Chapter 1

x86 Assembly, 64 bit

. . .

This chapter was derived from a document written by Adam Ferrari and later updated by Alan Batson, Mike Lack, Anita Jones, and Aaron Bloomfield

1.1 Introduction

This small guide, in combination with the material covered in the class lectures on assembly language programming, should provide enough information to do the assembly language labs for this class. In this guide, we describe the basics of 64-bit x86 assembly language programming, covering a small but useful subset of the available instructions and assembler directives. However, real x86 programming is a large and extremely complex universe, much of which is beyond the useful scope of this class. For example, there exists real (albeit older) x86 code running in the world was written using the 16-bit subset of the x86 instruction set. Using the 16-bit programming model can be quite complex – it has a segmented memory model, more restrictions on register usage, and so on. This was expanded into a 32-bit programming model in 1985, but that had the limit of only 4 Gb of memory. In this guide we'll restrict our attention to the more modern aspects of 64-bit x86 programming, and delve into the instruction set only in enough detail to get a basic feel for programming x86 compatible chips at the hardware level.

1.2 Registers

Modern 64-bit x86 processors have sixteen 64-bit general purpose registers, as depicted in Figure 1.1. The register names for the first eight registers are mostly historical in nature; the last eight registers were given sequential numbers. For example, RAX used to be EAX (in the 32-bit machine), which used to be called the "accumulator" since it was used by a number of arithmetic operations, and RCX (32-bit version: ECX) was known as the "counter" since it was used to hold a loop index. Whereas most of the registers have lost their special purposes in the modern instruction set, by convention, two are sometimes reserved for special purposes – the stack pointer (RSP) and the base pointer (RBP).

In all cases, subsections of the registers may be used. For example, the least significant 2 bytes of RAX can be treated as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit

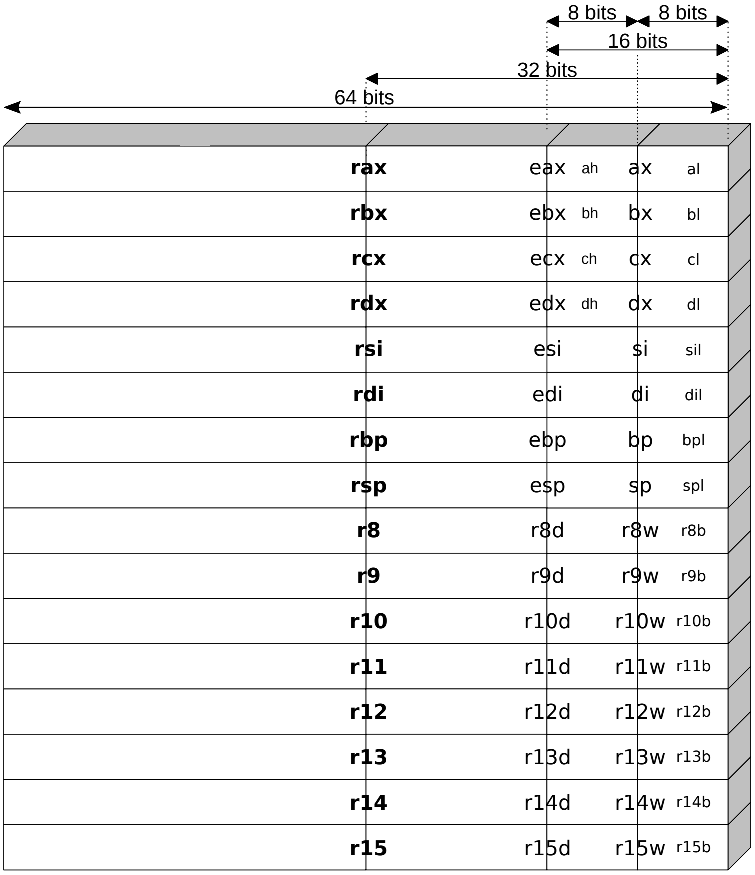

Figure 1.1: The x86 register set

register called AL, while the most significant byte of AX can be used as a single 8-bit register called AH. It is important to realize that these names refer to the same physical register. When a two-byte quantity is placed into DX, the update affects the value of RDX (in particular, the least significant 16 bits of RDX). These "sub-registers" are mainly hold-overs from older, 16-bit versions of the instruction set. However, they are sometimes convenient when dealing with data that are smaller than 64-bits (e.g., 1-byte ASCII characters). Note that four of the registers (EAX, EBX, ECX, and EDX) have an addition "sub-register" spot: the second to last byte of the register.

When referring to registers in assembly language, the names are not case-sensitive. For example, the names RAX and rax refer to the same register.

1.3 Memory and Addressing Modes

1.3.1 Declaring Static Data Regions

You can declare static data regions (analogous to global variables) in x86 assembly using special assembler directives for this purpose. Data declarations should be preceded by the .DATA directive. Following this directive, the directives DB, DW, and DD can be used to declare one, two, and four byte data locations, respectively. Declared locations can be labeled with names for later reference - this is similar to declaring variables by name, but abides by some lower level rules. For example, locations declared in sequence will be located in memory next to one another. Some example declarations are depicted in Listing 1.1.

Listing 1.1: Declaring x86 memory regions

```
section .data
var     DB      64      ; Declare a byte containing the value 64. Label the
                        ; Memory location "var".
var2    DB      ?       ; Declare an uninitialized byte labeled "var2".
        DB      10      ; Declare an unlabeled byte initialized to 10. This
                        ; byte will reside at the memory address var2+1.
var3    DQ      ?       ; Declare an uninitialized 64-bit quantity ('Q' is for
                        ; quad-word)
X       DW      ?       ; Declare an uninitialized two-byte word labeled "X".
Y       DD      3000    ; Declare 32 bits of memory starting at address "Y"
                        ; initialized to contain 3000.
Z       DD      1,2,3   ; Declare three 4-byte words of memory starting at
                        ; address "Z", and initialized to 1, 2, and 3,
                        ; respectively. E.g. 3 will be stored at address Z+8.
```

The last example in Listing 1.1 illustrates the declaration of an array. Unlike in high level languages where arrays can have many dimensions and are accessed by indices, arrays in assembly language are simply a number of cells located contiguously in memory. Two other common methods used for declaring arrays of data are the TIMES directive and the use of string literals. The TIMES directive tells the assembler to duplicate an expression a given number of times. For example, the statement "TIMES 4 DB 2" is equivalent to "2, 2, 2, 2". Some examples of declaring arrays are depicted in Listing 1.2.

Listing 1.2: Declaring x86 arrays in memory

```
section .data
bytes   TIMES 10 DB     ?           ; Declare 10 uninitialized bytes starting at
                                    ; the address "bytes".
arr     TIMES 100 DD    0           ; Declare 100 4 bytes words, all initialized
                                    ; to 0, starting at memory location "arr".
str     DB        'hello', 0        ; Declare 5 bytes starting at the address
                                    ; "str" initialized to the ASCII character
                                    ; values for the characters 'h', 'e', 'l',
                                    ; 'l', 'o', and '\0' (NULL), respectively.
```

1.3.2 Addressing Memory

Modern x86-compatible processors are capable of addressing up to 2^{64} bytes of memory; that is, memory addresses are 64-bits wide. For example, in Listings 1.1 and 1.2, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 64-bit quantities that specify addresses in memory. In addition to supporting referring to memory regions by labels (i.e. constant values), the x86 provides a flexible scheme for computing and referring to memory addresses:

x86 Addressing Mode Rule – Up to two of the 64-bit registers and a 64-bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre-multiplied by 2, 4, or 8.

To see this memory addressing rule in action, we'll look at some example mov instructions. As we'll see later in Section 1.4.1, the `mov` instruction moves data between registers and memory. This instruction has two operands – the first is the destination (where we're moving data *to*) and the second specifies the source (where we're getting the data *from*). Some examples of mov instructions using address computations that obey the above rule are shown in Listing 1.3.

Listing 1.3: Valid x86 addressing modes

```
mov rax, [rbx]            ; Move the 8 bytes in memory at the address contained
                          ; in EBX into EAX
mov [var], rbx            ; Move the contents of EBX into the 8 bytes at memory
                          ; address "var" (Note, "var" is a 32-bit constant).
mov rax, [rsi -8]         ; Move 8 bytes at memory address ESI+(-8) into EAX
mov [rsi+rax], cl         ; Move the contents of CL into the byte at address
                          ; ESI+EAX
mov edx, [esi+4*ebx]      ; Move the 4 bytes of data at address ESI+4*EBX into
                          ; EDX. This is only 4 bytes because we use the
                          ; 4-byte "sub-registers" of edx, esi, and ebx instead
                          ; of rdx, rsi, and rbx
```

Some examples of incorrect address calculations are shown in Listing 1.4.

Listing 1.4: Invalid x86 addressing modes

```
mov rax, [rbx-rcx]        ; Can only add register values
mov [rax+rsi+rdi], rbx    ; At most 2 registers in address computation
```

1.3.3 Size Directives

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, in all of the above instructions, the size of the memory regions could be inferred from the size of the register operand – when we were loading a 64-bit register, the assembler could infer that the region of memory we were referring to was 8 bytes wide. When we were storing the value of a one byte register to memory, the assembler could infer that we wanted the address to refer to a single byte in memory. However, in some cases the size of a referred-to memory region is ambiguous. Consider the instruction `mov [ebx], 2`.

Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 64-bit integer representation of 2 into the 4-bytes starting at address EBX. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives `BYTE PTR`, `WORD PTR`, `DWORD PTR`, and `QWORD PTR` serve this purpose. For examples, see Listing 1.5.

Listing 1.5: x86 size directive usage

```
mov BYTE PTR [ebx], 2        ; Move 2 into the single byte at memory
                             ; location EBX
mov WORD PTR [ebx], 2        ; Move the 16-bit integer representation of 2
                             ; into the 2 bytes starting at address EBX
mov DWORD PTR [ebx], 2       ; Move the 32-bit integer representation of 2
                             ; into the 4 bytes starting at address EBX
mov QWORD PTR [ebx], 2       ; Move the 64-bit integer representation of 2
                             ; into the 8 bytes starting at address EBX
```

1.4 Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow. In this section, we will look at important examples of x86 instructions from each category. This section should not be considered an exhaustive list of x86 instructions, but rather a useful subset.

In this section, we will use the following notation:

- <reg64> - means any 64-bit register described in Section 2, for example, ESI.
- <reg16> - means any 16-bit register described in Section 2, for example, BX.
- <reg32> - means any 32-bit register described in Section 2, for example, BX.
- <reg8> - means any 8-bit register described in Section 2, for example AL.
- <reg> - means any of the above.
- <mem> - will refer to a memory address, as described in Section 1.3.2, for example `[EAX]`, or `[var+4]`, or `DWORD PTR [RAX+RBX]`.
- <con64> - means any 64-bit constant.
- <con32> - means any 32-bit constant.
- <con16> - means any 16-bit constant.
- <con8> - means any 8-bit constant.
- <con> - means any of the above sized constants.

1.4.1 Data Movement Instructions

Instruction: mov

Syntax:

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Semantics: The mov instruction moves the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Examples:

```
mov rax, rbx              ; transfer ebx to eax
mov BYTE PTR [var], 5     ; store the value 5 into the byte at
                          ; memory location ``var''
```

Instruction: push

Syntax:

```
push <reg>
push <mem>
push <con64>
```

Semantics: The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements RSP by 8, then places its operand into the contents of the 64-bit location at address [RSP]. RSP (the stack pointer) is decremented by push since the x86 stack grows down – i.e. the stack grows from high addresses to lower addresses.

Examples:

```
push rax     ; push the contents of rax onto the stack
push [var]   ; push the 8 bytes at address ``var'' onto stack
```

Instruction: pop

Syntax:

```
pop <reg>
pop <mem>
```

Semantics: The pop instruction removes the 8-byte data element from the top of the hardware supported stack into the specified operand (i.e. register or memory location). Specifically, pop first moves the 8 bytes located at memory location [RSP] into the specified register or memory location, and then increments SP by 8.

Examples:

```
pop rdi      ; pop the top element of the stack into RDI.
pop [rbx]    ; pop the top element of the stack into memory at
             ; the four bytes starting at location RBX.
```

Instruction: lea

Syntax:

```
lea <reg64>,<mem>
```

Semantics: The lea instruction places the address specified by its second operand into the register specified by its first operand. Note, the contents of the memory location are not loaded – only the effective address is computed and placed into the register. This is useful for obtaining a “pointer” into a memory region.

Examples:

```
lea rax, [var]        ; the address of ``var'' is placed in RAX
lea rdi, [rbx+4*rsi]  ; the value RBX+4*RSI is placed in RDI
```

1.4.2 Arithmetic and Logic Instructions

Instruction: add, sub

Syntax:

```
add <reg>,<reg>       sub <reg>,<reg>
add <reg>,<mem>       sub <reg>,<mem>
add <mem>,<reg>       sub <mem>,<reg>
add <reg>,<con>       sub <reg>,<con>
add <mem>,<con>       sub <mem>,<con>
```

Semantics: The add instruction adds together its two operands, storing the result in its first operand. Similarly, the sub instruction subtracts its second operand from its first. Note, whereas both operands may be registers, at most one operand may be a memory location.

Examples:

```
add rax, 10               ; add 10 to the contents of RAX.
sub [var], rsi            ; subtract the contents of RSI from
                          ; the 64-bit integer stored at
                          ; memory location ``var''.
add BYTE PTR [var], 10    ; add 10 to the single byte stored
                          ; at memory address ``var''.
```

Instruction: inc, dec

Syntax:

```
inc <reg>        dec <reg>
inc <mem>        dec <mem>
```

Semantics: The inc instruction increments the contents of its operand by one, and similarly dec decrements the contents of its operand by one.

Examples:

```
dec rax               ; subtract one from the contents of RAX.
inc DWORD PTR [var]   ; add one to the 64-bit integer stored at
                      ; memory location ``var''.
```

Instruction: imul

Syntax:

```
imul <reg64>,<reg64>
imul <reg64>,<mem>
imul <reg64>,<reg64>,<con>
imul <reg64>,<mem>,<con>
```

Semantics: The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above). The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e., first) operand must be a register. The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Examples:

```
imul rax, [var]       ; multiply the contents of RAX by the
                      ; 64-bit contents of the memory location
                      ; ``var''.  Store the result in RAX.
imul rsi, rdi, 25     ; multiply the contents of RDI by 25.
                      ; Store the result in RSI.
```

Instruction: idiv

Syntax:

```
idiv <reg64>
idiv <mem>
```

Semantics: The idiv instruction is used to divide the contents of the 64 bit integer RDX:RAX (constructed by viewing RDX as the most significant eight bytes and RAX as the least significant eight bytes) by the specified operand value. The quotient result of the division is stored into RAX, while the remainder is placed in RDX. This instruction must be used with care. Before executing the instruction, the appropriate value to be divided must be placed into RDX and RAX. Clearly, this value is overwritten when the idiv instruction is executed.

Examples:

```
idiv rbx                ; divide the contents of RDX:RAX by the
                        ; contents of RBX. Place the quotient
                        ; in RAX and the remainder in RDX.
idiv DWORD PTR [var]    ; same as above, but divide by the
                        ; 64-bit value stored at memory
                        ; location ``var''.
```

Instruction: and, or, xor

Syntax:

```
and <reg>,<reg>     or <reg>,<reg>     xor <reg>,<reg>
and <reg>,<mem>     or <reg>,<mem>     xor <reg>,<mem>
and <mem>,<reg>     or <mem>,<reg>     xor <mem>,<reg>
and <reg>,<con>     or <reg>,<con>     xor <reg>,<con>
and <mem>,<con>     or <mem>,<con>     xor <mem>,<con>
```

Semantics: These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Examples:

```
and rax, 0fH          ; clear all but the last 4 bits of RAX.
xor rdx, rdx          ; set the contents of RDX to zero.
```

Instruction: not

Syntax:

```
not <reg>
not <mem>
```

Semantics: Performs the logical negation of the operand contents (i.e., flips all bit values).

Examples:

```
not BYTE PTR [var]    ; negate all bits in the byte at the
                      ; memory location ``var''.
```

Instruction: neg

Syntax:

```
neg <reg>
neg <mem>
```

Semantics: Performs the arithmetic (i.e., two's complement) negation of the operand contents.

Examples:

```
neg rax            ; negate the contents of RAX.
neg [var]          ; negate the contests of ``var''
```

Instruction: shl, shr

Syntax:

```
shl <reg>,<con8>        shr <reg>,<con8>
shl <mem>,<con8>        shr <mem>,<con8>
shl <reg>,cl            shr <reg>,cl
shl <mem>,cl            shr <mem>,cl
```

Semantics: These instructions shift the bits in their first operand's contents left and right (`shl` and `shr`, respectively), padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 64.

Examples:

```
shl rax 5        ; shift the contents of rax left by 5 bit
                 ; positions
shr [var] 3      ; shift the contents of ``var'' right by 3
                 ; bit positions
```

1.4.3 Control Flow Instructions

In this section, we will refer to labeled locations in the program text as <label>. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example, consider the code fragment in Listing 1.6. The second instruction in this code fragment is labeled "begin". Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name "begin" instead of having to refer to the memory address as an integer.

Listing 1.6: x86 labeled code location

```
        mov rsi, [rbp+8]
begin:  xor rcx, rcx
        mov rax, [rsi]
```

Instruction: jmp

Syntax: `jmp <label>`

Semantics: Transfers program control flow to the instruction at the memory location indicated by the operand.

Examples:

```
jmp begin         ; jumps to the ``begin'' label
```

Instruction: jCC

Syntax:

```
je <label> ; Jump when equal
jne <label> ; Jump when not equal
jz <label> ; Jump when last result was zero
jg <label> ; Jump when greater than
jge <label> ; Jump when greater than or equal to
jl <label> ; Jump when less than
jle <label> ; Jump when less than or equal to
```

Semantics: These instructions are conditional jumps that are based on the status of a set of **condition codes** that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation (e.g. add, sub, etc.) was zero. Otherwise, control proceeds to the next instruction in sequence after the jz. These conditional jumps are the underlying support needed to implement high-level language features such as “if” statements and loops (e.g. “while” and “for”).

A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

Examples:

```
cmp rax, rbx     ; if the contents of rax are less than or
jle done         ; equal to the contents of ebx, jump to the
                 ; code location labeled ``done''.
```

Instruction: cmp

Syntax:

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
cmp <mem>,<con>
```

Semantics: Compares the two specified operands, setting the condition codes in the machine status word appropriately. In fact, this instruction is equivalent to the sub instruction, except the result of the subtraction is discarded.

Examples:

```
cmp DWORD PTR [var], 10   ; if the 4 bytes stored at memory
jeq loop                  ; location ``var'' equal the 4-byte
                          ; integer value 10, then jump to the
                          ; code location labeled loop
```

Instruction: call

Syntax: `call <label>`

Semantics: This instruction implements a subroutine call that operates in cooperation with the subroutine return instruction, ret, described below. This instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. The added value of this instruction (as compared to the simple jmp instruction) is that it saves the location to return to when the subroutine completes.

Examples:

```
call my_subroutine    ; jumps to the ``my_subroutine'' label,
                      ; pushing the return address onto the
                      ; stack
```

Instruction: ret

Syntax: `ret`

Semantics: In cooperation with the call instruction, the ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

Examples:

```
ret          ; returns to the address on the top of the stack
```

1.5 Basic Program Structure

Given the above repertoire of instructions, you are in a position to examine the basic skeletal structure of an assembly language subroutine suitable for linking into C++ code. Unlike C++, which is often used for the development of complete software systems, assembly language is most often used in cooperation with other languages such as Fortran, C, and C++. Commonly, most of a project is implemented in the more convenient high-level language, and assembly language is used sparingly to implement extremely low-level hardware interfaces or performance-critical "inner loops." Thus, in addition to understanding how to program in assembly language, it is equally important to understand how to link assembly language code into high-level language programs.

Listing 1.7: x86 code to return 2

```
global returnTwo

section .data
        var DD 2

section .text

returnTwo:
        mov eax, [var]
        ret
```

Before examining the linkage conventions, we must first examine the basic structure of an assembly language file. To do this, we can compare a very simple assembly language file to an equivalent C++ file. In Listings 1.7 and 1.8 we see two files, one in C++, the other in x86 assembly. Each file includes a function (albeit an ugly one) to return the integer value 2.

Note that the fucntion shown in assembly in Listing 1.7 returns a 32-bit value, since it is put into register eax. This corresponds to retnring an `int` in C or C++. If we wanted to return a 64-bit value, which corresponds to returning a `long`, then we would put the return value into rax.

The top of the assembly file contains two directives that indicate the instruction set and memory model we will use for all work in this class (note, there are other possibilities – one might use only the older 80286 instruction set for wider compatibility, for example).

Next, where in the C++ file we find the declaration of the global variable "var", in the assembly file we find the use of the .DATA and DD directives (described in Section 3.1) to reserve and initialize a 4-byte (i.e., integer-sized) memory region labeled "var".

Next in each file, we find the declaration of the function named returnTwo. In the C++ file we have declared the function to be extern "C". This declaration indicates that the C++ compiler should use C naming conventions when labeling the function returnTwo in the resulting object file that it produces. In fact, this naming convention means that the function returnTwo should map to the label _returnTwo in the object code. In the assembly code, we have labeled the beginning of the subroutine _returnTwo using the PROC directive, and have declared the label _returnTwo to be public. Again, the result of these actions will be that the subroutine will map to the symbol _returnTwo in the object code that the assembler generates.

Listing 1.8: C++ code to return 2

```
int var = 2;

extern "C" returnTwo();

int returnTwo() {
  return var;
}
```

The function bodies are straight-forward. As we will see in more detail in Section 6, return values for functions are placed into EAX by convention, hence the instruction to move the contents of "var" into EAX in the assembly code.

Given these equivalent function definitions, use of either version of the function is the same. A sample call to the function returnTwo is depicted in Listing 1.9. This C++ code could be linked to either definition of the function and would produce the same results (note, we could not link to both definitions, or the linker would produce a "multiply defined symbol" error. The mechanics of program linking will be discussed in an associated document that relates to the specific programming environment that you will use to assemble and run programs.

Listing 1.9: Calling returnTwo() from C++

```
#include <iostream>
using namespace std;

extern "C" int returnTwo();

int main() {
  cout << "calling␣returnTwo()␣returned:␣"
       << returnTwo() << endl;
  return 0;
}
```