Compiler Construction
SMD163

Lecture 9: Implementing functions and objects

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander
Contains material generously provided by Mark P. Jones

COMPUTER SCIENCE
AND ELECTRICAL ENGINEERING
LULEÅ UNIVERSITY OF TECHNOLOGY

Implementing Functions

Functions and Procedures

- Functions are an important and widespread *abstraction* mechanism in many different programming languages:
  - A first step to reusable code.
  - A wide spectrum of language designs.
- A procedure is a function that doesn't return any useful value; it is used for its effect only.
- How are functions implemented? What kind of data structures do we require?

Designs on Functions:

There are a wide range of different design choices:

- What types of arguments can be passed as parameters or returned as results?
- Can the number of arguments vary?
- Are recursive functions permitted?
- Are nested functions permitted?
- Are functions first-class values?
- Can functions have side-effects?

Implementations of Functions:

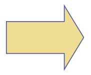
- How does the function determine the correct return address when its done?
- How are parameters passed to functions?
- How are function results returned?
- Where are the values of local variables stored?
- What happens to local variables when the function returns?
- The answers to these questions determine the calling convention for a particular implementation.

A very naive Calling Convention:

- As a first attempt, we might assign fixed variables for function parameters, locals, and results.

```
int f(int x) {
   int y;
   ... x ... y ...
   return ...
}

... f(42) ...
```



```
        int     f_x;
        int     f_retv, f_y;
        address f_return;

        f_x = 42;
        f_return = lab1;
        goto f;
lab1:   temp = f_retv;
        ...
        ... temp ...

f:      ... f_x ... f_y ...
        f_retv = ...
        goto *f_return;
```

Evolving a Better Convention:

- Our naive calling convention has many limitations, and is not used in modern systems.
- But we can develop it, focusing on one aspect at a time, to obtain a better calling convention.
- As we do, the pseudo-code on the previous code will start to look more an more like assembly language …

Finding Return Addresses:

- A function may be called from many different points in a single program – The code for the function must be able to find the appropriate point in the program to "return" to after a function call is complete.
- If you call a number of functions, you'd expect the last one called to be the first to return.
- Thus a LIFO, or stack structure, will probably play an important role …

Using call and ret:

◆ Using the call and ret instructions:

```
int f(int x) {
   int y;
   ... x ... y ...
   return ...
}

... f(42) ...
```

➡

```
            int  f_x;
            int  f_retv, f_y;

            f_x = 42;
            call f;
            temp = f_retv;
            ...
            ... temp ...

f:          ... f_x ... f_y ...
            f_retv = ...
            ret
```

◆ Allows nested and recursive calls …

Finding the Return Value:

It is awkward and inefficient to access the return value of a function through a global variable.

Much better to use a register!

```
            int f_x, f_y;
            f_x  = 42;
            call f;
            temp = %eax;
            ...
            ... temp ...

f:          ... f_x ... f_y ...
            %eax = ...
            ret
```

Potential Complications:

◆ The return value may not fit into a register!
- For example, floating point values on a 386 require 64 bits … %eax only holds 32 bits.

◆ Compilation schemes need to take the demands by function calls into account:
- We might compile f(x) + f(y) as:

```
movl  x, f_x
call  f
pushl %eax        <<--- save %eax
movl  y, f_y
call  f
movl  %eax, %ebx
popl  %eax        <<--- restore %eax
addl  %ebx,%eax
```

- Without special action, the result of the first call might be overwritten by the second.

A Curious Asymmetry:

◆ Most (all?) programming languages have a notion of functions …

◆ … and most of these allow functions with multiple arguments …

◆ … only very few allow functions with multiple results.

◆ There's no technical reason for that!

Finding Parameters:

- Using fixed variables to pass function parameters gives strange results if there are recursive function calls:

```
int fact(int n) {
  if (n==0) then
    return 1;
  else
    return n*fact(n-1);
}
```

```
fact:   movl  fact_n,%eax
        cmpl% $0,%eax
        jnz   L0
        movl  $1,%eax
        ret
L0:     < ... fact_n = fact_n-1 ... >
        call  fact
        imull fact_n,%eax
        ret
```

- Intuitively, we expect a new, but <u>temporary</u>, variable n each time we enter the fact function.

(Re)using the Stack:

- Parameters can be passed to a function by pushing values onto the stack. For example, a call:

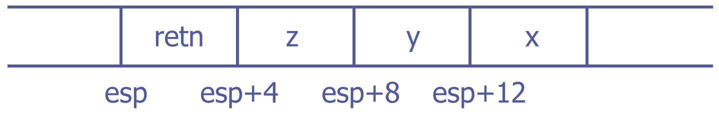
```
f(x,y,z)
```

might be implemented by:

```
pushl   x           -- save parameters
pushl   y
pushl   z
call    f           -- call function
addl    $12,%esp    -- pop parameters
```

Accessing Parameters:

- How do we access the values of function parameters in the body of that function?



- Indexed addressing, relative to the stack pointer:
  - For example, 4(%esp) refers to the contents of the memory at address esp+4.
  - We replace uses of the parameter name x with 12(%esp)… similarly for other parameters …
- But this mapping can change during evaluation …

As the Stack Shifts …

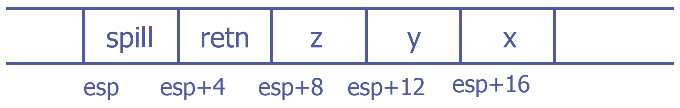
- For example, suppose that wanted to compile:

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

```
f: movl   12(%esp),%eax
   imull  %eax,%eax
   pushl  %eax        -- save %eax
   movl   12(%esp),%eax
   imull  %eax,%eax
   popl   %ebx
   addl   %ebx,%eax
   ret
```

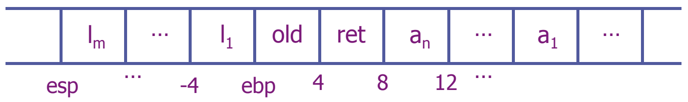
NOT 8(%esp)!

- After the pushl, the stack looks like:



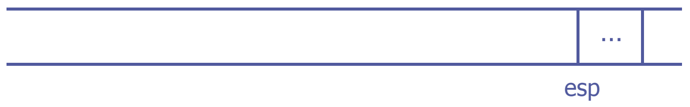
Using a Base Pointer:

- The standard calling convention on the 386 uses the base pointer register, ebp, register to access parameters.
- For each function/procedure call, we set up an activation record or stack frame:

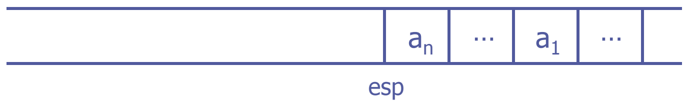


  - Actual parameters: a_1, …, a_n are the function’s arguments. We can refer to a_n as 8(%ebp), etc.
  - Return address: ret is the return address.
  - Base Pointer: old is the base pointer for the calling function.
  - Local variables: l_1, …, l_m are the function’s local variables. We can refer to l_1 as -4(%ebp), etc.

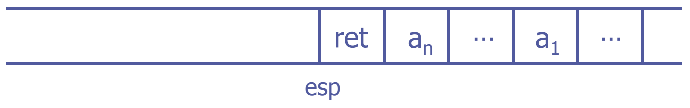
Building the Stack Frame:

- 

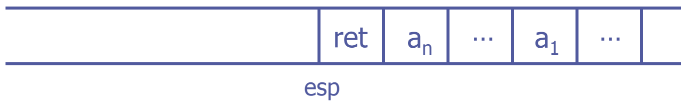
- The caller starts by pushing the arguments:



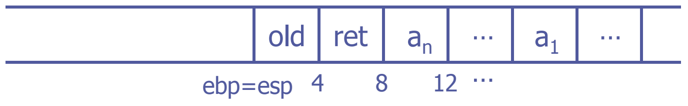
- Then it executes a call instruction, which pushes the return address:



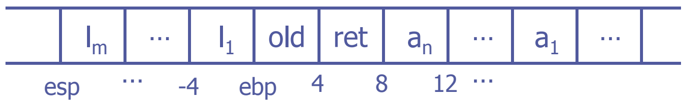
Building the Stack Frame (ctd):

- 

- The callee saves the old base pointer, and sets a new value:



- Then it decrements the stack pointer to reserve space for local parameters:



Function Prologue:

- The code that builds the stack frame at the beginning of a function is called the prologue:
  - At the beginning of a function body, the parameters and return address have already been pushed onto the stack. We need to:

```
push  %ebp                  ; save old base pointer
mov   %esp,%ebp             ; and set new value
```

    If local variables taking M bytes of storage are required, then we need to reserve space for them:

```
subl  $M,%esp               ; allocate space for locals
```

Function Epilogue:

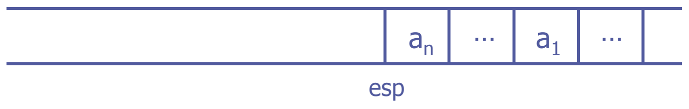
- When the function completes, we must dismantle the stack frame and return the machine to the state it was in before the call. The code to do this is called the epilogue:
  - Running the previous procedure in reverse:

```
mov   %ebp,%esp          ; discard locals and temps
popl  %ebp               ; recover original base ptr
ret                      ; return
```

  - On a 386, the first two instructions here can be replaced by the more efficient, but otherwise equivalent, leave instruction.

Removing the Parameters:

- Once we return to the caller, the result of the function is in eax, but the parameters are still on the stack.



- We restore the stack pointer to its original value by adding on the number of bytes used by the parameters:

```
addl   $N, %esp
```

- If no parameters were passed, then this step can be omitted.

Parameters in Registers?

- It can also be beneficial to allow parameters to be passed to functions in registers.
- But, again, caller and callee must have a protocol that allows them to agree how this is done.
- Further complications arise in languages that allow programmers to take the address of a parameter … in that case, the callee must still write the parameter into memory, even if is passed in a register.

Saving Registers:

- What happens if we call a function in the middle of an expression when there are values in registers that we want to preserve?
- Unless the function was written with this in mind, it might overwrite those registers with other values …

But who should save them?

- Some part of the code has to take responsibility for preserving register contents (e.g., by saving them on the stack.)
  - The caller: knows which registers have values that must be preserved, but doesn't know which the callee will actually use.
  - The callee: … the same situation, reversed.
  - Neither one has enough information to guarantee that it can do a good job.
- We can solve these problems using interprocedural register allocation if the caller and callee are in the same compilation unit.

Alternative Approaches:

- If there were more registers, we might designate some of them as caller saves, and others as callee saves.
- … but figuring out how to use them to best effect can be hard.

A Caller Saves protocol:

- Otherwise, we need to adopt a policy/protocol for saving registers.
- For the MiniJava compiler, we will use the caller saves convention.
- For example, if ebx and ecx contain values that we want to keep, then we

```
pushl %ebx ; pushl %ecx
```

before the call, and

```
popl %ecx; popl %ebx
```

after the call. (NOTE: order reversed!)

Summary:

- A wide variety of design decisions are involved in the implementation of a function calling mechanism, with varying degrees of flexibility.
- Return addresses, function parameters, and local variables can be allocated storage on the stack.
- The most important data structure here is the activation record (a.k.a. stack frame).
- Function parameters and local variables can be accessed from the current activation record using either the stack pointer or a base pointer.

Implementing Objects

Implementing Objects:

- We have made considerable use of objects and classes in this course. How are these features implemented?
  - How are objects represented?
  - How do we access the instance variables of an object?
  - How do we call a member function?
  - How do derived classes work?
  - How does the program choose which version of a virtual function should be executed?

Representing Objects:

- Simple structures, for example:

```
class Date {
   int day;
   int month;
   int year;
}
```

  can be represented by a single block of storage, divided into fields:

day	month	year

- … Much like the implementation of arrays except that, at the language level, different types (and sizes) of element may appear in a single structure while all elements of an array have the same type.
- C.f. the struct declarations in C.

Allocating Objects:

- Objects in Java have so called reference semantics; i.e. objects are *identified with their addresses*.
- To create a new object, we must make sure we give it a new and *unique* address.
- For that reason, objects are allocated on the heap – a data structure commonly implemented as a linked list of free memory blocks.
- Heap allocation can be implemented directly as a simple run-time system routine, or we can reuse the function malloc() from the C standard library.
- (Heap storage reclamation will be deferred for now – see lecture 12 for more on this topic)

Allocating Objects:

- For example, the Java statement

```
Date today = new Date();
```

corresponds to the following code in C:

```
Date* today = (Date*)malloc(sizeof(Date));
```

[Note how the reference semantics is made explicit in C by the use of * in the types.]

Accessing Fields:

- If our target is C, we just have to note that obj.field in Java means obj->field in C.
- To generate machine code, however, we need detailed knowledge of the layout of a structure.
- Such information can be described by an environment, mapping field names to pairs containing the <u>type and offset</u> of each field:

  { day→(int,0), month→(int,4), year→(int,8) }

- This info is conveniently stored as part of the representation of the Date type.

Checking and Compiling Fields:

- So, to compile an expression e.m we need to check that
  - e evaluates to an object of some class type T.
  - T contains a field m of type S with offset o.
- If a is the address that e produces, then the value e.m of type S will be stored at address (a + o).
- Why do we need the type S of m? We might need to determine the <u>size</u> of the data at offset o!
- Note, though: In the MiniJava compiler, pointers as well as integers and booleans will all be stored using 32 bits (i.e., the size of fields never varies).

Member Functions:

- We can associate functions with classes:

```
class Date {
   int   day;
   int   month;
   int   year;
   void  set(int,int,int) …
   …
}

Date today;
…
today.set(28,2,2002);
```

Putting the definitions of the functions in the same place as the type definition makes the relationship more explicit. (Core idea behind object-orientation!)

Compiling Member Functions:

Member functions are treated like ordinary functions *except that*:

- The compiler assigns a *new name* to the function, known only to the compiler, to avoid clashes with other global or member functions of the same name. (Also known as name-mangling.)
- The compiler adds an *extra argument* to the function, which is a pointer to the object for which it was invoked. (Called this in Java and C++.)

For Example:

The following Java program …

```
class Date {
    int  day;
    int  month;
    int  year;
    void set(int d, int m, int y) {
        day = d; month = m; year = y;
    }
    ...
  }
  ...
  Date today;
  ...
  today.set(28,2,2002);
```

should be understood as …

Continued …

```
class Date {
    int  day;
    int  month;
    int  year;
};
...
void Date_set(Date this, int d, int m, int y) {
    this.day = d; this.month = m; this.year = y;
}
...
Date today;
...
Date_set(today,28,2,2002);
```

Note that references to a component x of a class in a member function are just abbreviations for this.x.

Derived classes (Subclasses):

◈ Consider a simple hierarchy using derived classes:

```
class Base {                    //         +-- Left
    int x,y;                    //         |
}                               //  Base -|
                                //         |
class Left extends Base {       //         +-- Right
    int l;
}

class Right extends Base {
    int r;
}
```

◈ How do we represent objects of classes Base, Left, and Right?

Nested Representations?

- Nest one structure inside another:

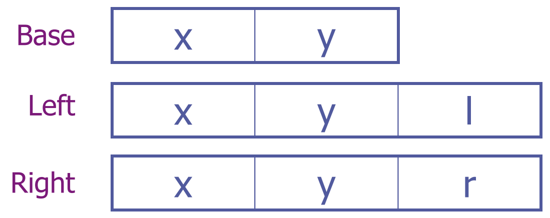
```
class Base {                    //          +-- Left
   int x,y;                     //          |
}                               //  Base -+
                                //          |
class Left  {                   //          +-- Right
   Base super;
   int l;
}

class Right {
   Base super;
   int r;
}
```

If l is an object of type Left, then references to l.x in the source would be translated to l.super.x.

Overlapping Layouts:

- Or perhaps we can be careful about object layout to make sure that common parts of a structure are always stored at the same offset…



- The x field is always at offset 0, y is always at 4, etc..
- Remember: we can always use a (pointer to a) Right or Left where a Base is required.

Virtual functions:

- Suppose that we extend the current example to include (virtual) functions:

```
class Base {
   int x,y;
   int  f(int i) {…}
   void g() {…}
}

class Left extends Base {
   int l;
   int f(int i) {…}
}
```

```
class Right extends Base {
   int r;
   void g(void) {…}
}
```

Suppose b is of type Base.

How do we implement a call like b.f(y)?

Calling virtual functions:

- If f was an "ordinary" member function, then the meaning of the call would be determined by the type of b.
  - For example, if b has type Base, we would call the f function defined in the Base class (Base_f, after name-mangling).
  - This can be determined at compile-time.
- If f is a virtual function, then there are other possibilities to consider:
  - If b was constructed using new Base(), or new Right(), then we want the Base implementation.
  - If b was constructed using new Left(), then we want the Left implementation.
- How do we choose?

Storing Tags in Objects:

- One approach is to store an extra “tag” field in each object:

Base	BASE	x	y	
Left	LEFT	x	y	l
Right	RIGHT	x	y	r

… and Using a Lookup Table:

- In addition, we store a table listing the addresses of each function that should be called, depending on the tag:

	BASE	LEFT	RIGHT
F	Base_f	Left_f	Base_f
G	Base_g	Base_g	Right_g

- Now we can implement a call like b.f(y) using:

(lookup[tag(b)][F])(b, y).

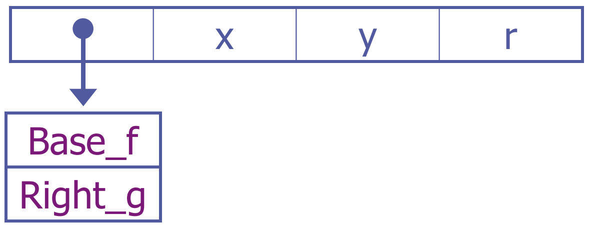
Using Arrays and Tags:

This method has some problems:

- If we are compiling just one file in a large program, how do we figure out the dimensions of the lookup table? How do we choose values for the indices?
- Assuming that we solve the first problem, there are likely to be a lot of unused entries in the table; a waste of space.

Instead: Virtual Function Tables!

- By contrast, we do know exactly how many virtual functions there are for any given class.



- Now we can implement a call like b.f(y) using:

((b.vptr[0]))(b, y).

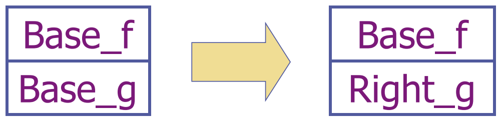
f at offset 0

Virtual Function Tables:

- The array of virtual function addresses is called a virtual function table (vtable).
- The same virtual function table can be shared by all instances of a given class; a suitable pointer will be loaded into the appropriate position in the object when storage for that object is first allocated.
- We only need to fix the order of the entries in the vtable at compile time.

Inheritance:

- The virtual function table for a derived class can be obtained from the table for the base class:
  - Copy entries that are not overridden in the derived class;
  - Replace entries corresponding to new implementations of base class virtual functions;
  - Append new entries for new virtual functions.

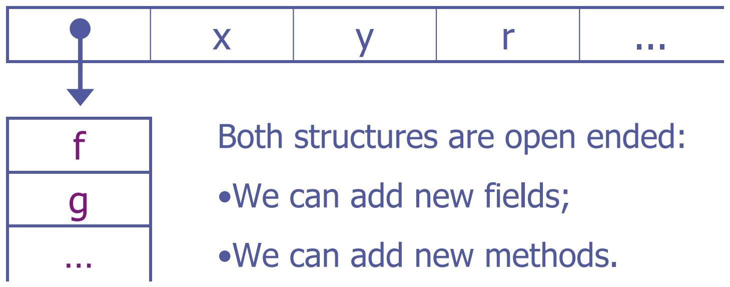


Vtable for Base

Vtable for Right

Extensibility:

- The beauty of this scheme is its extensibility. Any object that is derived from Right will have at least the following structure:



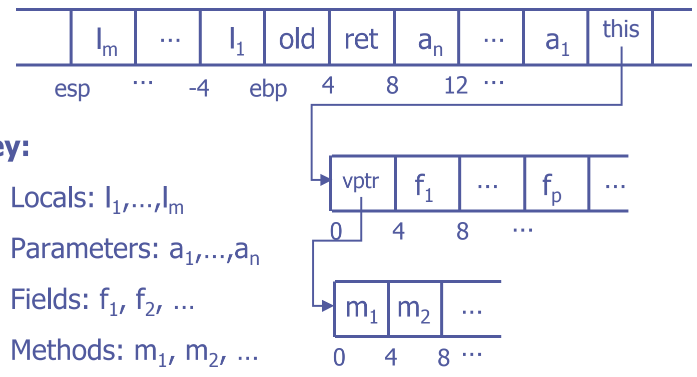
Both structures are open ended:

•We can add new fields;

•We can add new methods.

Method Stack Frames:

- Combining with the ideas from the last lecture, the stack frame for a typical method call looks something like this:



Key:

Locals: $l_1,\ldots,l_m$

Parameters: $a_1,\ldots,a_n$

Fields: f_1, f_2, …

Methods: m_1, m_2, …

Example:

- The code that we need to generate for a call obj.m_2(7) is as follows: (assume obj at offset 16 from ebp)

```
movl  16(%ebp),%eax     ; get obj
pushl %eax              ; pass as parameter (“this”)
movl  (%eax),%ebx       ; get vptr
movl  4(%ebx),%ebx      ; get address of m2
pushl $7                ; pass 7 as parameter
call  *%ebx             ; invoke function (indirect!)
addl  $8,%esp           ; pop parameters
                        ; (result in %eax)
```

Summary:

In this lecture we have seen:

- Objects can be implemented by contiguous blocks of storage, with fields referenced by their offset from the start of the object.
- Member functions can be implemented by functions passing an extra argument, this, as an implicit parameter.
- Virtual functions can be implemented using virtual function tables.
- Extensibility is built in to the data structures!