Module 06: CS31003: Compilers: Run-time Environments

Pralay Mitra
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

pralay@cse.iitkgp.ac.in
ppd@cse.iitkgp.ac.in

August 06 (Lab), October 01 & 14, 2019

Module Objectives

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

Module Outline

- Binding Protocol
- Memory Organization
- Symbol Table, Activation Record, Stack Frame
- Function Call Protocol
- Optimization & IO
- Handling various types and scopes
    - `double`
    - Pointer
    - `struct`
    - Array
    - Function Pointer
    - Nested Blocks
    - Global / Static
    - Mixed

Lab Focus

- Binding Protocol
- Memory Organization
- Symbol Table, Activation Record, Stack Frame
- Function Call Protocol (`int`)
- Optimization & IO

Symbol Table to Activation Record

3-Address Code	Target Code
Symbol Table (Function) ● 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.	**Activation Record** ● 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

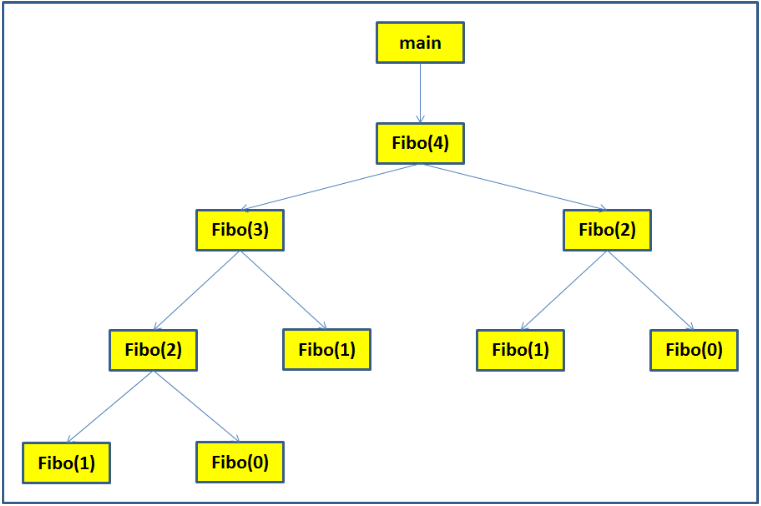
Actual Params	The actual parameters used by the calling procedure (often placed in registers for greater efficiency).
Returned Values	Space for the return value of the called function (often placed in a register for efficiency). Not needed for void type.
Return Address	The return address (value of the 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).

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 – Fibo



Activation Records – Fibo

Prm	
RV	...
Lnk	null
Prm	4
RV	...
Lnk	m()
Prm	3
RV	...
Lnk	f(4)
Prm	2
RV	...
Lnk	f(3)
Prm	1
RV	...
Lnk	f(2)

Prm	
RV	...
Lnk	null
Prm	4
RV	...
Lnk	m()
Prm	3
RV	...
Lnk	f(4)
Prm	2
RV	...
Lnk	f(3)
Prm	0
RV	...
Lnk	f(2)

Prm	
RV	...
Lnk	null
Prm	4
RV	...
Lnk	m()
Prm	3
RV	...
Lnk	f(4)
Prm	1
RV	...
Lnk	f(3)

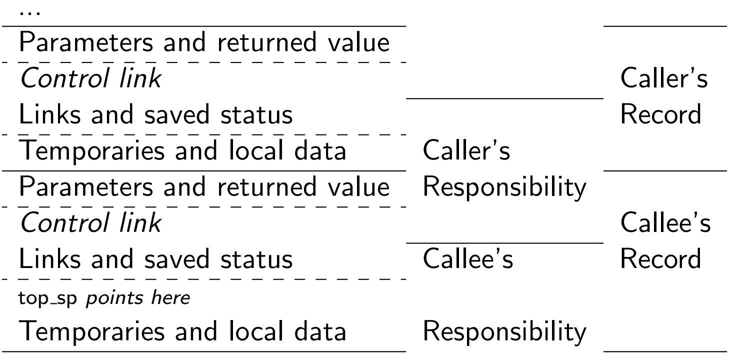
Prm	
RV	...
Lnk	null
Prm	4
RV	...
Lnk	m()
Prm	2
RV	...
Lnk	f(4)
Prm	1
RV	...
Lnk	f(2)

Prm	
RV	...
Lnk	null
Prm	4
RV	...
Lnk	m()
Prm	2
RV	...
Lnk	f(4)
Prm	0
RV	...
Lnk	f(2)

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 & Return Sequences



...

Parameters and returned value

Control link

Links and saved status

Temporaries and local data

Parameters and returned value

Control link

Links and saved status

top_sp *points here*

Temporaries and local data

Caller's Responsibility

Callee's Responsibility

Caller's Record

Callee's Record

Calling & Return Sequences

- **Calling Sequences**:
  The calling sequence and its division between caller and callee is as follows:
  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 top_sp to the position shown – just past the caller's local data and temporaries and the callee's parameters and status fields.
  3. The callee saves the register values and other status information.
  4. The callee initializes its local data and begins execution.

Calling & Return Sequences

- **Return Sequence**:
  A suitable, corresponding return sequence is:
  1. The callee places the return value next to the parameters.
  2. 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.
  3. 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.

AR in VS: Function

Function Call and `int` Data Type

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				
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
t1	int	temp	4	−4

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
t1	int	temp	4	−12
t2	int	temp	4	−16

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

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

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

```
PUBLIC    _main
EXTRN    __RTC_CheckEsp:PROC
; Function compile flags: /Odtp /RTCsu
_TEXT    SEGMENT
_c$ = -12     ; size = 4
_b$ = -8      ; size = 4
_a$ = -4      ; size = 4
_argc$ = 8    ; size = 4
_argv$ = 12   ; size = 4
_main    PROC

; 6    : void main(int argc, char *argv[]) {

    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

; 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   : }

    xor    eax, eax
    add    esp, 12 ; 0000000cH
    cmp    ebp, esp
    call   __RTC_CheckEsp
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_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$IMZ    ENDS
_TEXT    SEGMENT
_z$ = -4      ; size = 4
_x$ = 8       ; size = 4
_y$ = 12      ; size = 4
_add    PROC

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

    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR [ebp-4], 0xccccccccH

; 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     : }

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

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

Run-Time Error Checking on Stack Frame in Visual Studio

- **Enable Stack Frame Run-Time Error Checking** (/GZ)[1]: Used to enable and disable the run-time error checks feature (prefer /RTC). With this option, uninitialized variables are automatically assigned to `0xccccccccH` (at byte level). It is distinct and easy to identify if the program ends up using an uninitialized variable. Interestingly, in x86 assembly, the op-code `0xcc` is the `int 3` op-code, which is the software breakpoint interrupt. So, if you ever try to execute code in uninitialized memory that has been filled with that fill value, you'll immediately hit a breakpoint, and the operating system will let you attach a debugger (or kill the process).

[1] Source: http://msdn.microsoft.com/en-us/library/hddybs7t.aspx, http://stackoverflow.com/questions/370195/when-and-why-will-an-os-initialise-memory-to-0xcd-0xdd-etc-on-malloc-free-new

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

AR of main()

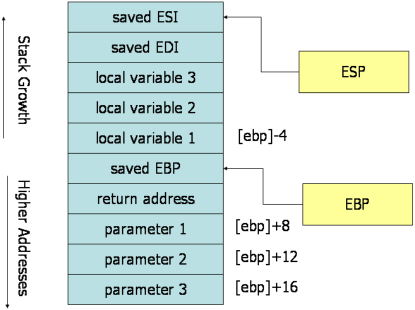
1012	−12	`c`
1016	−8	`b = 3`
1020	−4	`a = 2`
1024		`ebp`
1028		`RA`
1032	+8	`argc`
1036	+12	`argv`

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



Stack Growth

Higher Addresses

saved ESI

saved EDI

local variable 3

local variable 2

local variable 1

saved EBP

return address

parameter 1

parameter 2

parameter 3

ESP

EBP

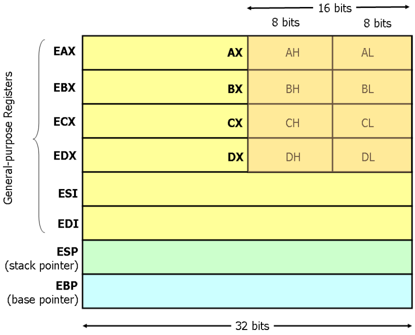
[ebp]-4

[ebp]+8

[ebp]+12

[ebp]+16

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

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

```
PUBLIC    _main
; Function compile flags: /Odtp /RTCsu /ZI
_TEXT     SEGMENT
_c$ = -32     ; size = 4
_b$ = -20     ; size = 4
_a$ = -8      ; size = 4
_argc$ = 8    ; size = 4
_argv$ = 12   ; size = 4
_main     PROC ; COMDAT

; 6    : void main(int argc, char *argv[]) {

    // PROLOGUE of _main
    // Save the ebp of the caller of _main
    push   ebp
    // Set the ebp of _main
    mov    ebp, esp
    // Create space for local and temporary in the AR of _main
    sub    esp, 228                ; 000000e4H = 32 + 4 + 192
    // Save machine status
    push   ebx
    push   esi
    push   edi
    // Fill the fields of the AR with 0xccccccccH
    lea    edi, DWORD PTR [ebp-228]
    mov    ecx, 57                 ; 00000039H = 228/4
    mov    eax, -858993460         ; ccccccccH
    rep stosd                      ; Store String (doubleword) from eax
                                   ; at edi repeating ecx times
```

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

```
; 7    :    int a, b, c;
; 8    :    a = 2;

  // Copy 2 in DWORD starting at _a$[ebp]
  mov    DWORD PTR _a$[ebp], 2

; 9    :    b = 3;

  // Copy 3 in DWORD starting at _b$[ebp]
  mov    DWORD PTR _b$[ebp], 3

; 10   :    c = add(a, b);

  // Push parameters in the AR of _add
  // Note the right-to-left order
  mov    eax, DWORD PTR _b$[ebp]
  push   eax  ; Value of b is passed
  mov    ecx, DWORD PTR _a$[ebp]
  push   ecx  ; Value of a is passed
  // Return Address gets pushed
  call   _add
  // Re-adjust esp on return from _add
  add    esp, 8 ; pop params
  // Copy return value from eax
  mov    DWORD PTR _c$[ebp], eax

; 11   :    return;
; 12   : }
```

```
  // EPILOGUE of _main
  xor    eax, eax
  // Restore machine status
  pop    edi
  pop    esi
  pop    ebx
  // Annul the space for local and
  // temporary in the AR of _main
  add    esp, 228 ; 000000e4H
  // Check the correctness of esp
  cmp    ebp, esp
  call   __RTC_CheckEsp
  mov    esp, ebp
  // Restore the ebp of the caller
  // of _main
  pop    ebp
  // Return type void -
  // nothing to return
  ret    0
_main    ENDP
_TEXT    ENDS
```

- `DWORD PTR`: Double Word Pointer – Refers to 4 consecutive bytes
- `add()` returns `int` value through `eax`
- C++ style comments added for better understanding

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`	

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

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

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

    // PROLOGUE of _add
    // Save the ebp of the caller of _add (_main)
    push    ebp
    // Set the ebp of _add
    mov     ebp, esp
    // Create space for local and temporary in the AR of _add
    sub     esp, 204                    ; 000000ccH = 8 + 4 + 192
    // Save machine status
    push    ebx
    push    esi
    push    edi
    // Fill the fields of the AR with 0xccccccccH
    lea     edi, DWORD PTR [ebp-204]
    mov     ecx, 51                     ; 00000033H = 204/4
    mov     eax, -858993460             ; ccccccccH
    rep stosd
```

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

```
; 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    : }

    // EPILOGUE of _add
    // Restore machine status
    pop    edi
    pop    esi
    pop    ebx
    // Annul the space for local and
    // temporary in the AR of _add
    mov    esp, ebp
    // Restore the ebp of the caller
    // of _add (_main)
    pop    ebp
    // Return through eax -
    // no direct return
    ret    0
_add    ENDP
_TEXT    ENDS
```

- add() returns int value through eax

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`	

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

Loc.	Code	esp	ebp	eax	ecx	Stack / Reg.	Value
		1028	?	?	?		
0x00	push ebp	1024				[1024] =	ebp
0x01	mov ebp, esp		1024				
0x03	sub esp, 228	796					
0x09	push ebx	792				[792] =	ebx
0x0a	push esi	788				[788] =	esi
0x0b	push edi	784				[784] =	edi
0x0c	lea edi, [ebp-228]					edi =	796
0x12	mov ecx, 57				57	ecx =	57
0x17	mov eax, 0xccccccccH ;#fill			#fill		eax =	#fill
0x1c	rep stosd					[796:1023] =	#fill
0x1e	mov _a$[ebp], 2 ; _a$=-8					a = [1016] =	2
0x25	mov _b$[ebp], 3 ; _b$=-20					b = [1004] =	3
0x2c	mov eax, _b$[ebp]					eax =	[1004] = 3
0x2f	push eax	780		3		[780] =	eax = 3
0x30	mov ecx, _a$[ebp]				2	ecx =	[1016] = 2
0x33	push ecx	776				[776] =	ecx = 2
0x34	call _add	772				[772] =	epi = 0x39
					epi =	_add (0x50)	
	; On return	776		5	51	epi =	[772]
0x39	add esp, 8	784					
0x3c	mov _c$[ebp], eax ; _c$=-32					c = [992] =	eax = 5
0x3f	xor eax, eax			0		eax =	0
0x41	pop edi	788				edi =	[784]
0x42	pop esi	792				esi =	[788]
0x43	pop ebx	796				ebx =	[792]
0x44	mov esp, ebp	1024					
0x46	pop ebp	1028	?			ebp =	[1024]
0x47	ret 0	1032					

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

Loc.	Code	esp	ebp	eax	ecx	Stack/Reg.	Value
		772	1024	3	2		
0x50	push ebp	768				[768] =	ebp
0x51	mov ebp, esp		768				
0x53	sub esp, 204	564					
0x59	push ebx	560				[560] =	ebx
0x5a	push esi	556				[556] =	esi
0x5b	push edi	552				[552] =	edi
0x5c	lea edi, [ebp-204]					edi =	564
0x62	mov ecx, 51				51	ecx =	51
0x67	mov eax, 0xccccccccH ;#fill			#fill		eax =	#fill
0x6c	rep stosd					[564:767] =	#fill
0x6e	mov eax, _x$[ebp] ;_x$=8			2		eax =	x = [776] = 2
0x71	add eax, _y$[ebp] ;_y$=12			5		eax =	eax+=y=[780]=3
0x74	mov _z$[ebp], eax ;_z$=-8					z = [760] =	eax = 5
0x77	mov eax, _z$[ebp]			5		eax =	z = [760] = 5
0x7a	pop edi	556				edi =	[552]
0x7b	pop esi	560				esi =	[556]
0x7c	pop ebx	564				ebx =	[560]
0x7d	mov esp, ebp	768					
0x7f	pop ebp	772	?			ebp =	[768]
0x80	ret 0	776				epi =	[772]

Notes on Stack Frame in Visual Studio

- **Debug Information Format – Edit + Continue** (/ZI)[2]: 192 are bytes allocated in the frame to support the Edit + Continue feature. It allows one to edit the code while a breakpoint is active and add local variables to a function.
- **Buffer Security Check** (/GS)[3]: Detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters. On functions that the compiler recognizes as subject to buffer overrun problems, the compiler allocates space on the stack before the return address. On function entry, the allocated space is loaded with a *security cookie* that is computed once at module load. On function exit, and during frame unwinding on 64-bit operating systems, a helper function is called to make sure that the value of the cookie is still the same. A different value indicates that an overwrite of the stack may have occurred. If a different value is detected, the process is terminated.

[2] Source: http://msdn.microsoft.com/en-us/library/958x11bc.aspx, http://stackoverflow.com/questions/3362872/explain-the-strange-assembly-of-empty-c-main-function-by-visual-c-compiler

[3] Source: http://msdn.microsoft.com/en-us/library/8dbf701c.aspx

I/O and Optimized Build

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
```

- Library function scanf called by convention
- lea used for address parameter in scanf

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    :
; 15    :      return;
; 16    : }

    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
```

- Library function `printf` called by convention
- 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   :
; 15   :     return;
; 16   : }

    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!