Linkage with C and C++

Object Files
Names and Visibility
Calling Conventions

Object Files

- Object files (Windows .obj, Linux .o) are an intermediate form of machine code that is not executable
  - These are inputs to a linker which links multiple modules into one executable program
- Object Files contain unresolved references to procedures or data located in other modules
  - When developing a program as a set of independent modules, all offsets in a segment are relative to the segment registers of that module
  - When several modules are combined the offsets have to be adjusted whenever segments are shared

Language Independent

- Object files are where the language disappears
  - The basic idea of object files is to allow programmers to write and assemble (or compile) individual pieces of programs and then to link them together to make the final program.
- For most languages you can work without ever being aware of the existence or presence of object files
- When building mixed-language programs the each language is used to create one or more object files which are then linked into a single executable
- This scheme permits you to mix "modules" written in different languages as long as you follow the proper rules of design and visibility of names.

Sharing Names

- Names or symbols are the “links” by which code in one object file refers to data or code in another object file
- Names can be public (published in the object file) or local
- Global or Public directives cause names to made available in the object file
- To refer to a name defined elsewhere, an extern directive is needed

Two Sides of the Same Coin

- The **extern** directive tells the compiler/assembler that a name is defined elsewhere. The cctual spelling of “extern" may vary
  - C extern
  - Pascal external
  - MASM EXTRN
  - NASM extern
- A Global (NASM) or PUBLIC (MASM) directive is used in a module whose names will be referenced by other module
  - Causes names to be exported to the obj file in a PUBDEF record (Public Names Definition Record)
  - These directive are pretty much peculiar to assembler
- All HLLs however support some syntactic mechanism by which public names can be exported to the .obj file

main4:1

```
%include "asm_io.inc"
segment .data
  sum     dd   0
segment .bss
  input   resd 1

segment .text
 global  _asm_main
 global  _asm_main
 extern get_int, print_sum
_asm_main:
   enter 0,0     ; setup routine
   pusha
```

sub4:1

```
%include "asm_io.inc"
segment .data
prompt db ") Enter an integer (0 to
  quit): ", 0

segment .bss
segment .text
global  get_int, print_sum
```

sub4:2

```
; get_int – prompt and read integer
; Parameters (in order pushed on stack)
;   number of input (at [ebp + 12])
;   address of word to store input into (at [ebp + 8])
segment .data
prompt db ") Enter an integer (0 to quit): ", 0
segment .text
get_int:
  push ebp
  mov ebp, esp
  mov eax, [ebp + 12]
  call print_int
  mov eax, prompt
  call print_string
  call read_int
  mov ebx, [ebp + 8]
  mov [ebx], eax        ; store input into memory
  pop ebp
  ret
```

sub4:3

```
; print_sum
; Parameter:
;   sum to print out (at [ebp+8])
; Note: destroys value of eax
segment .data
  result db "The sum is ", 0

segment .text
print_sum:
  push ebp
  mov ebp, esp
  mov eax, result
  call print_string

  mov eax, [ebp+8]
  call print_int
  call print_nl
  pop ebp
  ret
```

Interfacing Assembler with C

- In the following example we have
- foo.c
  - A C program that declares a global variable int foo
  - The C program calls a function bar, written in assembler, that modifies foo
  - The C program also refers a variable dvar defined in assembler
- Bar.asm
  - The assembler program refers to a variable foo defined in the C program
  - It defines a variable called dvar that is accessed from C
  - It defines a function void bar(void) that refers to the global variable foo

The C side of the coin

```
void bar(void);    /*  resolved by linker */
int foo;
extern int dvar; /* dvar is defined elsewhere */
/* foo is public because it is a global variable */
int main () {
   foo = 1;
   bar();
   printf("\nValue of foo = %i", foo);
   dvar *= foo + 1;
   printf("\nValue of dvar = %i", dvar);
   return 0;
}
```

The asm side of the coin

```
extern _foo    ; foo is defined elsewhere
global _bar, _dvar
segment .data
_dvar dd 123
segment .text
_bar:
  inc dword [_foo]
  ret
```

- Assemble and run

```
nasm -fwin32 bar.asm
cl foo.c bar.obj
foo
```

- Output

```
Value of foo = 2
Value of dvar = 369
```

Variations on a Theme

- Using the stack we don't have to make names visible across modules

```
segment .text
global _bar2
%define fooptr dword [ebp+8]
_bar2:
  push ebp           ; set up stack frame
  mov ebp, esp
  mov eax, fooptr    ; get reference var
  inc dword [eax]    ; compute with it
  mov eax, [eax]     ; return value in eax
  pop ebp
  ret
```

Variations on a Theme

- Here we pass a local (automatic) variable

```
int bar2(int*);   /* bar is defined elsewhere */
int main () {
  int foo, foo2;
  foo = 41;
  foo2 = bar2(&foo);
  printf("Value of foo2 = %i", foo2);
  return 0;
}
```

Using ESP

- If we don't use the stack we don't need a stack frame

```
segment .text
global _bar2
%define fooptr dword [esp+4]
_bar2:
  mov eax, fooptr    ; get reference var
  inc dword [eax]    ; compute with it
  mov eax, [eax]     ; return value in eax
  ret
```

Calling Conventions

- Calling conventions specify a number of items
  1. How are parameters passed to a function?
  2. Are parameters passed left to right or vice versa
  3. Who cleans up the stack?
  4. How are results from value-returning functions returned?
  5. What registers need to be preserved by a function?
  6. How are names decorated or mangled?
  7. Are names case-sensitive?

- Calling conventions are compiler and OS-specific
- We will discuss a few fairly general Windows conventions and then look at cdecl in Linux gcc

Parameter Order

- When calling func(a,b,x) we can push left-to-right or right-to-left
- Left to right

Left to right		Right-to-left
a	EBP+16	x
b	EBP+12	b
x	EBP+8	a
Return eip	EBP+4	Return eip
Caller's ebp	EBP	Caller's ebp

Parameter Order

- Many languages use left-to-right parameter pushing

  But many languages that allows variable length parameter lists OR optional parameters uses right to left pushing ("right pusher")

  In particular C and C++ are right-pushers

  Note that right pushing always leaves the leftmost (and known parameters) at known offsets from the base pointer

Stack Cleanup

- Most languages clean up the stack before returning by using the RET imm instruction
- C/C++ as usual are the exceptions:
  - The CALLER will clean up parameters the stack by using an ADD ESP, n instruction after the function call
- Again note that stack cleanup MUST be done by the caller if variable length parameter lists are permitted
  - Some languages handle variable length parameter lists using a "param array" – a pointer to a dynamic array of parameters

Returning Values from Functions

- Function return values for simple types are almost universal:

bytes	AL
words	AX
dwords	EAX (or DX:AX in 16 bits)
qwords	EDX:EAX
floats	ST(0) [top of x87 register stack]

- Note that the issue is not so much type as size
  - Both ints and pointers are returned in EAX
- For sizes other than those listed above, functions either
  - (A) return a pointer to a data structure
  - OR
  - (B) return a data structure on the stack.
- Usually small values less than 32 bits are zero or sign extended into eax

Preserving Registers

- The issue of which registers are to be preserved is very much compiler – specific
- Compilers follow such conventions internally and expect externally-defined functions to do the same
- Conventions vary between compilers even in the same language
  - To be language-independent you can preserve all registers except for EAX
- Failure to preserve registers can lead to crashes or even worse -- programs that behave incorrectly without crashing

Name Decoration and Mangling

- Many compilers add characters to names in their internal symbol tables
- When the characters are uniformly applied to all names, we call it "decoration"/
  - Most C compilers add a leading underscore (more to follow)…
- C++ compilers allow function overloading, where the same function name is used for several implementations that may differ in the type or order of their parameters and/or return types
- These compilers add parameter type and order information to the names in the symbol table. This process is called "name mangling"

Name Mangling Example

- Create dummy C++ programs with empty functions:

```
void test() {
}
void test(int) {
}
void test(float,double) {
}
```

- And compiler to assembler code
  - -S most C and C++ compilers
  - -FAs Microsoft C and C++

Output from cl.exe

```
PUBLIC?test@@YAXXZ                        ; test
?test@@YAXXZ PROC NEAR                    ; test
  pushebp
  mov ebp, esp
  pop ebp
  ret 0
?test@@YAXXZ ENDP                         ; test

PUBLIC?test@@YAXH@Z                       ; test
?test@@YAXH@Z PROC NEAR                   ; test
  pushebp
  mov ebp, esp
  pop ebp
  ret 0
?test@@YAXH@Z ENDP                        ; test

PUBLIC?test@@YAXMN@Z                      ; test
?test@@YAXMN@Z PROC NEAR                  ; test
  pushebp
  mov ebp, esp
  pop ebp
  ret 0
?test@@YAXMN@Z ENDP                       ; test
```

Output from Borland C++

```
@@test$qv   proc near
?live16385@0: ;      void test() {
  push      ebp
  mov       ebp,esp
  pop       ebp
  ret
@@test$qv   endp ;               }
@@test$qi   proc near
?live16386@0: ;      void test(int) {
  push      ebp
  mov       ebp,esp
  pop       ebp
  ret  ;         }
@@test$qi   endp
@@test$qfd  proc near
?live16387@0: ;      void test(float,double) {
  push      ebp
  mov       ebp,esp
  pop       ebp
  ret  ;         }
@@test$qfd  endp
```

More Examples

Compiler	void h(int)	void h(int, char)	void h(void)
Intel C++ 8.0 for Linux	_Z1hi	_Z1hic	_Z1hv
HP aC++ A.05.55 IA-64	_Z1hi	_Z1hic	_Z1hv
GCC 3.*x* and 4.*x*	_Z1hi	_Z1hic	_Z1hv
GCC 2.9x	h__Fi	h__Fic	h__Fv
HP aC++ A.03.45 PA-RISC	h__Fi	h__Fic	h__Fv
Digital Mars C++	?h@@YAXH@Z	?h@@YAXHD@Z	?h@@YAXXZ
Borland C++ v3.1	@h$qi	@h$qizc	@h$qv
OpenVMS C++ V6.5 (ARM)	H__XI	H__XIC	H__XV
OpenVMS C++V6.5 ANSI	CXX$__7H__FI0ARG51T	CXX$__7H__FIC26CDH77	CXX$__7H__FV2CB06E8
OpenVMS C++ X7.1 IA-64	CXX$_Z1HI2DSQ26A	CXX$_Z1HIC2NP3LI4	CXX$_Z1HV0BCA19V
SunPro CC	__1cBh6Fi_v_	__1cBh6Fic_v_	__1cBh6F_v_
Tru64 C++ V6.5 ARM	h__Xi	h__Xic	h__Xv
Tru64 C++ V6.5 ANSI	__7h__Fi	__7h__Fic	__7h__Fv
Watcom C++ 10.6	W?h$n(i)v	W?h$n(ia)v	W?h$n()v

Name Decoration

- This term is sometimes used as a synonym for name mangling
- Here we use it to refer to the decoration of names with various symbols depending on calling convention
- Name decoration is OS and compiler specific

Calling Conventions

- Calling conventions specify stack cleanup convention, order in which parameters are pushed, and how names are decorated
- These are from MS Visual Studio C++

Keyword	Stack cleanup	Parameter passing
__cdecl	Caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	n/a	Load parameters onto CLR expression stack in order (left to right).
__stdcall	Callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	Callee	Stored in registers, then pushed on stack
__thiscall	Callee	Pushed on stack; this pointer stored in ECX

Associated Name Decoration

- The calling convention also determines how names are decorated internally
- From MS Visual Studio C++

`int _cdecl    f (int x) { return 0; }`	_f
`int _stdcall  g (int y) { return 0; }`	_g@4
`int _fastcall h (int z) { return 0; }`	@h@4

How to Avoid Name Mangling

- In C++You can use the "extern" directive to specify the __cdecl calling convention and thereby avoid C++ name mangling

```
extern "C" int add (int *a, int b);
```

- OR

```
extern "C" {
   int add (int *a, int b);
   int sub (int *a, int b);
}
```

cdecl and Linux gcc (elf format)

- Unfortunately gcc does not decorate names with an underscore when elf (Executable and Linkable Format) object files are the target output format

A Program Skeleton (gcc)

```
; file: skel.asm
; This file is a skeleton that can be used to start asm programs.
%include "asm_io.inc"
segment .data
; initialized data is put in the data segment here
;
segment .bss
;
; uninitialized data is put in the bss segment
;
segment .text
        global  asm_main
asm_main:
        enter   0,0             ; setup routine
        pusha
; code is put here in the text segment. Do not modify the code
; before or after this comment.
;
        popa
        mov     eax, 0          ; return back to C
        leave
        ret
```

A Program Skeleton (not gcc)

```
; file: skel.asm
; This file is a skeleton that can be used to start asm programs.
%include "asm_io.inc"
segment .data
; initialized data is put in the data segment here
;
segment .bss
;
; uninitialized data is put in the bss segment
;
segment .text
        global  _asm_main
_asm_main:
        enter   0,0             ; setup routine
        pusha
; code is put here in the text segment. Do not modify the code
; before or after this comment.
;
        popa
        mov     eax, 0          ; return back to C
        leave
        ret
```

Saving Registers

- Tends to be compiler specific, but here are some general guidelines:

1. Segment registers CS, DS, ES, SS must be preserved (return from asm unmodified)
2. ebx, esi, edi and ebp must be preserved
   - epb of of course is the frame pointer
   - ebx, esi and edi are used for register variables
3. The accumulator eax is used for function results
4. Otherwise a program can modify ecx and edx

Compiling C/C++ to Assembler

- Nearly all C/C++ compilers will produce assembler listings
- This can be handy for a number of reasons:
  - segment directives
  - calling conventions
  - naming conventions
  - parameter passing conventions
  - function return values
- Compile the main module of the C++ program with -S or /FAs option.
  - Microsoft Visual C++:
    - **cl /FAs foo.c ==> foo.asm**
  - Borland C++
    - **bcc32 -S foo.c ==> foo.asm**
  - gcc (AT&T GAS assembler)
    - **gcc -S foo.c ==> foo.s**

Example: main5.c

```
#include <stdio.h>

#include "cdecl.h"

void PRE_CDECL calc_sum( int, int * ) POST_CDECL;
  /* prototype for assembly routine */

int main( void ) {
  int n, sum;

  printf("Sum integers up to: ");
  scanf("%d", &n);
  calc_sum(n, &sum);
  printf("Sum is %d\n", sum);
  return 0;
}
```

sub5.asm:1

```
%include "asm_io.inc"
; subroutine _calc_sum
; finds the sum of the integers 1 through n
; Parameters:
;   n    - what to sum up to (at [ebp + 8])
;   sump - pointer to int to store sum into (at
  [ebp+12])
; pseudo C code:
; void calc_sum( int n, int * sump ) {
;   int i, sum = 0;
;   for( i=1; i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }
segment .text
        global  _calc_sum
;
; local variable:
;   sum at [ebp-4]
```

sub5.asm:2

```
_calc_sum:
  push ebp
  mov ebp, esp
  sub esb, 4
  push ebx              ; IMPORTANT! Save for C

  mov dword [ebp-4],0   ; sum = 0
 dump_stack 1, 2, 4     ; print out stack
                        ; from ebp-8 to ebp+16
 mov ecx, 1             ; ecx is i in pseudocode
```

sub5.asm:3

```
for_loop:
  cmp ecx, [ebp+8]      ; cmp i and n
  jnle end_for          ; if not i <= n, quit
  add [ebp-4], ecx      ; sum += i
  inc ecx
  jmp short for_loop
end_for:
  mov ebx, [ebp+12]     ; ebx = sump
  mov eax, [ebp-4]      ; eax = sum
  mov [ebx], eax
  pop ebx               ; restore ebx
  mov esp, ebp
  pop ebp
  ret
```

Example: main6.c

```
#include <stdio.h>

#include "cdecl.h"

int PRE_CDECL calc_sum( int ) POST_CDECL;
 /* prototype for assembly routine */

int main( void ) {
  int n, sum;
  printf("Sum integers up to: ");
  scanf("%d", &n);
  sum = calc_sum(n);
  printf("Sum is %d\n", sum);
  return 0;
}
```

sub6.asm:1

```
segment .text
  global  _calc_sum
;
; local variable:
;   sum at [ebp-4]
_calc_sum:
  push ebp
  mov ebp, esp
  sub esb, 4

  mov dword [ebp-4],0   ; sum = 0
  mov ecx, 1            ; ecx is i in pseudocode
```

sub6.asm:2

```
for_loop:
  cmp ecx, [ebp+8]      ; cmp i and n
  jnle end_for          ; if not i <= n, quit
  add [ebp-4], ecx      ; sum += i
  inc ecx
  jmp short for_loop
end_for:
  mov eax, [ebp-4]      ; eax = sum
  mov esp, ebp
  pop ebp
  ret
```

Calling C Standard I/O Functions

• Just follow cdecl calling conventions

```
segment .data
  x dd 0
  format db "x = %d\n", 0

segment .text
  ...
  push dword [x] ; push x’s value
  push dword format ; push address of format string
  call _printf ; note underscore!
  add esp, 8 ; remove parameters from stack
```