Compilation Principle
编 译 原 理

第22讲：目标代码生成(2)

张献伟

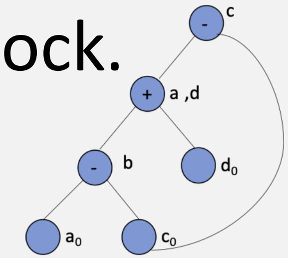
xianweiz.github.io

DCS290, 6/9/2022

Quiz Questions

QUIZ

- Q1: DAG of the basic block.




```
b = a – c
a = b + d
c = a – c
d = b + d
```

- Q2: optimize the code.

```
x = 5
a = 2 * x
c = b * 4
d = a + b * 4
for(i=0; i<10; i++) f(c*d + i)
```

```
c = b << 2
d = 10 + c
e = c *d
for(i=0; i<10; i++) f(e + i)
```

- Q3: list different levels of code optimization.

  Peephole, Local, Loop, Global.

- Q4: main tasks of target code generation?

  Instruction selection, register allocation, instruction ordering.

- Q5: what are $sp and $fp registers for?

  $sp: stack pointer

  $fp: frame pointer

Final Exam

- 考试时间:
  - 6.28/周二，14:30 – 16:30
- 关于试卷
  - 中文（专业术语标注英文）
  - A、B卷，学院指定
- 成绩计算
  - 期末：60%
  - 平时：40%
    - 课堂：10%
    - 作业：30%

- 题型及分值
  - 一、判断题（10分）
    - 10小题，每小题1分
  - 二、填空题（10分）
    - 8小题，10个空白，每空白1分
  - 三、简答题（15分）
    - 3小题，每小题5分
  - 四、应用题（40分）
    - 3小题，10分+15分+15分
  - 五、综合应用题（25分）
    - 1小题，每小题25分

Code Generation Strategy

- For each expression *e* we generate MIPS code that:
  - Computes the value of *e* into *$t0*
  - Preserves *$sp* and the contents of the stack
- We define a code generation function *cgen(e)*
  - Its result is the code generated for *e*

- Code generation for constants
  - The code to evaluate a constant simply copies it into the register: *cgen(i) = li $t0 i*
    - Note that this also preserves the stack, as required

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

Code Generation for ALU

- Default

```
cgen(e1 + e2):
        # stores result in $t0
        cgen(e1)
        # pushes $t0 on stack
        addiu $sp $sp -4
        sw $t0 0($sp)
        # overwrites result in $t0
        cgen(e2)
        # pops value of e1 to $t1
        lw $t1 4($sp)
        addiu $sp $sp 4
        # performs addition
        add $t0 $t1 $t0
```

→

```
cgen(e1 + e2):
        # stores result in $t0
        cgen(e1)
        # copy result of $t0 to $t1
        move $t1 $t0
        # stores result in $t0
        cgen(e2)
        # performs addition
        add $t0 $t1 $t0
```

- Possible optimization: put the result of *e1* directly in register *$t1*? What if 3 + (7 + 5)?

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

Code Generation for Conditional

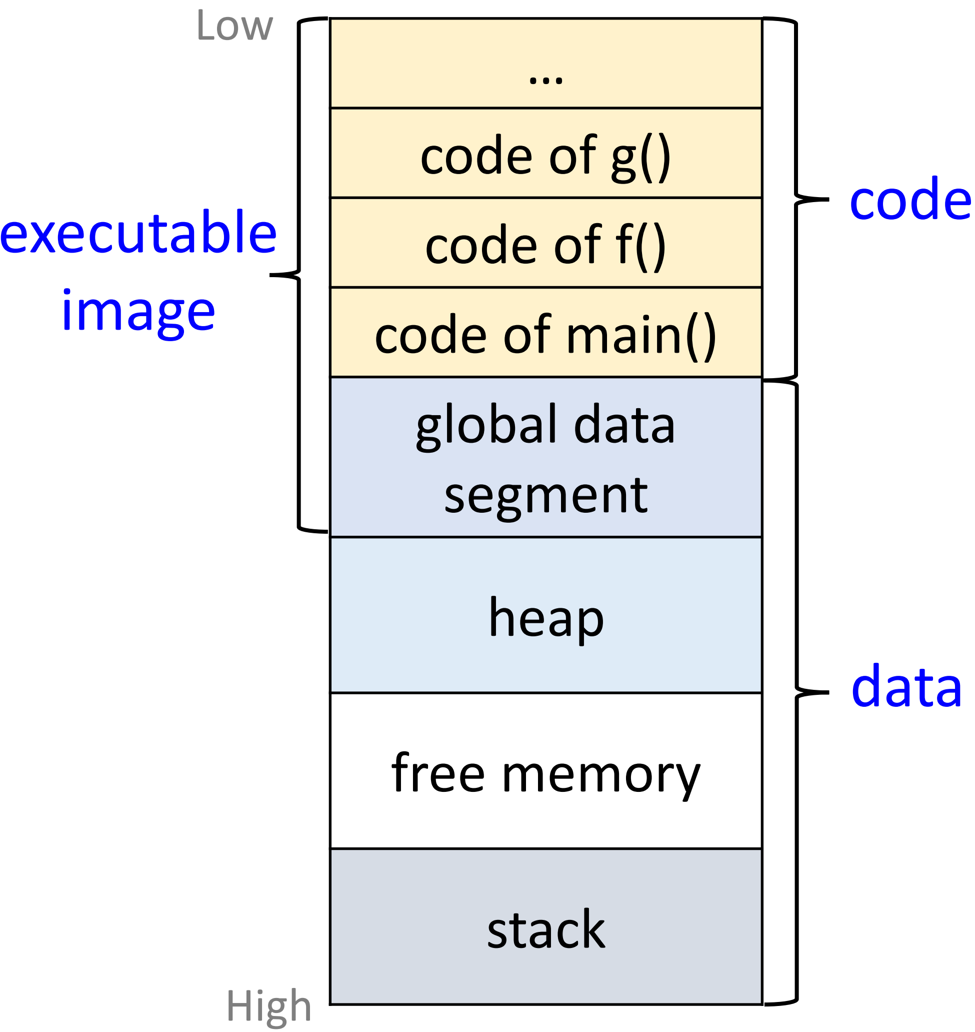
- We need flow control instructions
- New instruction: *beq reg1 reg2 label*
  - Branch to label if *reg1 == reg2*
- New instruction: *b label*
  - Unconditional jump to *label*

```
cgen(if e1 == e2 then e3 else e4):
        cgen(e1)
        # pushes $t0 on stack
        addiu $sp $sp -4
        sw $t0 0($sp)
        # overwrites $t0
        cgen(e2)
        # pops value of e1 to $t1
        lw $t1 4($sp)
        addiu $sp $sp 4
        # performs comparison
        beq $t0 $t1 true_branch
    false_branch:
        cgen(e4)
        b end_if
    true_branch:
        cgen(e3)
    end_if:
```

中山大學 SUN YAT-SEN UNIVERSITY

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

Example Memory Layout



- Code
  - the size of the generated target code is fixed at compile time
- Global/**static**
  - the size of some program data objects, e.g., global constants, are known at compile time
- **Stack**
  - store dynamic data structures
- **Heap**
  - manage long-lived data

Activation[活动]

- Compiler typically allocates memory in the unit of procedure[以过程调用为单位]
- Each execution of a procedure is called as its **activation**[活动]
  - An execution of a procedure starts at the beginning of the procedure body
  - When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called
- **Activation record** (AR)[活动记录] is used to manage the information needed by a single execution of a procedure
- **Stack** is to hold activation records that get generated during procedure calls

ARs in Stack Memory[在栈中管理]

- Manage ARs like a stack in memory[AR栈管理]
  - On function entry: AR instance allocated at top of stack
  - On function return: AR instance removed from top of stack
- Hardware support[硬件支持]
  - Stack pointer ($SP) register[栈指针]
    - $SP stores address of top of the stack
    - Allocation/de-allocation can be done by moving $SP
  - Frame pointer ($FP) register[帧指针]
    - $FP stores base address of current frame
    - **Frame**: another word for activation record (AR)
    - Variable addresses translated to an offset from $FP
  - $FP and $SP together delineate the bounds of current AR

Contents of ARs

- Example layout of a function AR

Temporaries	临时变量
Local variables	局部变量
Saved Caller/Callee Register Values	保存的寄存器值
Saved Caller's Instruction Pointer ($IP)	保存的调用者指令指针
Saved Caller's AR Frame Pointer ($FP)	保存的调用者帧指针
Parameters	参数
Return Value	返回值

- Registers such as $FP and $IP overwritten by callee → Must be saved to/restored from AR on call/return
  - Caller's $IP: where to execute next on function return (a.k.a. return address: instruction following function call)
  - Caller's $FP: where $FP should point to on function return
  - Saved Caller/Callee Registers: other registers (will discuss)

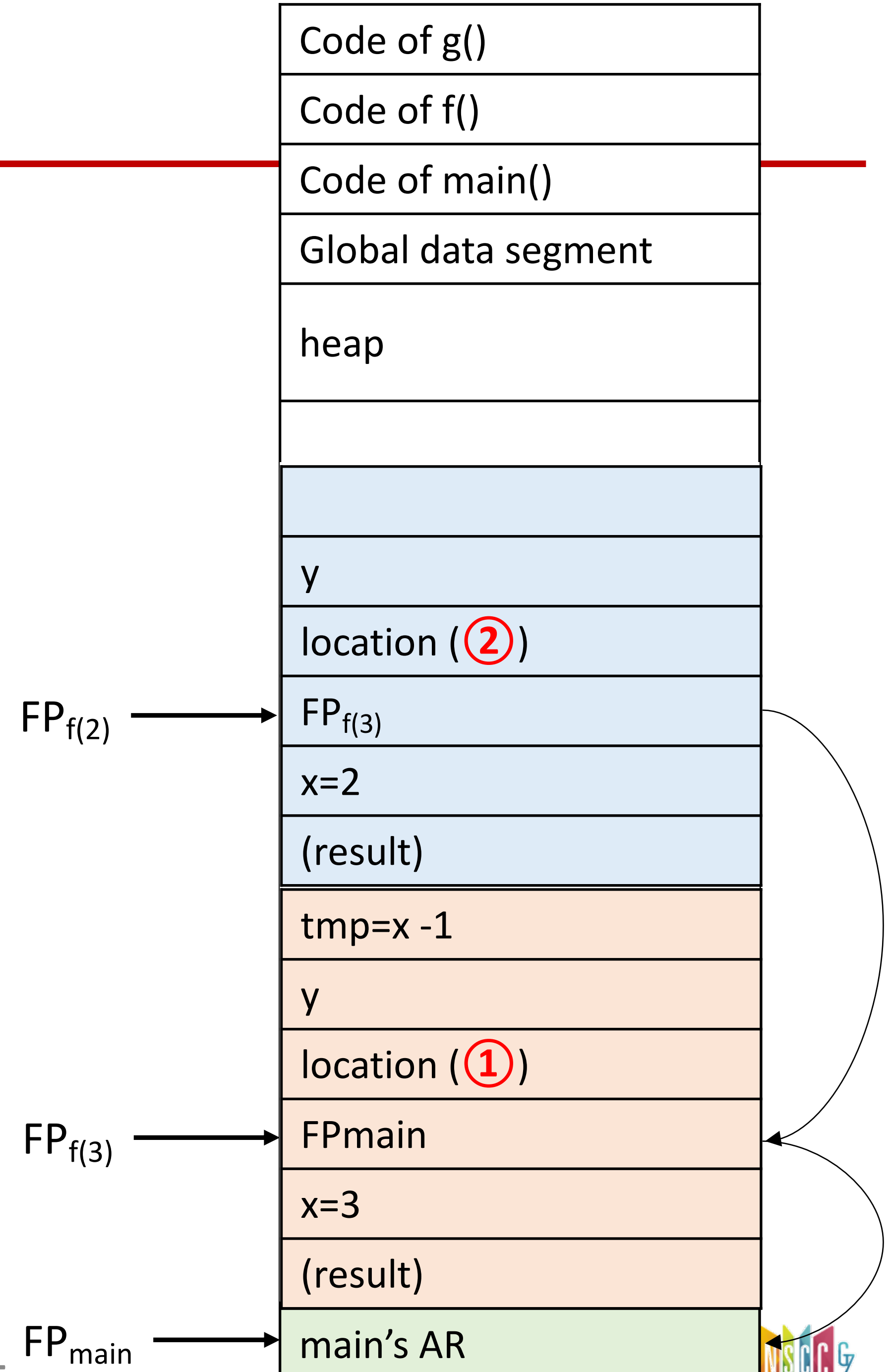
Example

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Instruction Pointer ($IP)
Saved Caller's AR Frame Pointer ($FP)
Parameters
Return Value

```
int g() {
  return 1;
}

int f(int x) {
  int y;
  if (x==2)
    y = 1;
  else
    y = x + f(x-1);
   ② ...
   return y;
}

int main() {
  f(3);
  ① ...
}
```



https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Caller/Callee Conventions[规范]

- Important registers should be saved across function calls
  - Otherwise, values might be overwritten
- But, who should take the responsibility?
  - The caller knows which registers are important to it and should be saved
  - The callee knows exactly which registers it will use and potentially overwrite
  - However, in the typical “block box” programming, caller and callee don’t know anything about each other’s implementation
- Potential solutions
  - **Sol1**: caller to save any important registers that it needs before calling a func, and to restore them after (but not all will be overwritten)
  - **Sol2**: callee saves and restores any registers it might overwrite (but not all are important to caller)

Caller/Callee Conventions (cont.)

- Caller and callee should cooperate
- <u>Caller</u>: save and restore any of the following caller-saved registers that it cares about

  $t0-$t9 $a0-$a3 $v0-$v1

  - The callee may freely modify these registers, under the assumption that the caller already saved them
- <u>Callee</u>: save and restore any of the following callee-saved registers that it uses

  $s0-$s7 $ra

  - The caller may assume these registers are not changed by the callee

Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 ⋯ 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 ⋯ 9.
s0 - s7	16 - 23	Saved Registers 0 ⋯ 7.
k0 - k1	26 - 27	Kernel Registers 0 ⋯ 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

Detailed Calling Steps

- The **caller** sets up for the call via these steps[调用者]
  - 1) Make space on stack for and save any caller-saved registers
  - 2) Pass arguments by pushing them on the stack, one by one, right to left
  - 3) Execute a jump to the function (saves the next inst in $ra)

- The **callee** takes over and does the following[被调用者]
  - 4) Make space on stack for and save values of $fp and $ra
  - 5) Configure frame pointer by setting $fp to base of frame
  - 6) Allocate space for stack frame (total space required for all local and temporary variables)
  - 7) Execute function body, code can access params at positive offset from $fp, locals/temps at negative offsets from $fp

Detailed Calling Steps (cont.)

- When ready to exit, the **callee** does following[调用退出]
  - 8) Assign the return value (if any) to $v0
  - 9) Pop stack frame off the stack (locals/temps/saved regs)
  - 10) Restore the value of $fp and $ra
  - 11) Jump to the address saved in $ra

- When control returns to the **caller**, it cleans up from the call with the steps[调用返回]
  - 12) Pop the parameters from the stack
  - 13) Restore value of any caller-saved registers, pops spill space from stack

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, after saving address of next instruction in $ra

```
cgen(f(e1, …, en)):
        # pushes arguments (reverse order)
        cgen(en)
        addiu $sp $sp -4
        sw $a0 0($sp)
        …
        cgen(e1)
        addiu $sp $sp -4
        sw $a0 0($sp)
        # saves FP
        addiu $sp $sp -4
        sw $fp 0($sp)
        # pushes return address
        addiu $sp, $sp, -4
        sw $ra, 0($sp)
        # begins new AR in stack
        move $fp, $sp
        # jumps to func entry (update $ra)
        jal f_entry
```

Code Generation for Function Definition

- New instruction: *jr reg*
  - Jump to address in register reg

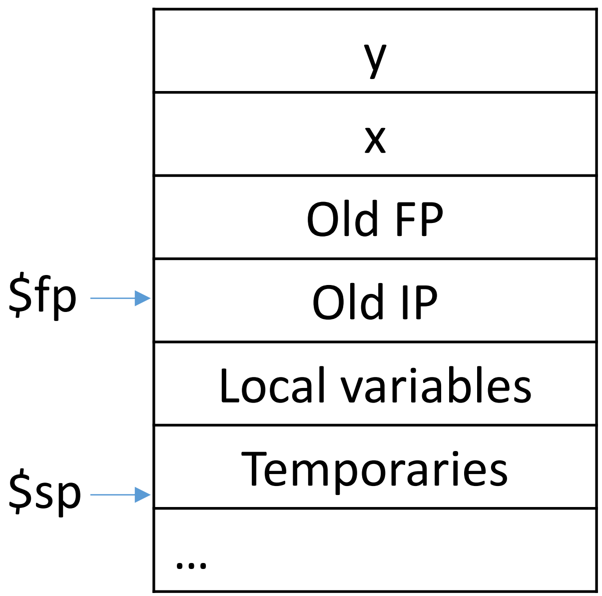
```
cgen(def f(x1,…,xn) = e):
f_entry: cgen(e)
          # removes AR from stack
          move $sp $fp
          # pops return address
          sw $ra 0($sp)
          addiu $sp $sp 4
          # pops old FP
          lw $fp 0($sp)
          addiu $sp $sp 4
          # jumps to return address
          jr $ra
```

Code Generation for Variables

- 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
  - Thus, access to locations in the stack frame cannot use $sp-relative addressing
- **Solution**: use the frame pointer $fp instead
  - Always points to the return address on the stack
  - Since it does not move, it can be used to find the variables

Example

- Local variables are referenced from an offset from $fp
  - $fp is pointing to old $ip (return address)
- For a function *def f(x,y) = e* the activation and frame pointer are set up as follows:



x: +8($fp)

y: +12($fp)

First local variable: -4($fp)

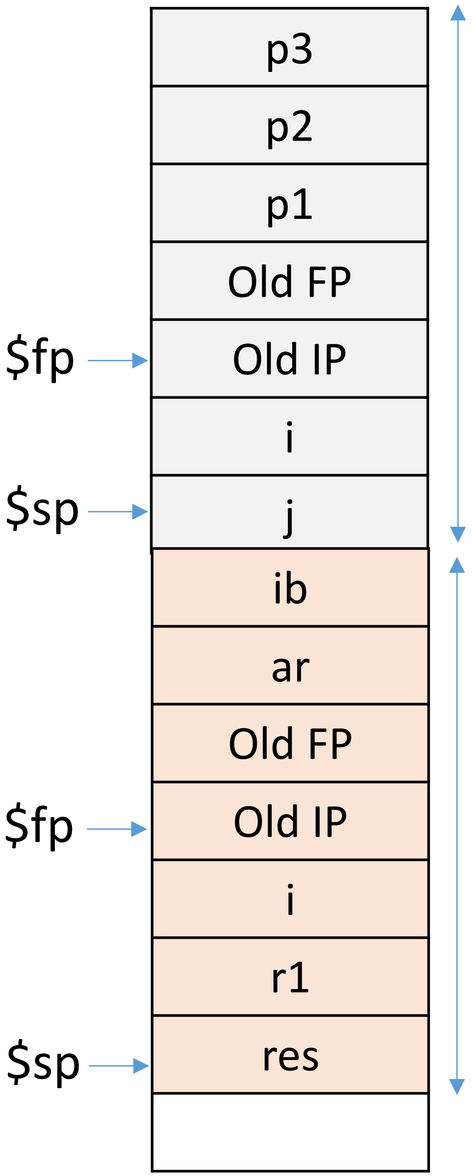
The parameters are pushed right to left by the caller

The locals are pushed left to right by the callee

Example

```
double fun1(int p1, double p2, int p3) {
    int i, j;
    res = fun2(p1*p2, j);
    return res;
}
```

```
double fun2(double ar, int ib) {
    int i, r1;
    double res;
    …
    return res;
}
```



Code Generation for OO

- Objects are like structs in C
  - Objects are laid out in contiguous memory
  - Each member variable is stored at a fixed offset in object
- Unlike structs, objects have member methods
- Two types of member methods:
  - **Nonvirtual** member methods: cannot be overridden

    ```
    Parent obj = new Child();
    obj.nonvirtual(); // Parent::nonvirtual() called
    ```

    Method called depends on (static) reference type

    Compiler can decide call targets statically
  - **Virtual** member methods: can be overridden by child class

    ```
    Parent obj = new Child();
    obj.virtual(); // Child::virtual() called
    ```

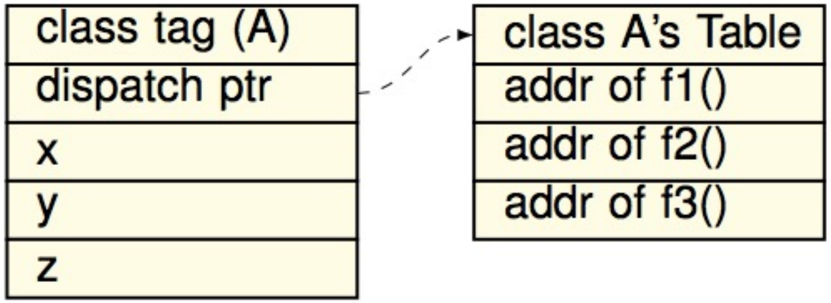
    Method called depends on (runtime) type of object

    Need to call different targets depending on runtime type

Static and Dynamic Dispatch

- **Dispatch**: to send to a particular place for a purpose
  - I.e., to jump to a (particular) function
- **Static Dispatch**: selects call target at compile time
  - Nonvirtual methods implemented using static dispatch
  - Implication for code generation:
    - Can hard code function address into binary
- **Dynamic Dispatch**: selects call target at runtime
  - Virtual methods implemented using dynamic dispatch
  - Implication for code generation:
    - Must generate code to select correct call target
- How?
  - At compile time, generate a **dispatch table** for each class, containing call targets for all virtual methods of that class
  - At runtime, each object has a pointer to its dispatch table, which is indexed into to find call target for its runtime type

Typical Object Layout



- Class tag is used for dynamic type checking
- Dispatch ptr is a pointer to the dispatch table
- Compiler translates member accesses to offset accesses

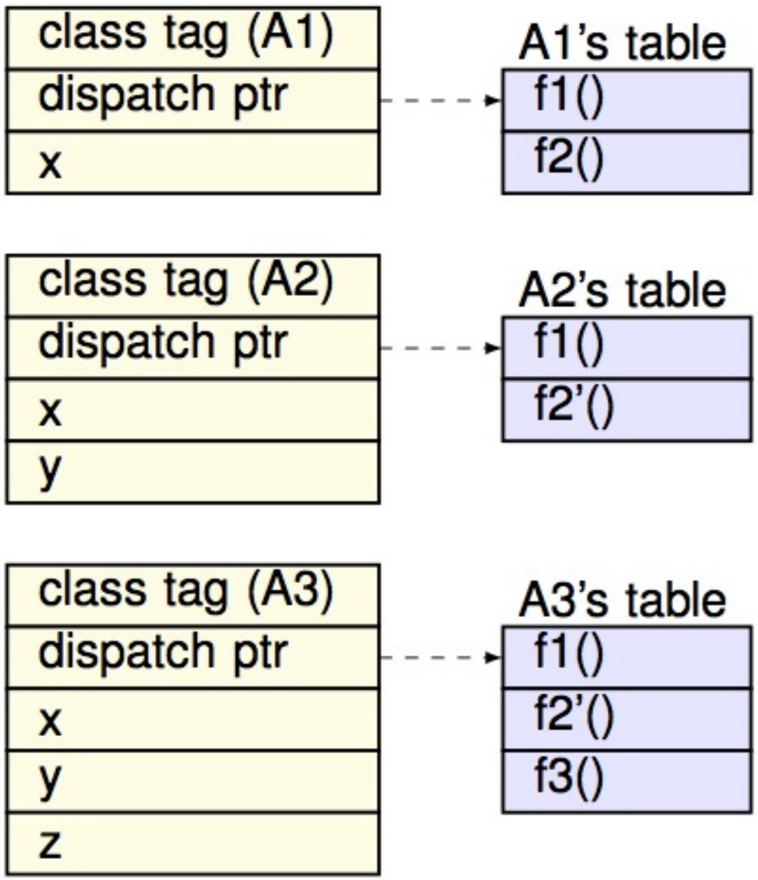
```
if(...) obj = new Parent()
else  obj = new Child();
obj.x = 10;          // move 10, x_offset(obj)
obj.f2();            // call f2_offset(obj.dispatch_ptr)
```

- Offsets must remain identical regardless of object type
  - How to layout object and dispatch table to make it so?

Inheritance and Subclasses

- Invariant: the offset of a member variable or member method is the same in a class and all of its subclasses

```
class A1 {
   int x;
   virtual void f1() { ... }
   virtual void f2() { ... }
}
class A2 inherits A1 {
   int y;
   virtual void f2() { ... }
}
class A3 inherits A2 {
   int z;
   virtual void f3() { ... }
}
```



A Question …

```cpp
#include <iostream>
using namespace std;

class A1 {
  public:
    virtual void f1() { cout << "base.f1\n"; }
    virtual void f2() { cout << "base.f2\n"; }
    void f3() { cout << "base.f3\n"; }
  private:
    char a;
    int x;
    int y;
    static int z;
};

int main(int argc, char* argv[]) {
    A1 a1;
    cout << "sizeof(a1) = " << sizeof(a1) << "\n";

    return 0;
}
```

- What is the output?
  - **24** (on my 64-bit MBA)
- How come?
  - Fields (12B)
    - char a: 1 --> 4
    - int x: 4
    - int y: 4
  - Functions (8B)
    - virtual: 8B
  - Alignment
    - 12+8 --> 24

[1] Determining the Size of a Class Object
[2] sizeof class in C++