

# COOL Project Code Generation

CS2210

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Stack Machines

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Code Generation Models

- Evaluate all expression on stack
  - Stack machine
  - Conceptually very simple
    - Very slow
    - COOL provides support routines for this (ok for a toy compiler w/o optimization)
- Use processor registers to compute expressions
  - Used in practice
  - Much faster
  - Easier to optimize
  - Have to do (simple) register allocation

CS2210 Compiler Design 2003/04

---

---

---

---

---

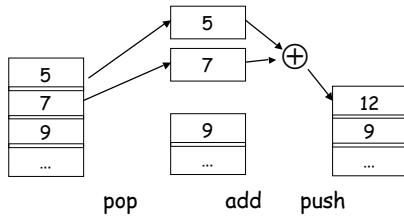
---

---

---

## Example of Stack Machine Operation

- The addition operation on a stack machine



CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Example of a Stack Machine Program

- Consider two instructions
  - **push i** - place the integer *i* on top of the stack
  - **add** - pop two elements, add them and put the result back on the stack
- A program to compute 7 + 5:

```
push 7  
push 5  
add
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Why Use a Stack Machine ?

- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme
- And therefore a simpler compiler

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Why Use a Stack Machine ?

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
  
- No need to specify the location of the result
  
- Instruction "add" as opposed to "add r<sub>1</sub>, r<sub>2</sub>"
  - ⇒ Smaller encoding of instructions
  - ⇒ More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
  
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
  
- The "add" instruction is now
  - acc ← acc + top\_of\_stack
  - Only one memory operation

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
  
- For an operation op(e<sub>1</sub>, ..., e<sub>n</sub>) push the accumulator on the stack after computing each of e<sub>1</sub>, ..., e<sub>n-1</sub>
  - After the operation pop n-1 values
  
- After computing an expression the stack is as before

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Stack Machine with Accumulator. Example

- Compute  $7 + 5$  using an accumulator

acc

7

5

+

12

stack

...

7

...

$acc \leftarrow 7$      $acc \leftarrow 5$      $acc \leftarrow acc + top\_of\_stack$   
 push acc                    pop

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## A Bigger Example: $3 + (7 + 5)$

Code	Acc	Stack
$acc \leftarrow 3$	3	<init>
push acc	3	3, <init>
$acc \leftarrow 7$	7	3, <init>
push acc	7	7, 3, <init>
$acc \leftarrow 5$	5	7, 3, <init>
$acc \leftarrow acc + top\_of\_stack$	12	7, 3, <init>
pop	12	3, <init>
$acc \leftarrow acc + top\_of\_stack$	15	3, <init>
pop	15	<init>

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Notes

- It is very important that the stack is preserved across the evaluation of a subexpression
  - Stack before the evaluation of  $7 + 5$  is 3, <init>
  - Stack after the evaluation of  $7 + 5$  is 3, <init>
  - The first operand is on top of the stack

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- We want to run the resulting code on the MIPS processor (or simulator)
- We simulate stack machine instructions using MIPS instructions and registers

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Simulating a Stack Machine...

- The accumulator is kept in MIPS register `$a0`
- The stack is kept in memory
- The stack grows towards lower addresses
  - Standard convention on the MIPS architecture
- The address of the next location on the stack is kept in MIPS register `$sp`
  - The top of the stack is at address `$sp + 4`

---

---

---


---

---

---

---

---



## MIPS Assembly

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
- 32 general purpose registers (32 bits each)
  - We will use `$sp`, `$a0` and `$t1` (a temporary register)

CS2210 Compiler Design 2003/04

■ Read the SPIM handout for more details

---

---

---

---

---

---

---

---

## A Sample of MIPS Instructions

- `lw reg1 offset(reg2)`
  - Load 32-bit word from address `reg2 + offset` into `reg1`
- `add reg1 reg2 reg3`
  - `reg1 ← reg2 + reg3`
- `sw reg1 offset(reg2)`
  - Store 32-bit word in `reg1` at address `reg2 + offset`
- `addiu reg1 reg2 imm`
  - `reg1 ← reg2 + imm`
  - "u" means overflow is not checked
- `li reg imm`
  - `reg ← imm`

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## MIPS Assembly. Example.

■ The stack-machine code for `7 + 5` in MIPS:

```

acc ← 7          li $a0 7
push acc        sw $a0 0($sp)
                addiu $sp $sp -4
acc ← 5          li $a0 5
acc ← acc + top_of_stack  lw $t1 4($sp)
                add $a0 $a0 $t1
                addiu $sp $sp 4
    
```

• We now generalize this to a simple language...

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## A Small Language

- A language with integers and integer operations

`P → D; P | D`

`D → def id(ARGS) = E;`

`ARGS → id, ARGS | id`

`E → int | id | if E1 = E2 then E3 else E4`

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## A Small Language (Cont.)

- The first function definition `f` is the "main" routine
- Running the program on input `i` means computing `f(i)`
- Program for computing the Fibonacci numbers:
  - `def fib(x) = if x = 1 then 0 else`  
                   `if x = 2 then 1 else`  
                   `fib(x - 1) + fib(x - 2)`

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Code Generation Strategy

- For each expression `e` we generate MIPS code that:
  - Computes the value of `e` in `$a0`
  - Preserves `$sp` and the contents of the stack
- We define a code generation function `cgen(e)` whose result is the code generated for `e`

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:

`cgen(i) = li $a0 i`

- Note that this also preserves the stack, as required

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Code Generation for Add

```

cgen(e1 + e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  
```

- Possible optimization: Put the result of  $e_1$  directly in register  $\$t1$  ?

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Code Generation for Add. Wrong!

- Optimization: Put the result of  $e_1$  directly in  $\$t1$ ?

```

cgen(e1 + e2) =
  cgen(e1)
  move $t1 $a0
  cgen(e2)
  add $a0 $t1 $a0
  
```

- Try to generate code for :  $3 + (7 + 5)$

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Code Generation Notes

- The code for  $+$  is a template with "holes" for code for evaluating  $e_1$  and  $e_2$
- Stack machine code generation is recursive
- Code for  $e_1 + e_2$  consists of code for  $e_1$  and  $e_2$  glued together
- Code generation can be written as a recursive-descent of the AST
  - At least for expressions

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Code Generation for Sub and Constants

- New instruction: `sub reg1 reg2 reg3`
  - Implements  $reg_1 \leftarrow reg_2 - reg_3$ 

```
cgen(e1 - e2) =  
  cgen(e1)  
  sw $a0 0($sp)  
  addiu $sp $sp -4  
  cgen(e2)  
  lw $t1 4($sp)  
  sub $a0 $t1 $a0  
  addiu $sp $sp 4
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for Conditional

- We need flow control instructions
- New instruction: `beq reg1 reg2 label`
  - Branch to label if  $reg_1 = reg_2$
- New instruction: `b label`
  - Unconditional jump to label

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for If (Cont.)

```
cgen(if e1 = e2 then e3 else e4)  
  =  
  cgen(e1)  
  sw $a0 0($sp)  
  addiu $sp $sp -4  
  cgen(e2)  
  lw $t1 4($sp)  
  addiu $sp $sp 4  
  beq $a0 $t1 true_branch  
  false_branch:  
  cgen(e4)  
  b end_if  
  true_branch:  
  cgen(e3)  
  end_if:
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For  $f(x_1, \dots, x_n)$  push  $x_n, \dots, x_1$  on the stack
    - These are the only variables in this language

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## The Activation Record (Cont.)

- The stack discipline guarantees that on function exit  $\$sp$  is the same as it was on function entry
- We need the return address
- It's handy to have a pointer to the current activation
  - This pointer lives in register  $\$fp$  (frame pointer)
  - Reason for frame pointer will be clear shortly

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## The Activation Record

- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices
- Picture: Consider a call to  $f(x,y)$ , The AR will be:
 

FP		}	AR of f
	old fp		
	y		
	x		
SP			

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: `jal label`
  - Jump to label, save address of next instruction in `$ra`
  - On other architectures the return address is stored on the stack by the "call" instruction

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for Function Call (Cont.)

```

cgen(f(e1, ..., en)) =
  sw $fp 0($sp)
  addiu $sp $sp -4
  cgen(en)
  sw $a0 0($sp)
  addiu $sp $sp -4
  ...
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal f_entry
  
```

CS2210 Compiler Design 2003/04

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register `$ra`
- The AR so far is  $4*n+4$  bytes long

---

---

---

---

---

---

---

---

## Code Generation for Function Definition

```

cgen(def f(x1, ..., xn) = e) =
  move $fp $sp
  sw $ra 0($sp)
  addiu $sp $sp -4
  cgen(e)
  lw $ra 4($sp)
  addiu $sp $sp z
  lw $fp 0($sp)
  jr $ra
  
```

CS2210 Compiler Design 2003/04

- Jump to address in register `$ra`
- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- $z = 4*n + 8$

---

---

---

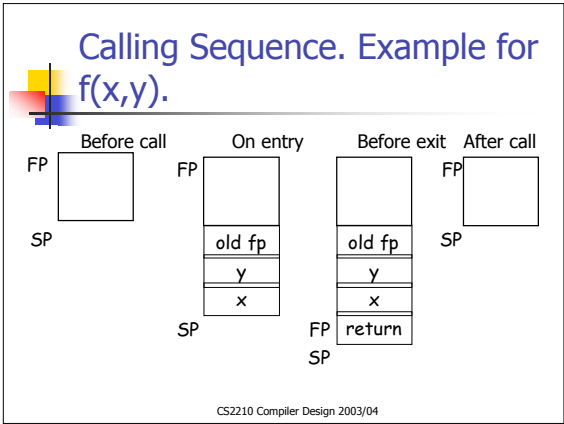
---

---

---

---

---




---

---

---

---

---

---

---

---

- ### Code Generation for Variables
- Variable references are the last construct
  - The "variables" of a function are just its parameters
    - They are all in the AR
    - Pushed by the caller
  - Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from `$sp`
- CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

- ### Code Generation for Variables (Cont.)
- Solution: use a frame pointer
    - Always points to the return address on the stack
    - Since it does not move it can be used to find the variables
  - Let  $x_i$  be the  $i^{\text{th}}$  ( $i = 1, \dots, n$ ) formal parameter of the function for which code is being generated
- $$\text{cgen}(x_i) = \text{lw } \$a0 \text{ } z(\$fp) \quad (z = 4*i)$$
- CS2210 Compiler Design 2003/04

---

---

---

---

---

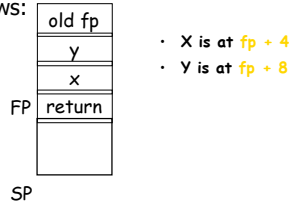
---

---

---

## Code Generation for Variables (Cont.)

- Example: For a function `def f(x,y) = e` the activation and frame pointer are set up as follows:



CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Recommend to not use a stack machine
  - You learn more
  - Alternative not much more complicated

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## A Better Way

- Idea: Keep temporaries in the AR
- The code generator must assign a location in the AR for each temporary

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Example

```
def fib(x) = if x = 1 then 0 else
            if x = 2 then 1 else
            fib(x - 1) + fib(x - 2)
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## How Many Temporaries?

■ Let  $NT(e)$  = # of temps needed to evaluate  $e$

- $NT(e_1 + e_2)$ 
  - Needs at least as many temporaries as  $NT(e_1)$
  - Needs at least as many temporaries as  $NT(e_2) + 1$
- Space used for temporaries in  $e_1$  can be reused for temporaries in  $e_2$

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## The Equations

```
NT(e1 + e2) = max(NT(e1), 1 + NT(e2))
NT(e1 - e2) = max(NT(e1), 1 + NT(e2))
NT(if e1 = e2 then e3 else e4) = max(NT(e1), 1 + NT(e2), NT(e3), NT(e4))
NT(id(e1, ..., en)) = max(NT(e1), ..., NT(en))
NT(int) = 0
NT(id) = 0
```

Is this bottom-up or top-down?  
 What is  $NT(\dots\text{code for fib}\dots)$ ?

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## The Revised AR

- For a function definition  $f(x_1, \dots, x_n) = e$  the AR has  $2 + n + NT(e)$  elements
  - Return address
  - Frame pointer
  - $n$  arguments
  - $NT(e)$  locations for intermediate results

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Picture

Old FP
$x_n$
...
$x_1$
Return Addr.
Temp $NT(e)$
...
Temp 1

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---



## Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for + (original)

```
cgen(e1 + e2) =  
  cgen(e1)  
  sw $a0 0($sp)  
  addiu $sp $sp -4  
  cgen(e2)  
  lw $t1 4($sp)  
  add $a0 $t1 $a0  
  addiu $sp $sp 4
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Code Generation for + (revised)

```
cgen(e1 + e2, nt) =  
  cgen(e1, nt)  
  sw $a0 nt($fp)  
  cgen(e2, nt + 4)  
  lw $t1 nt($fp)  
  add $a0 $t1 $a0
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Notes

- The temporary area is used like a small, fixed-size stack
- Can construct `cgen` for other constructs

CS2210 Compiler Design 2003/04

---

---

---

---


---

---

---

---





## Implementation Alternative

- Do expression evaluation in registers
  - much faster
  - easier to optimize
  - have to write your own code generation support routines :-(  
    - But not too difficult and you may find it easier in some respects

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Suggested Code Generation Algorithm (cf. Aho ch. 9)

- Walk tree and generate CFG with basic blocks of 3-address code
  - Use new temporary name for every sub-expression t1, t2, ... don't worry about actual registers and reusing temporary locations
  - Extension: transform this to SSA form
    - Have to compute dominators and iterated DF
    - Do this only once you have the first part done
- One optimization you can do here:
  - Do not use getreg/putreg allocation but perform register allocation on this CFG and generate code from it
    - Good speedups possible
- Generate code for each basic block
  - 3-address code statement by statement

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Register Allocation

- Write a support routine that gives you (virtual) registers to do computation
  - Getreg / putreg (can be found in Aho ch. 9.6)
  - May return a real register or a stack memory location
    - Since on RISC all operations have to be performed in registers:
      - Reserve 2 registers for evaluation
      - Bring in-memory (stack) operands first into this expression register(s)
      - Evaluate and store back
- Alternative: perform register allocation on the CFG
  - This counts as an optimization

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Expression Evaluation

- Use an address descriptor
  - a map of names of variables & temporaries to register locations
- $x := y \text{ op } z$ 
  - $L = \text{getreg}()$  for the result of the computation
  - $y'$  = location of  $y$  if not in a register call  $\text{getreg}()$  and generate a `ld`-instruction
    - same for  $z$
  - generate  $L := y' \text{ op } z'$
  - update  $x$ 's address map to indicate that  $x$  is in  $L$  now

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Object Layout

- OO implementation = Stuff from last lecture + More stuff
- OO Slogan: If  $B$  is a subclass of  $A$ , then an object of class  $B$  can be used wherever an object of class  $A$  is expected
- This means that code in class  $A$  works unmodified for an object of class  $B$

CS2210 Compiler Design 2003/04

---

---

---


---

---

---

---

---



## Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Object Layout Example

```

Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d };
};

Class B inherits A {
  b: Int <- 2;
  f(): Int { a };
  g(): Int { a <- a - b };
};

Class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a * c };
};

```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Object Layout (Cont.)

- Attributes **a** and **d** are inherited by classes **B** and **C**
- All methods in all classes refer to **a**
- For **A** methods to work correctly in **A**, **B**, and **C** objects, attribute **a** must be in the same "place" in each object

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Object Layout (Cont.)

An object is like a **struct** in C. The reference **foo.field** is an index into a **foo** struct at an offset corresponding to **field**

Objects in Cool are implemented similarly

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is **self** and the fields are the object's attributes

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Cool Object Layout

- The first 3 words of Cool objects contain header information.

Class Tag	0
Object Size	4
Dispatch Ptr	8
Attribute 1	12
Attribute 2	16
...	

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Cool Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots
- Lay out in contiguous memory

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Subclasses

Observation: Given a layout for class **A**, a layout for subclass **B** can be defined by extending the layout of **A** with additional slots for the additional attributes of **B**

Leaves the layout of **A** unchanged  
(**B** is an extension)

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Layout Picture

Offset	0	4	8	12	16	20
Class						
A	Ata g	5	*	a	d	
B	Bta g	6	*	a	d	b
C	Cta g	6	*	a	d	c

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses
  - Any method for an  $A_1$  can be used on a subclass  $A_2$
- Consider layout for  $A_n < \dots < A_3 < A_2 < A_1$

Header
$A_1$ attrs.
$A_2$ attrs
$A_3$ attrs
...

*$A_1$  object*

*$A_2$  object*

*$A_3$  object*

*What about multiple inheritance?*

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Dynamic Dispatch

- Consider the following dispatches (using the same example)

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Object Layout Example (Repeat)

```
Class A {  
  a: Int <- 0;  
  d: Int <- 1;  
  f(): Int { a <- a + d };  
};  
  
Class B inherits A {  
  b: Int <- 2;  
  f(): Int { a };  
  g(): Int { a <- a - b };  
};  
  
Class C inherits A {  
  c: Int <- 3;  
  h(): Int { a <- a * c };  
};
```

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Dynamic Dispatch Example

- `e.g()`
  - `g` refers to method in `B` if `e` is a `B`
- `e.f()`
  - `f` refers to method in `A` if `f` is an `A` or `C` (inherited in the case of `C`)
  - `f` refers to method in `B` for a `B` object
- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
- A *dispatch table* indexes these methods
  - An array of method entry points
  - A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Dispatch Table Example

Offset	0	4
Class		
A	fA	
B	fB	g
C	fA	h

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset  $O_f$  in the dispatch table at compile time

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---

## Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch  $e.f()$  we
  - Evaluate e, giving an object x
  - Call  $D[O_f]$ 
    - D is the dispatch table for x
    - In the call, self is bound to x

CS2210 Compiler Design 2003/04

---

---

---

---

---

---

---

---