Lecture 01: C and x86 (64-bit) Programming Refresher

August 26, 2024

Overview

C Programming

- Primitive Data Types
- Derived Data Types
- Functions
- Control Structures
- Preprocessor

x86_64 Programming

- Registers and Flags
- Instruction Syntax
- Data and Arithmetic
- Control and Subroutines

C Programming Language

Hello, World!

```
#include <stdio.h>

int main(int argc, char** argv) {
  printf("Hello, World!\n");
  return 0;
}
```

- The first line includes the standard input/output library.
- The next line declares the `main()` function, which is the entry point.
- The call to `printf()` prints the given text to “standard out.”
- The `return` line causes `main()` to terminate, returning 0.
- Because it was the entry, this becomes the program’s “exit code.”

C Programming Language

Primitive Types: Integer

Types

`char` "character", typically 8 bits or 1 byte

`int` "integer", typically 32 or 64 bits

Sign Modifiers

`signed` default, 2's complement signed

`unsigned` all values positive or 0

Size Modifiers

`short` typically 16 bits

`long` typically 32 or 64 bits

`long long` typically 64 or 128 bits

C Programming Language

Primitive Types: Floating Point

`float` “single-precision floating point” (IEEE754), 32 bits

`double` “double-precision floating point” (IEEE754), 64 bits

- Some implementations allow `short float` (16-bit) and `long double` (128-bit) variants.

`void` nothing

- Used as the return type for subroutines (no value returned)

`enum` integer with named values

- `enum DemoEnum { ZERO, ONE, TEN=10 };`
- `enum DemoEnum demo_enum = ONE;`

Boolean virtual type

- any non-zero evaluates as “true”
- “true” is cast to the integer value 1

C Programming Language

Derived Types: Pointer

Pointers are declared using the `*` modifier. While semantically, it affects the type; syntactically, it binds the variable.

Declare `int *a, b, c;` // only `a` is a pointer

Declare `void *p;` // typeless pointer, address

Assign `p = &b;`

Assign `a = (int*) p;`

Dereference `c = *a;`

Dereference `*a = c;`

Arrays are declared using the `[n]` modifier.

Declare `int arr[10];`

Declare `int param[];` // parameter or initialized

Index `c = arr[5];`

Index `arr[23] = c;` // no bounds checking

Index `arr[-3] = c;` // really

Structures (like records, or method-less classes) are declared using the `struct` keyword.

Declare `struct DemoStruct { int a; float b; } demo_struct;`

Declare `struct DemoStruct demo_struct2;`

Assign `demo_struct2 = demo_struct;`

Access `demo_struct.a = 6;`

Access `float f = demo_struct.b;`

Unions are like structures, but members share the same memory location – they are aliased. They are declared using the `union` keyword.

Declare `union DemoUnion { int c; float d; } demo_union;`

Declare `union DemoUnion demo_union2;`

Assign `demo_union2 = demo_union;`

Access `demo_union.c = 6;`

Access `float f = demo_union.d;`

Programmer can define new type names using the `typedef` keyword.

Define `typedef struct _DemoType { int a; float b; } DemoType, *DemoTypeP;`

Declare `DemoType demo;`

Declare `DemoTypeP p, q;` // Both `p` and `q` are pointers

Access `p.a = 6;`

Dereference `(*q).a = 6;`

Dereference `q->a = 6;` // convenient syntax

- On the left side of =, an array is an array. You can index it, but not assign it.
- On the right side of =, an array behaves like a pointer to its first element.
- On either side, a pointer is a pointer.
- On either side, a pointer can be indexed like an array.
- You can obtain the address of (a pointer to) any variable or function using the & symbol.
- Pointer arithmetic is dangerous – behaves more like indexing.

C Programming Language

Strings

Though a derived type, strings are first-class citizens in C.

Declare `char* str;` // pointer to a character array

Literals `str = "Here's a string.";` // null-terminated constant

- String literals are all stored in a (usually read-only) constant pool

Index `char c = str[4];` // copies the ASCII value of the apostrophe into `c`

Functions are declared using `(...)`

```
int f(int x, int y) {
  return x + 2*y;
}

void println(char* line) { // "subroutine"
  printf("%s\n", line);
}
```

Like arrays, functions can behave as pointers, and pointers on the right can behave as functions.

```
int f(int x, int y) {
  return x + 2*y;
}

typedef int (*Func3D)(int,int);

Func3D g = f; // f behaves like a pointer

int main() {
  int z = g(3, 4); // g behaves like a function
}
```

C Programming Language

Control: `if` block

Conditional execution: `if (`*cond*`)` *stmt*

```
if (a == 1) {
    printf("One\n");
} else if (a == 2) {
    printf("Two\n");
} else {
    printf("Neither\n");
}
```

C Programming Language

Control: Ternary operator `?:`

Similar to `if`: Choice of two values

cond `?` *expT* `:` *expF*

```
printf("%s\n", a == b ? "Equal" : "Unequal");
```

C Programming Language

Control: `switch` block

Choice of block by expression and labels: `switch (`*exp*`) {` *labeled statements* `}`

```
switch (a) {
    case 1:
        printf("One\n");
        break;
    case 2:
        printf("Two\n");
        break;
    default:
        printf("Neither\n");
}
```

Note you usually need `break` to prevent multiple cases from executing.

C Programming Language

Control: `while` loop

Repetition while a condition is true: `while (`*cond*`)` *stmt*

```
while (a > 0) {
    a >>= 1;
}
```

Can use `break` and `continue` to go to the end or beginning of the loop, respectively.

C Programming Language
Control: `for` loop

A structured variant of `while`: `for (`*init*`;` *cond*`;` *step*`)` *stmt*

```
for (int i = 0; i < 10; i++) {
    printf("%d\n");
}
```

C Programming Language

Control: `do`...`while` loop

A post-test variant of `while`: `do` *stmt* `while (`*cond*`);`

```
do {
    a = (a + 1) % 10;
} while (a != 0);
```

C Programming Language

Control: `goto`

The gateway to spaghetti code. There are more structured things you can probably use.

Consider `switch`, which is basically a scoped variable `goto`.

Consider `break` and `continue` which are essentially `goto`s to escape a loop or repeat it early.

If those do not suffice, then consider the example:

```
repeat: a = (a + 1) % 10;
if (a != 0) goto repeat;
```

C Programming Language
Preprocessor

- C source files are not given directly to the compiler
- They are pre-processed to create the actual source
- For more detailed (and insane) examples, use Google
- Processes '#' directives

`#include` <*file*>
: pastes the pre-processed contents of *file* into the source

`#define` *A* *B*
: causes *A* to be replaced by *B* from this point forward

`#define` *A*`(...)` *B*
: function-like macro definition

`#ifdef` *A*
: conditions the text from here until `#endif` on whether *A* is defined

C Programming Language

Preprocessor Misc

`#include` is not quite like Java's `import`.

- It only includes the header source
- A linker must still incorporate the library
- `#include`-ing a header twice will cause the source to be processed twice
- Headers are often guarded against double inclusion using `#ifdef`

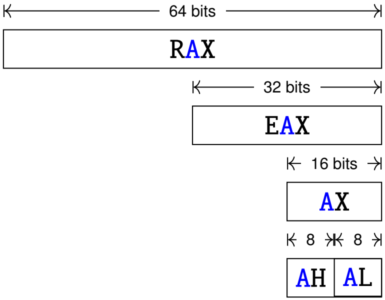
x86_64 Assembly

- Far fewer data types and control structures available
- Registers of fixed sizes, but interpreted based on instruction operand sizes and types
- Access to memory by using memory operands (not load/store)
- Control is implemented via jumps and conditional jumps, basically `goto` only
- Assumes an in-memory stack (top is given by the Stack Pointer)

x86_64 Assembly

Registers

Primarily for signed or unsigned integer types, but they can hold any data 64 bits or less.



$\times\{\mathtt{RAX}, \mathtt{RBX}, \mathtt{RCX}, \mathtt{RDX}\}$

- $\{\mathtt{RSP}_{64}, \mathtt{ESP}_{32}, \mathtt{SP}_{16}\} \times \{\mathtt{RSP}, \mathtt{RBP}, \mathtt{RSI}, \mathtt{RDI}\}$
  - Stack Pointer
  - Base Pointer
  - Source Index
  - Destination Index
- $\{\mathtt{R8}_{64}, \mathtt{R8D}_{32}, \mathtt{R8W}_{16}, \mathtt{R8B}_{8}\} \times \{\mathtt{R8} \ldots \mathtt{R15}\}$
- Many other extension / floating point registers

x86_64 Assembly

Flags

- FLAGS register
- CF PF AF ZF SF DF OF, etc.
- Manage modes and status
- Indicate outcomes of previous instructions
- Some instructions take flag input
  - carry bit (ADC, SBB)
  - conditional flags (JA, JG)

x86_64 Assembly

Instruction Syntax

Composed of opcodes and operands

- opcode: `ADD`, `MOV`, etc.
- operands

Operand	**Intel**	**AT&T**
Register name	`RAX`	`%rax`
Immediate value	`0x8C`	`$0x8C`
Memory address	`[RSI+0x08001F86]`	`0x08001f86(%rsi)`

Intel (Windows, nasm, IDA Pro)

```
OP DST,SRC
```

AT&T (UNIX, GNU as, gdb)

```
op %src,%dst
```

x86_64 Assembly

Basic Data Instructions

Copying / Moving data

```
MOV EAX,EBX
MOV [0x30004040],EBX
```

Arithmetic

```
ADD EAX,EBX
SUB [0x30004040+EBX],0x01
MUL EDX
DIV EBX
```

Floating point

- Use Google

x86_64 Assembly
Basic Control Instructions

Unconditional jump

```
JMP SOME_LABEL
```

Conditional jump

```
CMP EAX,0x07
 JZ SOME_LABEL
```
(zero / equal)

Other conditions / location formats

`JA .+0x10` (above, unsigned / relative)

`JG 0x40001610` (less than, signed / absolute)

x86_64 Assembly

Subroutine Execution

```
CALL SOME_FUNCTION
```

`CALL` pushes `RIP` then jumps to the callee.

```
RET
```

`RET` pops the old `RIP`, returning to the caller.

Caller and callee must agree on conventions for saving registers, passing parameters, and communicating results. This varies by operating system, and it's different for 32-bit than 64-bit.

x86 has a special "Stack Pointer" (RSP), which points to the top of the stack. The stack is a place in memory used to track program execution and store variables. Some instructions, e.g., CALL and RET use the stack automatically. You can also push and pop to the stack using PUSH and POP.

```
PUSH RAX
 POP RBX
```

PUSH decrements RSP by the size of the operand, then writes it to memory at [RSP]. Thus, the stack "grows" into smaller addresses. POP reads memory from [RSP] into the operand, then increments RSP. You can also read and write RSP directly.

x86_64 Assembly

Stack Variables

"Stack Variables," used to implement dynamic or local variables, are variables stored on the stack. Typically, you manually "allocate" space by subtracting the total size from RSP. Then, you access memory relative to the stack pointer. As a convenience, x86 also has a "base pointer" (RBP). Since RSP moves around so much, you can store a less volatile pointer in RSP.

```
PUSH RBP // save the caller's RBP
 MOV RBP,RSP // establish callee's base pointer
 SUB RSP,0x10 // allocate 16 bytes of stack space
 MOV dword ptr [RBP-0x4],0x1234 // a 4-byte variable
```

x86_64 Assembly

Example Blocks

Conditional execution

```
 CMP EBX,10
 JNZ L1
 INC RAX
L1: ...
```

Function call (Microsoft)

```
 MOV RCX,2
CALL SOME_FUNCTION
 MOV RBX,RAX
```

Multiplication

```
 MOV EDI,[ECX+0x5b0]
 MOV EBX,[ECX+0x5b4]
IMUL EDI,EBX
```

Function (Microsoft)

```
SOME_FUNCTION:
 LEA RAX, [RCX*4+RCX]
 RET
```

Recap

C Programming

- Primitive and Derived Types
- Functions and Control Structures
- Preprocessor

x86_64 Programming

- Data, Registers, Flags, Arithmetic
- Control and Subroutines
- Syntax