Compiling C with Clang by examples

$$C \xrightarrow{\text{Clang}} x86$$

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

October 25, 2017

Contents

Structure of the module

Parsing ✓

- Progression from: Language + Logic, Models of Computation
- abstract machines, formal, "mathy"

Compiling C with Clang

- Progression from: Computer Systems + Architecture, C/C++
- not so formal, by example, x86 machine code

Implementing functional languages

- Progression from: functional programming
- builds on abstract machines and C stack

Example

C code

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

x86 generated by Clang

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```

The assembly code does not look much like the source code.
What happened to variables?
What happened to types?

Aims and overview

- We will see some typical C code compiled to x86 assembly by LLVM/Clang
- Emphasise general principles used in almost all compilers
- Use Clang on C and x86 for example and concreteness
- What Clang does, not details of how it does it internally
- Enough to compile some C code by hand line by line
- C language features $\mapsto$ sequence of assembly instructions + addresses
- Various language features on top of vanilla functions
- Optimizations

Clang and LLVM, the bestest and mostest compiler

Clang is the bestest C/C++ compiler
`http://clang.llvm.org`
LLVM is the mostest compiler infrastructure
`http://llvm.org`
Apple uses it
`https://developer.apple.com/xcode/`
Many projects, for example:
Emscripten: An LLVM to JavaScript Compiler
Rust: “a safe, concurrent, practical language” (as per blurb)
A not too technical intro to LLVM:
`http://www.aosabook.org/en/llvm.html`

Using Clang

Please do experiments yourself for seeing how LLVM/Clang compiles C.
Clang comes with XCode on OS X.
If you do not have LLVM on your computer:
ssh into a lab machine and type
module load llvm
To compile, type
clang -S test.c
Then the assembly code will be in test.s
Function frodo will be labelled frodo: in test.s
For optimization, use
clang -S -O3 test.c

Target architecture for Clang output

We will only need a tiny subset of assembly.
Quite readable.
Instruction we will need:

```
mov push pop call ret jmp add mul test be lea
```

The call instruction pushes the current instruction pointer onto the stack as the return address
ret pops the return address from the stack and makes it the new instruction pointer
A nice target architecture should have lots of general-purpose registers with indexed addressing.
Like RISC, but x86 is getting there in the 64-bit architecture

Assembly generated by clang is x86 in AT&T syntax

mov syntax is target-last:
`mov` x y is like y = x ;
r prefix on registers means 64 bit register
movq etc: q suffix means quadword = 64 bits
% register
$ constant
`%rbp` = base pointer = frame pointer in general terminology
`%rsp` = stack pointer, push and pop use it
indexed addressing `-24(%rbp)`

Typical C code to compile

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

Parameters/arguments:
x and y
Local/automatic variables
a and b

More precisely, x and y are *formal* parameters.
In a call f(1,2), 1 and 2 are the *actual* parameters.
We will use the words "parameter" and "argument" interchangeably.

Two big ideas in compiling functions

stack $\leftrightarrow$ recursion

compare: parsing stack

many abstract and not so abstract machines use stacks

including JVM

In C: one stack frame per function call

Names $\rightarrow$ indices

Names can be compiled into indices, discovered many times

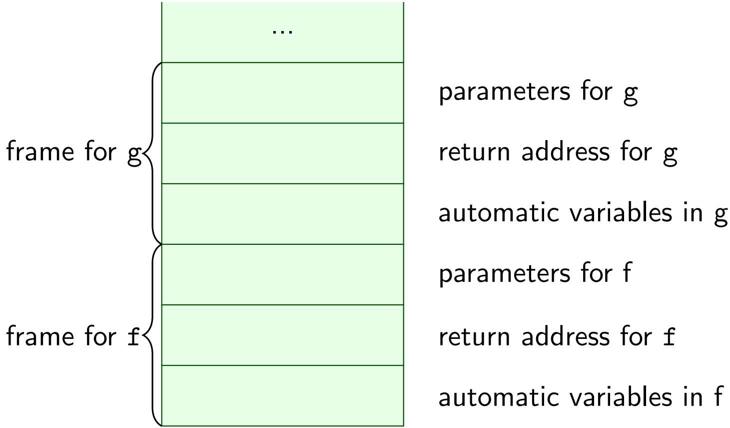
In C: variables become small integers to be added to the base pointer

Stack frame details

The details differ between architectures (e.g., x86, ARM, SPARC)
Ingredients of stack frames, in various order, some may be missing:
return address
parameters
local vars
saved frame pointer
caller or callee saved registers
static link (in Pascal and Algol, but not in C)
this pointer for member functions (in C++)

A traditional stack layout (but not Clang)

Convention: we draw the stack growing downwards on the page.
Suppose function `g` calls function `f`.



There may be more in the frame, e.g. saved registers

What about recursive functions?

Consider the standard example of recursion:

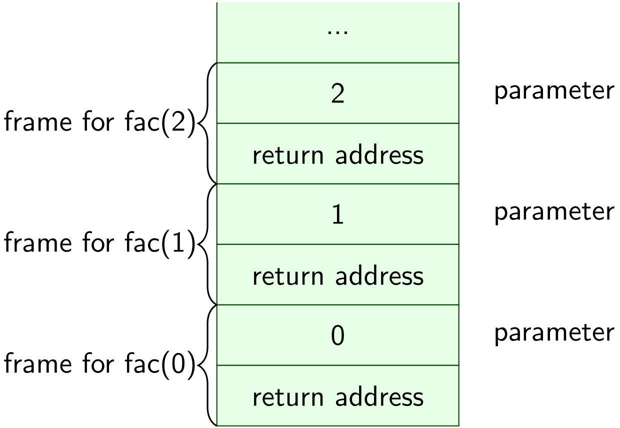
```
long factorial(long n)
{
  if(n == 0)
    return 1;
  else
    return factorial(n - 1) * n;
}
```

Call stack: one frame per function call

Recursion example: fac(n) calls fac(n - 1). Each recursive call gets a smaller parameter.

The return address points into the code segments, not the stack or heap.

What are the return addresses?



Return address example

```
long factorial(long n)
{
  if(n == 0)
    return 1;
  else
    return factorial(n - 1) * n;
}
```

The return address is a pointer to the compiled code. The returned value is returned into the hole ◯ position in the last statement,

```
return ◯ * n;
```

Thus when the function returns, 1 is plugged into the hole, then 2, then 6, ...

The return address represents a continuation.

Calling conventions and stack frame layout

The calling convention differs between compilers and architectures

Old school:

push arguments onto stack, then do a call instruction (which pushes return address)

Modern architectures have many registers

$\Rightarrow$ pass arguments in registers when possible; Clang does this

Some RISC architectures put return address into a link register

more exotic: SPARC has register windows for parameter passing

Stack frame in clang C calling convention on x86

Clang passes parameters in registers `rdi`, `rds`, ...
The parameters also have a slot in the frame

return addr	
old bp	← frame/base pointer
parameter n	
⋮	
parameter 1	
local var	
⋮	
local var	← stack pointer, if used

Clang function idiom

```
http://llvm.org/docs/LangRef.html#calling-conventions

f:
pushq %rbp
movq %rsp, %rbp
    ... body of function f
popq %rbp
ret
```

parameters are passed in registers rdi, rsi
return value is passed in register rax

Computing the index in the frame

Simple in principle:
walk over the syntax tree and keep track of declarations
The declarations tell us the size: long x means x needs 8 bytes
That is why C has type declarations in the first place

```
long f(long x, long y) // put y at -8 and x at -16
{
    long a;    // put a at -24
    long b;    // put b at -32
    a = x;    // now we know where a and x are
              // relative to rbp
}
```

Exercise: what happens if we also have char and float declarations?

Clang stack frame example

```
long f(long x, long y) // put y at -8 and x at -16
{
    long a;    // put a at -24
    long b;    // put b at -32
    ...
}
```

return addr	
old rbp	← base pointer rbp
x	← rbp - 8
y	← rbp - 16
a	← rbp - 24
b	← rbp - 32

Compiled with clang -S

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

```
f:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq -8(%rbp), %rsi
addq $42, %rsi
movq %rsi, -24(%rbp)
movq -16(%rbp), %rsi
addq $23, %rsi
movq %rsi, -32(%rbp)
movq -24(%rbp), %rsi
imulq -32(%rbp), %rsi
movq %rsi, %rax
popq %rbp
ret
```

$$\begin{array}{rcl}
\mathtt{x} & \mapsto & \mathtt{rdi} \\
\mathtt{y} & \mapsto & \mathtt{rsi} \\
\mathtt{x} & \mapsto & \mathtt{rbp} - 8 \\
\mathtt{y} & \mapsto & \mathtt{rbp} - 16 \\
\mathtt{a} & \mapsto & \mathtt{rbp} - 24 \\
\mathtt{b} & \mapsto & \mathtt{rbp} - 32 \\
\text{return value} & \mapsto & \mathtt{rax}
\end{array}$$

Optimization: compiled with clang -S -O3

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```

`lea` = load effective address

can be used for indexed addressing, but here it is just used for addition

`y + 23`

Leaf functions

- at run time, the function calls form a tree, a little like a parse tree
- A "leaf" function is one that does not call any functions.
- It is a leaf in the control flow graph/tree:
  leaf = node without children.
- Leaf functions can be compiled more simply:
- no need to adjust stack pointer
- no need to save registers into frame
- Some leaf functions can work entirely on registers, which is efficient
- many function calls are leaf function calls

Many arguments ⇒ spill into the stack

Some passed on the stack, not in registers. These have positive indices relative to the stack pointer.

```
long a(long x1, long x2,
long x3, long x4, long x5,
long x6, long x7, long x8)
{
  return x1 + x7 + x8;
}
```

```
a:
addq 8(%rsp), %rdi
addq 16(%rsp), %rdi
movq %rdi, %rax
ret
```

We will not use lots of arguments. While a C compiler must allow it, it is not good style.

Stretch exercise on calling conventions

Here is a function definition very similar to one in the original manual on B, the predecessor of C.

```
void printn(n, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
/* print n arguments as integers */
{
  int i, *p;
  p = &x0;
  for(i=0; i<n; i++) printf("%d\n", p[i]);
}
```

Explain how this function could work. You may assume that all parameters are passed on the stack, and in reverse order.
Will it still work in Clang?

Calling functions

A function call

```
f(E1, ... , En)
```

is broken down into into steps like this

```
arg1 = E1;
...
argn = En;
call f;
```

where `argi` are the argument positions of the calling convention.
Note: in C, the order for computing `argi` is unspecified to give the compiler the freedom to optimize.
If a function calls some function (i.e., it is non-leaf), it needs to adjust the stack pointer.

Calling another function example

```c
long f(long x)
{
  return g(x + 2) - 7;
}
```

```
f:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
movq -8(%rbp), %rdi
addq $2, %rdi
callq g
subq $7, %rax
addq $16, %rsp
popq %rbp
ret
```

The stack pointer rsp is updated: first by -16 then by +16.

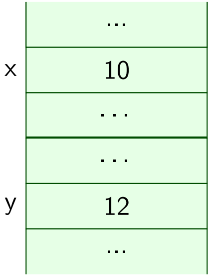
```
subq $16, %rsp
...
addq $16, %rsp
```

Referring to other stack frames

- A function can refer to its own stack frame via the frame pointer
- Depending on the language, it may also refer to frames of other calls
- In Pascal: via static link or using var parameters
- In C: when taking address of local variables or parameters
- In C++: call-by-reference
- in C, call-by-reference is reduced to call-by-value + pointer

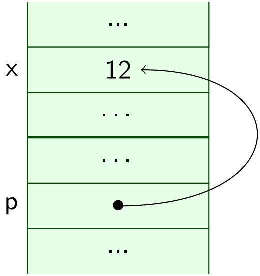
Call-by-value + assignment modifies only local copy

```
void f(int y)
{
    y = y + 2; // draw stack after this statement
}

void g()
{
    int x = 10;
    f(x);
}
```



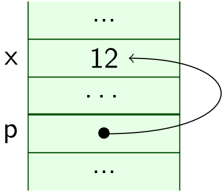
Call by reference in C = call by value + pointer

```
void f(int *p)
{
    *p = *p + 2; // draw stack after this statement
}

void g()
{
    int x = 10;
    f(&x);
}
```



Escaping variables and stack frames

The compiler normally tries to place variables in registers when optimizing.
The frame slot may be ignored.
However, if we apply & to a variable, its value must be kept in the frame slot.
The variable "escapes" from the frame in the sense that it can be referred to from outside.



The compiler must check whether & is applied before it can optimize to a register

Call with pointer: calling function

```
void f(long x, long *p)
{
  *p = x;
}

long g()
{
  long a = 42;
  f(a + 1, &a);
  return a;
}
```

```
g:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
leaq -8(%rbp), %rsi
movq $42, -8(%rbp)
movq -8(%rbp), %rax
addq $1, %rax
movq %rax, %rdi
callq f
movq -8(%rbp), %rax
addq $16, %rsp
popq %rbp
ret
```

a corresponds to `-8(%rbp)`

Note how lea corresponds to &

Call with pointer: called function

```c
void f(long x, long *p)
{
  *p = x;
}

long g()
{
  long a = 42;
  f(a + 1, &a);
  return a;
}
```

```
f:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq -8(%rbp), %rsi
movq -16(%rbp), %rdi
movq %rsi, (%rdi)
popq %rbp
ret
```

Call with pointer: optimized with -O3

```
void f(long x, long *p)
{
  *p = x;
}
```

```
f:
movq %rdi, (%rsi)
ret
```

Pointers exercise: my god, it's full of stars

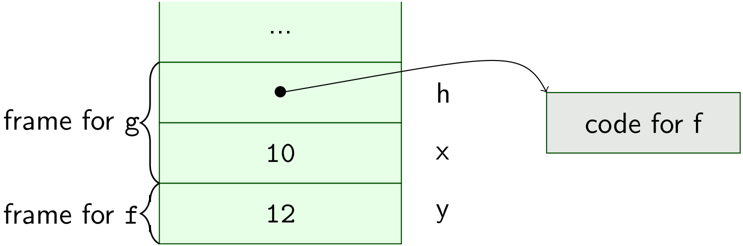
Suppose there are multiple pointer dereferences:

```
void f(long x, long ***p)
{
  ***p = x;
}
```

How should this be compiled? Hint: multiple indirect addressing, `(%rdi)`.
Draw the memory with arrows for pointers as illustration.

Function pointer as function parameter

```
void g(void (*h)(int))
{
    int x = 10;
    h(x + 2);
}

void f(int y) { ... }

... g(f) ...
```



Function pointer as a parameter

```c
long f(long (*g)(long))
{
  return g(42) + 2;
}
```

```
f:
pushq %rbp
movq %rsp, %rbp
movabsq $42, %rax
movq %rdi, -8(%rbp)
movq %rax, %rdi
callq *-8(%rbp)
addq $16, %rsp
popq %rbp
ret
```

In C, the $*$ in calling a function pointer can be left implicit.
In the assembly, it is `callq *-8(%rbp)`

Compiling, stack frames, and security

- understanding compiling is crucial for understanding memory corruption attacks
- most devastating attacks: "arbitrary code execution"
- classic attack: "smashing the stack for fun and profit"
- modern compilers automagically defend against it: stack canaries
- arms race between attack and defence
- advanced techniques: return-oriented programming
- attacker uses a special-purpose compiler
- re-use code snippets ("gadgets") as instruction set
- see for instance `https://www.researchgate.net/profile/Marco_Prandini/publication/260635323_Return-Oriented_Programming/links/560e942a08ae0fc513ed7710.pdf`

Buffer overflow on the call stack

```
int vulnerable_function()
{
    int winner = 0; // suppose this is security-critical
    char name[8];   // this is the buffer to be overflown

    printf("Please enter your name:\n");
    fgets(name, 200, stdin);
    ...
}
```

Input `blahblahbl` overflows the string variable on the stack:

	return address
`bl\0`	winner
`blahblah`	name

Note: the call stack grows towards lower machine addresses.

Stretch exercise on buffer overflow

Here is some code vulnerable to classic buffer overflow:

```
void bufwin()
{
  int winner = 0; // suppose this is security-critical
  char name[10]; // automatic, so stack-allocated
  printf("Please enter your name:\n");
  fgets(name, 200, stdin); // overflow array name
  if (winner) // attacker overflowed name into winner
    printf("You WIN, %s\n", name);
  else
    printf("You LOSE, %s\n", name);
}
```

On modern compilers, this attack will not succeed. Look into the assembly code and see how the defence works.

Optimizations

There are many different optimizations, for example:

- constant propagation: for example, if you have `x = 42`, the compiler may replace some occurrences of `x` by 42
- compile-time evaluation: for example, given `x = 42 + 23`, the compiler may calculate that $42+23 = 65$
- dead code elimination: if some code can never be reached, the compiler deletes it; it may also give warnings
- function linling, see below
- tail call optimization, see below

Exercise: write some code with opportunities for constant propagation, and see how Clang compiles it.

Function inlining

- Function inlining = do a function call at compile time.
- Saves the cost of a function call at runtime.
- Copy body of called function + fill in arguments.
- C even has a keyword `inline`, but Clang is smart enough to know when to inline anyway
- Modern C compilers do inlining, so C macros can be avoided.
- inlining may increase code size but gain speed
- inlining may enable further optimizations
- Modern C++ uses templates, which give the compiler many opportunities for inlining.
- Inlining has a clean theoretical foundation: beta reduction in the lambda calculus.

Function inlining example

```
long sq(long x)
{
  return x * x;
}

long f(long y)
{
  long z = sq(++y);
  return z;
}
```

```
f:
incq %rdi
imulq %rdi, %rdi
movq %rdi, %rax
ret
```

Note: cannot just do `++y * ++y`

Inlining exercise

Simplify the following C code as much as possible by performing function inlining.

```
long sq(long x) { return x * x; }

long f(long (*g)(long))
{
  return g(42) + 2;
}

long applyftosq()
{
  return f(sq);
}
```

Tail call optimization

Tail position = the last thing that happens inside a function body before the return.
Here f is in tail position:

```
return f(E);
```

f is not in tail position here:

```
return 5 + f(E);
```

or here:

```
return f(E) - 6;
```

A function call in tail position can be compiled as a jump.

Function pointer as a parameter

```c
long f(long (*g)(long))
{
  return g(42);
}
```

```
f:
pushq %rbp
movq %rsp, %rbp
movabsq $42, %rax
movq %rdi, -8(%rbp)
movq %rax, %rdi
callq *-8(%rbp)
addq $16, %rsp
popq %rbp
ret
```

Function as a parameter, tailcall optimized

```c
long f(long (*g)(long))
{
  return g(42);
}
```

```
f:
movq %rdi, %rax
movl $42, %edi
jmpq *%rax  # TAILCALL
```

Tailcall exercise

Explain which of these (if any) is a tail call.

```
return (*g)();
```

```
return *g();
```

Stretch exercise: vectorizing optimizations

Take some simple code, say factorial.

Compile it on the latest version of clang with all optimizations enabled.

Try to understand what the resulting code does.

It will probably look very different to the source code.

Loop unrolling to enable parallel computation.

Compiling structures and objects

Same idea as in stack frames:
access in memory via pointer + index
Structure definition tells the compiler the size and indices of the members.
No code is produced for a struct definition on its own.
But the compiler's symbol table is extended: it knows about the member names and their types.
Structure access then uses indexed addressing using those indices.

```
struct S {
  T1 x;
  T2 y;
};
```

Compiling structures and objects

Same idea as in stack frames:
access in memory via pointer + index
Structure definition tells the compiler the size and indices of the members.
No code is produced for a struct definition on its own.
But the compiler's symbol table is extended: it knows about the member names and their types.
Structure access then uses indexed addressing using those indices.

```
struct S {
  T1 x;
  T2 y;
};
```

x is a index 0
y is at `sizeof(T2)` + padding for alignment

Structure access

```c
struct S {
  long x;
  long y;
};

void s(struct S *p)
{
  p->x = 23;
  p->y = 45;
}
```

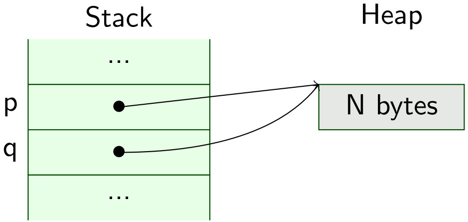
```
s:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    $23, (%rdi)
    movq    $45, 8(%rdi)
    popq    %rbp
    retq
```

$$\begin{array}{lll} \mathtt{x} & \mapsto & 0 \\ \mathtt{y} & \mapsto & 8 \end{array}$$

Stack and heap

Structures and objects are mainly on the heap, via `malloc` or `new`. The compiled code follows a pointer, whether this leads into the stack or heap.

```
p = malloc(N);
q = p;
```



Member functions of objects

- In C++, functions defined inside classes ("member functions") have access to other members of the class.
- Implemented using `this` pointer.
- The `this` pointer points at the object (not the class).
- In a call of a member function, the `this` pointer is passed like an additional parameter.
- Dereferencing this extra parameter accesses gives access to members.

Member function access to object example

```
class C {
  long n;
public:
  long f()
  {
    return n;
  }
};
```

Code for f:

```
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq -8(%rbp), %rdi
movq (%rdi), %rax
popq %rbp
ret
```

Exercise: draw the stack and heap to illustrate how the function above works, assuming the object has been allocated on the heap.

OO and LLVM

LLVM does not have a built-in notion of object.
Inside LLVM (in the IR), OO is decomposed into pointers, functions, and structures.

Exercise on structures

Suppose the following structure definition is given:

```
struct S {
  struct S *next;
  struct S *prev;
  long data;
};
```

Suppose a pointer `p` is held in register `%rsi`. Translate the following C statements to assembly:

```
p->data = 12345;
p->next = p->prev->next;
```

Stretch exercise on structures

Building on what you have learned about structures, how are C unions compiled?
For example, consider

```
union u {
  A x;
  B y;
};
```

What are the adresses for `A` and `B`?

More on C++

If you are interested in object-orientation, you can do more experiments by observing how Clang compiles C++ code.
Example: virtual function table, implemented using yet more pointers.
But compiled C++ is harder to read than for C, due to name mangling.

Summary of compiling C

- stack for function call, including recursive functions
- return address and frame pointer
- access to variables by frame pointer + offset
- names are compiled into integers, the offset
- C pointers can be compiled easily, including function pointers
- C structs also get compiled as base + offset

Where to next: λ

- C functions are deliberately simple: no nested functions
- is there a more powerful version of "first-class" function in programming languages?
- what is a good intermediate language for optimizations like inlining?
- answer: λ calculus as a theory of first-class functions