed through FPU register stack (`fld`)

main(): double type

```
PUBLIC    _main
EXTRN     __RTC_CheckEsp:PROC
CONST     SEGMENT
__real@400b333333333333 DQ
    0400b333333333333r    ; 3.4
__real@4004000000000000 DQ
    04004000000000000r    ; 2.5
CONST     ENDS
; Function compile flags: /Odtp /RTCsu
_TEXT     SEGMENT
_c$ = -24 ; size = 8
_b$ = -16 ; size = 8
_a$ = -8  ; size = 8
_main     PROC

; 6    : void main() {
    push   ebp
    mov    ebp, esp
    sub    esp, 24 ; 00000018H
    mov    eax, 0xccccccccH
    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

; 7    :     double a, b, c;
; 8    :     a = 2.5;
    fld    QWORD PTR __real@4004000000000000
    fstp   QWORD PTR _a$[ebp]
; 9    :     b = 3.4;
    fld    QWORD PTR __real@400b333333333333
    fstp   QWORD PTR _b$[ebp]
; 10   :     c = d_add(a, b);
    sub    esp, 8 ; push b
    fld    QWORD PTR _b$[ebp]
    fstp   QWORD PTR [esp]
    sub    esp, 8 ; push a
    fld    QWORD PTR _a$[ebp]
    fstp   QWORD PTR [esp]
    call   _d_add
    add    esp, 16 ; 00000010H - pop params
    fstp   QWORD PTR _c$[ebp]
; 11   :     return;
; 12   : }
    xor    eax, eax
    add    esp, 24 ; 00000018H
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
_main     ENDP
_TEXT     ENDS
```

- No `push` / `pop` for `QWORD` – using explicit manipulation of `esp` with load / store.
- Return value returned through FPU register stack (`fstp`)

ARs of main() and d_add(): double type

```
; Function compile flags: /Odtp /RTCsu
```

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

AR of main()

1000	−24	c
1004		5.9
1008	−16	b =
1012		3.4
1016	−8	a =
1020		2.5
1024		ebp
1028		RA

ebp = 1024

AR of d_add()

968	−4	z =
972		5.9
976		ebp = 1024
980		RA
984	+8	x
988		2.5
992	+16	y
996		3.4

ebp = 976

Example: main() & swap()

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
    return;
}
void main() {
    int a = 1, b = 2;
    swap(&a, &b);
    return;
}
```

```
swap:   t = *x;
        *x = *y;
        *y = t;
        return
main:   a = 1
        b = 2
        t1 = &a
        t2 = &b
        param t1
        param t2
        call swap, 2
        return
```

ST.glb				
swap	int* × int* → void	func	0	0
main	void → void	func	0	0
ST.swap()				
y	int*	prm	4	0
x	int*	prm	4	4
t	int	lcl	4	8

ST.main()				
a	int	lcl	4	0
b	int	lcl	4	4
t1	int*	lcl	4	8
t2	int*	lcl	4	12

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

ARs of main() and swap()

```
; Function compile flags: /Odtp /RTCsu
```

980	−4	t = 1
ebp → 984		ebp = 1024
988		RA
992	+8	ecx = 1016: x
996	+12	eax = 1004: y
1000		0xcccccccc
1004	−20	b = 2
1008		0xcccccccc
1012		0xcccccccc
1016	−8	a = 1
1020		0xcccccccc
ebp → 1024		ebp
1028		RA

ebp = 1024

Homework: Fibonacci Series