We have already seen how the `call` instruction is used to execute a subprogram.

`call` pushes the address of the instruction after the `call` onto the stack (the return address) and loads the instruction pointer (EIP) register with the operand (an address):

```
call foobar  # Jump to foobar
```

In the subprogram, a `ret` loads the EIP with the return address from the stack.

For subprograms with no arguments and no return value, `call` and `ret` are sufficient.

With arguments, the calling program and subprogram must agree on how the arguments are passed. This is known as a calling convention. The calling convention may differ from compiler to compiler.

To interface assembly with C/C++ programs, we must know the calling convention.

The CS220 library routines use a register based calling convention. All arguments and return values are passed in registers. This method is very fast and easy, but only permits a limited number of arguments. The arguments are also limited in size. (They have to fit into a register.)

The GNU C and C++ compilers (as well as MS VS C/C++) use the C calling (`cdecl`) convention for free (non-member) functions.

In the `cdecl` calling convention arguments are passed on the stack. They are pushed (before the CALL) in right to left order.

All integers (char, short, int) are pushed onto the stack as 32 bit values, as are pointers and floats. A long long int is pushed as two 32-bit values (least significant word pushed last!). A double occupies 8 bytes. A long double is pushed as extended precision (10 bytes) with two bytes of padding (12 bytes).

The arguments are popped off the stack by the calling program after the return.

After a call to a subprogram that takes a single argument the stack would appear as:

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>4(%esp)</td>
<td>(%esp)</td>
</tr>
</tbody>
</table>
```
### The cdecl Calling Convention

The argument can be accessed in the subprogram using the address 4(%esp). If the stack is used for local variable storage in the subprogram the value of the offset from the ESP to the arguments changes:

<table>
<thead>
<tr>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Return Address</th>
<th>Subprogram Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>8(%esp)</td>
<td>12(%esp)</td>
<td>(%esp)</td>
<td>(esp)</td>
</tr>
<tr>
<td>4(%esp)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Accessing Parameters

To solve this problem, the subprogram saves the value of EBP on the stack and then sets EBP equal to ESP. The EBP register is used to refer to data on the stack:

```
pushl %ebp      # save original EBP
movl %esp,%ebp  # new EBP = ESP
# SUBPROGRAM CODE HERE
movl %ebp,%esp  # restore SP
popl %ebp       # restore EBP
ret
```

### Removing Parameters

After the subprogram completes, the arguments that were pushed on the stack must be removed. The cdecl calling convention requires that the calling program do this:

```
pushl %eax      # push parameter on stack
call foobar     # call subprogram
addl $4,%esp    # adjust ESP
The subprogram is permitted to change the value on the stack!
```

### Local Variables

A reentrant subprogram can be called at any place, including from the subprogram itself. (Reentrant subprograms can be called recursively.) Subprograms that use only automatic variables are always reentrant.
AS08-C++ and Assembly

Local Variables

- Local variables are stored on the stack right after the saved EBP value. Local variables are accessed using negative offsets to EBP.
- IMPORTANT: (%ebp) is the old EBP value, 4(%ebp) is the return address, 8(%ebp), 12(%ebp), 16(%ebp), ... are subprogram parameters and -4(%ebp), -8(%ebp), -12(%ebp), ... are local variables.

The following C++ subroutine:

```cpp
void calcsum(int n, int *sump)
{
    int i, sum = 0;
    for (i = 1; i <= n; i++)
        sum += i;
    *sump = sum;
}
```

can be implemented in assembly as (Note: This subprogram requires that the calling program push *sump on the stack before n):

```assembly
calcsum:  push %ebp
            movl %esp,%ebp
            subl $4,%esp  # room for sum
            movl $0,-4(%ebp)  # sum = 0
            movl $1,%ebx  # ebx (i) = 1
            for_loop:
                cmpl 8(%ebp),%ebx    # is i <= n?
                jnle end_for
                addl %ebx,-4(%ebp)   # sum += i
                incl %ebx
                jmp  for_loop
            end_for:
                movl 12(%ebp),%ebx   # ebx = *sump
                movl -4(%ebp),%eax   # eax = sum
                movl %ebx,%ebx
                movl %ebx,%esp
                popl %ebp
            ret
```

- The enter instruction is equivalent to the three instructions at the beginning of the subprogram. enter takes two operands. The second operand is 0 in the cdecl calling convention. The first operand is the number of bytes needed by the local variables.

- The leave instruction is equivalent to the two instructions just before the ret. The leave instruction has no operands.

```assembly
leave          # ESP = EBP, pop EBP
ret
```

- The cdecl calling convention permits the EAX, ECX, and EDX registers to be used freely by the subprogram. If they are being used by the calling program, they should be pushed onto the stack (or otherwise saved) before calling the subprogram and then restored after the return.

- All other registers (EBX, EBP, EDI, ESI) must be preserved (typically by pushing onto the stack) if used by the subprogram.

- Using enter and leave the general form of a subprogram with local variables is:

  ```assembly
  foobar:
  enter $BYTES,0  # push EBP, EBP = ESP
  # make room for locals
  # SUBPROGRAM CODE HERE
  leave  # ESP = EBP, pop EBP
  ret
  ```

- The skeleton.s code uses the enter and leave instructions. (It does not reserve any room for local variables however.)
**AS08-C++ and Assembly**

**Using Registers**

- So a subprogram should have the form:

  ```assembly
  foobar:
  enter $BYTES,0 # Saves EBP for us
  pushl %ebx     # If used below
  pushl %edi     # If used below
  pushl %esi     # If used below
  # SUBPROGRAM CODE HERE
  popl  %esi     # Only if pushed above
  popl  %edi     # Only if pushed above
  popl  %ebx     # Only if pushed above
  leave          # Restores EBP
  ret
  ```

- The calling program should have the form:

  ```assembly
  pushl %eax     # Only if being used
  pushl %ecx     # Only if being used
  pushl %edx     # Only if being used
  # PUSH SUBPROGRAM ARGUMENTS HERE
  call foobar
  # ADJUST ESP FOR ARGUMENTS HERE
  popl %edx      # Only if pushed above
  popl %ecx      # Only if pushed above
  popl %eax      # Only if pushed above
  ```

**Return Values**

- According to the cdecl calling convention, the char, short, int and pointer types are returned in EAX. The long long type is returned in EDX:EAX. The float, double and long double types are returned in %st(0). All other FPU registers should be empty (free).

- The calling convention also dictates how structures and classes are passed and returned. (The details are left for research by the interested student.)

**An Example Subprogram**

- Consider a function that returns the sum of its two int arguments. The return is an int:

  ```assembly
  .globl _sumtwo
  .section .text
  _sumtwo:
  enter   $0, $0
  movl    8(%ebp),%eax
  addl    12(%ebp),%eax
  leave
  ret
  ```

- Here is a C++ driver:

  ```cpp
  #include <iostream>
  using namespace std;
  extern "C"
  int sumtwo(int a, int b);
  int main()
  {
    cout << sumtwo(2, 4) << "\n";
  }
  ```

- The program would be built using:

  ```sh
  g++ -o sumtwo driver.cpp sumtwo.s
  ```

- Note: The MinGW C/C++ compiler prepends an underscore to all function names and global variables (for compatibility with MS Visual Studio). We must manually add this to the name in assembly. (The Linux C/C++ compiler does not do this and the underscore should be omitted in assembly.)
The `extern "C"` qualifier tells the C++ compiler that we are calling a C type function. C++ allows function overloading (C does not) and achieves this through name mangling. (The function names at the assembly level are different than those used at the C++ level.) The `extern "C"` qualifier tells the C++ compiler that the routine name is not name mangled. It is not necessary to use the `extern "C"` qualifier in C.

Here is the corresponding assembly language driver:

```
pushl   $4         # Push args
pushl   $2
call    _sumtwo
addl    $8, %esp   # Pop args
```

```
call    print_int
```

```
call    print_nl
```

To compile (assemble):

```
g++ -o sumtwo driver.s sumtwo.s \
    -L cs220lib -lcs220
```

Here is a similar function but now the first argument is a double and the second is an int. The return is a double:

```
.globl _sumtwo
.section .text
_sumtwo:
    enter  $0, $0  # setup stack
    fldl  8(%ebp) # FPU: a
    fiaddl 16(%ebp)# FPU: a+b
    leave          # restore stack
    ret
```

Here is the new C++ driver:

```
#include <iostream>
using namespace std;
extern "C"
    double sumtwo(double a, int b);
int main()
{
    cout << sumtwo(2.3, 4) << "\n";
}
```

And the new corresponding assembly driver:

```
# Push 4 on stack
pushl   $4
# Move 2.3 onto stack
fldl    CON1
subl    $8, %esp
fstpl   (%esp)
call    _sumtwo
addl    $12, %esp # Pop args
call    print_double
ffree   %st(0)
```

A multi-module program consists of more than one object file. All of the programs that we have written are multi-module programs that consist of our own module and modules from the C library.

The linker combines the modules into an executable. It matches up references in one module to definitions in a different module.
AS08-C++ and Assembly

Multi-Module Programs

- The `global` (or `globl`) directive makes a label in the current module visible to other modules:
  .globl _asm_main
  .globl _sumtwo, _sumthree
- Global variables defined in the data or bss sections that are to be shared with other modules must also be declared global.

Other Calling Conventions

- The standard calling convention (`stdcall`) is similar to `cdecl`, but requires the subprogram to pop the arguments off of the stack. This is done by using an operand with `ret`. The ESP is incremented by the operand value after popping the return address:
  `ret $40`
- `stdcall` does not allow variable length argument lists. It is used by functions in the MS Windows API.

- The `thiscall` convention is used when calling C++ member functions. The object address is pushed on the stack after all of the arguments (in the first argument position). In MS C++ the address is passed in ECX.
- In the `fastcall` convention the first and second arguments are passed in the ECX and EDX registers respectively. All other arguments are pushed on the stack.

Name Mangling

- C++ uses name mangling to allow overloaded functions and methods. For example `foo::bar(int, long) const` is mangled as "bar__C3fooil" while `foo::bar(int, double)` is mangled as "bar__3foooid". Since our primary goal is understanding how function calls work, the details of name mangling will not be covered.

Other Calling Conventions

- GCC/G++ allows you to specify a calling convention other than the default with the `__attribute__` keyword:
  `int foo(int a) __attribute__((stdcall));`
- Calling convention details differ from compiler to compiler. This makes it difficult to interface code compiled with different compilers (GNU C++ and VS C++).