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 be 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 actual spelling of “extern” may vary.
  - C extern
  - Pascal external
  - NASM EXTRN
  - NASM extern
- A Global (NASM) or PUBLIC (MASM) directive is used in a module whose names will be referenced by other modules.
  - 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 read 1
segment .text
global __asm_main
global __asm_main
extern get_int, print_sum
__asm_main:
    enter 0,0 ; setup routine
    pusha
```
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

```c
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("Value of foo is \%i", foo);
   dvar *= foo + 1;
   printf("Value of dvar is \%i", dvar);
   return 0;
}
```

The asm side of the coin

```asm
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

```assembly
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

```c
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

```assembly
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 returned from value-returning functions?
  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
  - a
  - b
  - x
  - Return eip
  - Caller’s ebp
- Right-to-left
  - EBP+16
  - EBP+12
  - EBP+8
  - EBP+4
  - Return eip
  - 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
  - (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
  push ebp
  mov ebp, esp
  ret 0
  ?test@@YAXXZ ENDP ; test

- PUBLIC?test@YAXX@ ; test
  push ebp
  mov ebp, esp
  ret 0
  ?test@YAXX@ ENDP ; test

- PUBLIC?test@@YAXX@ ; test
  push ebp
  mov ebp, esp
  ret 0
  ?test@@YAXX@ ENDP ; test

- PUBLIC?test@YAXX@ ; test
  push ebp
  mov ebp, esp
  ret 0
  ?test@YAXX@ ENDP ; test
Output from Borland C++

```
7live138595; }; void test() {
push ebp
mov ebp, esp
pop ebp
ret
}@test$qv endp
```

More Examples

<table>
<thead>
<tr>
<th>Compiler</th>
<th>void h(int)</th>
<th>void h(int, char)</th>
<th>void h(void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel C++ 8.0 for Linux</td>
<td>_Z1hl</td>
<td>_Z1hlc</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>HP aCC A00.55 IA-64</td>
<td>_Z1hl</td>
<td>_Z1hlc</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>GCC 3.x and 4.x</td>
<td>_Z1hl</td>
<td>_Z1hlc</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>Clang 3.5</td>
<td>h_Fi</td>
<td>h_Fic</td>
<td>h_Fv</td>
</tr>
<tr>
<td>HP aCC A03.45 PA-RISC</td>
<td>h_Fi</td>
<td>h_Fic</td>
<td>h_Fv</td>
</tr>
<tr>
<td>OpenVMS C++</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ V6.5 ANSI</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ V6.5 (ARM)</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ X7.1 IA-64</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>SunPro CC</td>
<td>__7h__Fv</td>
<td>__7h__Fic</td>
<td>__7h__Fv</td>
</tr>
<tr>
<td>SunPro CC</td>
<td>__7h__Fi</td>
<td>__7h__Fic</td>
<td>__7h__Fi</td>
</tr>
<tr>
<td>Intel C++ 8.0 for Linux</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>HP aCC A00.55 IA-64</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>GCC 3.x and 4.x</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>Clang 3.5</td>
<td>h_Fi</td>
<td>h_Fic</td>
<td>h_Fv</td>
</tr>
<tr>
<td>HP aCC A03.45 PA-RISC</td>
<td>h_Fi</td>
<td>h_Fic</td>
<td>h_Fv</td>
</tr>
<tr>
<td>OpenVMS C++</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ V6.5 ANSI</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ V6.5 (ARM)</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>OpenVMS C++ X7.1 IA-64</td>
<td>_Z1hi</td>
<td>_Z1hi</td>
<td>_Z1hv</td>
</tr>
<tr>
<td>SunPro CC</td>
<td>__7h__Fv</td>
<td>__7h__Fic</td>
<td>__7h__Fv</td>
</tr>
<tr>
<td>SunPro CC</td>
<td>__7h__Fi</td>
<td>__7h__Fic</td>
<td>__7h__Fi</td>
</tr>
</tbody>
</table>

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++

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Stack cleanup</th>
<th>Parameter passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl</td>
<td>Caller</td>
<td>Pushes parameters on the stack, in reverse order (right to left)</td>
</tr>
<tr>
<td>clical</td>
<td>n/a</td>
<td>Load parameters onto CLR expression stack in order (left to right)</td>
</tr>
<tr>
<td>stdcall</td>
<td>Caller</td>
<td>Pushes parameters on the stack, in reverse order (right to left)</td>
</tr>
<tr>
<td>fastcall</td>
<td>Caller</td>
<td>Stored in registers, then pushed on stack</td>
</tr>
<tr>
<td>thiscall</td>
<td>Caller</td>
<td>Pushes an stack, this pointer stored in EAX</td>
</tr>
</tbody>
</table>

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; }
int _stdcall g (int y) { return 0; }
int _fastcall h (int z) { return 0; }
```

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)

```asm
#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)

```asm
#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
     ebp of course is the frame pointer
  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
parameter passing conventions
function return values
• Compile the main module of the C++ program with -S or /FS options.
  Microsoft Visual C++:
  cl /FS 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

```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", sum);
    return 0;
}
```
sub5.asm:1

; 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

; 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   ; restore ebx
  mov esp, ebp     ; pop ebx
  pop esp
  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
  mov [ebp-4], 0 ; sum = 0
  mov ecx, 1    ; ecx is i in pseudocode
  ret

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 ebx
  pop esp
  ret
Calling C Standard I/O Functions

• Just follow cdecl calling conventions

```assembly
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
```