Compiler Design

Lecture 13: Code generation : Memory management and function call (EaC Chapter 6&7)

Christophe Dubach

Winter 2022

Timestamp: 2022/02/15 11:05:00

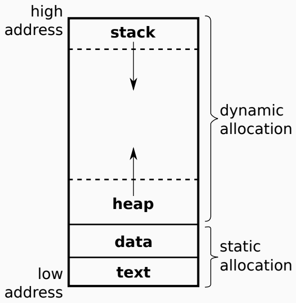
Table of contents

Memory management

Static versus Dynamic

- Static allocation: storage can be allocated directly by the compiler by simply looking at the program at compile-time. This implies that the compiler can infer storage size information.
- Dynamic allocation: storage needs to be allocated at run-time due to unknown size or function calls.

Heap, Stack, Static storage



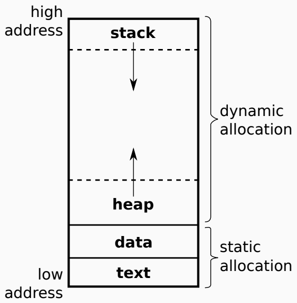
Static storage:

- Text: instructions
- Data
    - global variables
    - string literals
    - global arrays of fixed size
    - ...

Dynamic Storage:

- Heap: `malloc`
- Stack:
    - local variables
    - function arguments/return values
    - saved registers
    - register spilling (register allocation)

Example



```
char c;                  data
int arr[4];              data
void foo() {
  int arr2[3];           stack
  int* ptr =             stack
    (int*) malloc(sizeof(int)*2);    heap
  ...
  {
    int b;               stack
    ...
    bar("hello");        data
  }
  ...
}
```

Memory management

Static Allocation & Alignment

Scalars

Typically

- int and pointer types (*e.g.* char∗, int∗, void∗) are 32 bits (4 byte).
- char is 1 byte

However, it depends on the data alignment of the architecture.
A single char might occupy 4 bytes if data alignment is 4 bytes.

Example (static allocation):

```
.data
c: .space 1 # char
i: .space 4 # int

.text
lb $t0, c
lw $t1, i   # error!
```

Miss-aligned load
😠 forbidden!

```
.data
c: .space 4 # char
i: .space 4 # int

.text
lb $t0, c
lw $t1, i   # all good!
```

With padding added
All good ☺

Structures

In a C structure, all values are aligned to the data alignment of the architecture (unless `packed` directive is used).

Example C code (static allocation)

```
struct myStruct_t {
  char c;
  int x;
};
struct myStruct_t ms;
...
```

In this example, it is as if the value c uses 4 bytes of data.

```
.data
ms_myStruct_t_c:  .space  4
ms_myStruct_t_x:  .space  4

.text
...
```

Arrays

In contrast to structs, arrays are always compact in C with extra padding added in the end if required.

Example C code (static allocation):

```
char arr[7];
int i;
...
```

Corresponding assembly code:

```
.data
arr: .space 8
i:   .space 4

.text
...
```

Memory management

Stack allocation

Stack variable allocation

The compiler needs to keep track of local variables in functions.

These cannot be allocated statically (*i.e.* text section).

```
int foo(int i) {
  int a;
  a = 1;
  if (i==0)
    foo(1);
  print(a);
  a = 2;
}
void bar() {
  foo(0);
}
```

If a allocated statically, what is printed?

1

2

Local variables must be stored on the stack!

How to keep track of local variables on the stack?
⇒ Could use the `stack pointer`.

⚠ Problem: stack pointer can move.

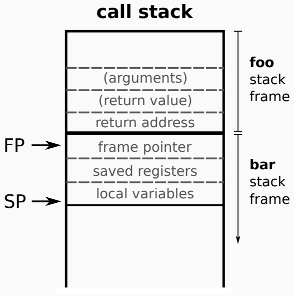
- *e.g.* dynamic memory allocation on the stack

Solution: use another pointer, the frame pointer

Frame pointer

- The frame pointer must be initialised to the value of the stack pointer, just when entering the function.
- Access to variables allocated on the stack can then be determined as a fixed offset from the frame pointer.

- The frame pointer (FP) always points to the beginning of the local variables of the current function, just after the arguments (if any).
- The stack pointer (SP) always points at the top of the stack (points to the last element pushed).



Memory management

Address of expressions

Visitor for generating addresses

Sometimes, the compiler needs to know the address of an expression (*e.g.* assignment, address-of operator):

```
struct vec_t {
  int x;
  int y;
};

void foo() {
  int i;
  struct vec_t v;
  int arr[10];

  i = 2;
  v.x = 3;
  arr[2] = 5;
}
```

Visitor for generating addresses

Specialized visitor that produces the address of an expression:

AddrGen Visitor

```
Register visitBinOp(BinOp bo) {
  error("Cannot request address for BinOP");
}

Register visitIntLiteral(IntLiteral it) {
  error("Cannot request address for IntLiteral");
}

Register visitVarExpr(VarExpr v) {
  Register resReg = newVirtualRegister();
  if (v.vd.isStaticAllocated())
    emit("la", resReg, v.vd.label);
  else if (v.cd.isStackAllocated())
    ...
  return resReg;
}

...
```

The `AddrGen` visitor will be called from the “normal” code generator visitor when needed:

Code generator visitor

```
...
Register visitAssign(Assign a) {
  Register addrReg = a.lhs.accept(new AddrGen());
  Register valReg  = a.rhs.accept(this);
  emit("sw", valReg, addrReg);
  return null; // different from C
}
```

Last words

Examples above have assumed variable stores an interger.

In case a variable represents an array or struct, you have to be careful:

- array are passed by reference
  (treat them exactly like pointers, no big deal)
- struct are passed by values

Assigning between two structs means copying field by field. Hence your code generator must check the type of variables when encountering an assignment and handle structures correctly.

Similar problem for struct and function call. Arguments and return values that are struct are passed by values.

Code with struct assignment:

```c
struct vec_t {
  int x;  int y;
};
void foo() {
  struct vec_t v1;
  struct vec_t v2;
  v1 = v2;
}
```

Equivalent to:

```c
struct vec_t {
  int x;  int y;
};
void foo() {
  struct vec_t v1;
  struct vec_t v2;
  v1.x = v2.x;
  v1.y = v2.y;
}
```

Pro tip

Instead of handling this complexity in the code generator, write a pass that runs before code generation to transform the AST to “inline” the struct assignments field by field.

Function calls

Function calls

```
int bar(int a) {
   return 3+a;
}
void foo() {
  ...
  bar(4)
  ...
}
```

- foo is the caller
- bar is the callee

What happens during a function call?

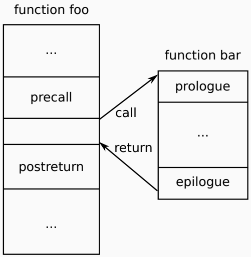
- The caller needs to pass the arguments to the callee
- The callee needs to pass the return value to the caller

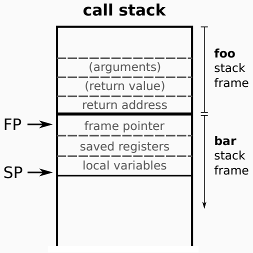
But also:

- The values stored in registers needs to be saved somehow.
- Need to remember where we came from to return to the call site.

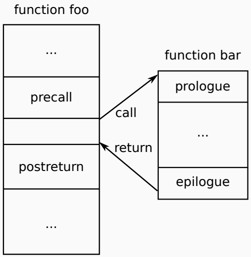
Possible convention:

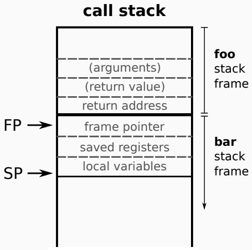
- precall:
    - pass the arguments via registers or push on the stack
    - reserve space on stack for return value (if needed)
    - push return address on the stack
- postreturn:
    - restore return address from the stack
    - read the return value from dedicated register or stack
    - reset stack pointer





- prologue:
    - push frame pointer onto the stack
    - initialise the frame pointer
    - save all the saved registers onto the stack
    - reserve space on the stack for local variables
- epilogue:
    - restore saved registers from the stack
    - restore the stack pointer
    - restore the frame pointer from the stack





💡 Other conventions are possible.

To simplify (for your project), we suggest you:

- save all the registers used by a function onto the stack;
- pass all arguments and return value via the stack
  (this is needed anyway when there are more than four arguments).

Example (callee)

```
int bar(int a) {
   int b;
   return 3+a;
}
```

```
bar:

addi $sp, $sp, -4   #
sw   $fp, ($sp)     # push frame pointer on the stack

move $fp, $sp       # initialise the frame pointer

addi $sp, $sp, -4   #
sw   $t0, ($sp)     # push $t0 onto the stack
addi $sp, $sp, -4   #
sw   $t1, ($sp)     # push $t1 onto the stack

addi $sp, $sp, -4   # reserve space on stack for b

li   $t0, 3         # load 3 into $t0

lw   $t1, 12($fp)   # load first argument from stack
add  $t0, $t0, $t1  # add $t0 and first argument

sw   $t0, 8($fp)    # copy the return value on stack

lw   $t0, -4($fp)   # restore $t0
lw   $t1, -8($fp)   # restore $t1

addi $sp, $sp, 16   # restore stack pointer

lw   $fp, ($fp)     # restore the frame pointer

jr   $ra            # jumps to return address
```

Example (caller)

```
void foo() {
  ...
  bar(4)
  ...
}
```

```
foo:
...

li   $t0, 4          #
addi $sp, $sp, -4    #
sw   $t0, ($sp)      # push argument on stack

addi $sp, $sp, -4    # reserve space on stack for return value

addi $sp, $sp, -4    #
sw   $ra, ($sp)      # push return address on stack

jal  bar             # call function

lw   $ra, ($sp)      # restore return address from the stack

lw   $t0, 4($sp)     # read return value from stack

addi $sp, $sp, 12    # reset stack pointer
...
```

Final words

Beware of passing structs! Needs to be passed by value.

C code example:

```
struct vec_t {
  int x;
  int y;
}

int foo(struct vec_t v) {
  return v.x;
}

void bar() {
  struct vect_t myvec;
  int i;
  i = foo(myvec);
}
```

Equivalent C code:

```
struct vec_t {
  int x;
  int y;
}

void foo(int x, int y){
  struct vec_t v;
  v.x = x; v.y = y;
  return v.x;
}

void bar() {
  struct vect_t myvec;
  int i;
  i = foo(myvec.x, myvec.y);
}
```

Final words

Beware of returning structs! Needs to be passed by value.

C code example:

```
struct vec_t {
  int x;
  int y;
}

struct vec_t foo() {
  struct vec_t v;
  v.x = 0; v.y = 1;
  return v;
}

void bar() {
  struct vect_t myvec;
  myvec = foo();
}
```

Equivalent C code:

```
struct vec_t {
  int x;
  int y;
}

void foo(struct vec_t * result){
  struct vec_t v;
  v.x = 0; v.y = 1;
  *result = v;
}

void bar() {
  struct vect_t myvec;
  struct vect_t result;
  foo(&result);
  myvec = result;
}
```

Pro tip

Instead of handling this complexity in the code generator, you could write a pass that runs before code generation to transform the AST to deal with returning/passing struct to functions.

You may have to call this pass together with other transformations passes iteratively until nothing changes in the AST (*e.g.* in case of nested structures).

Next lecture

Naive register allocator.