REVE1

Translation of C to Assembly

```
absdiff:
    cmpq   %rsi, %rdi
    jg     .L4
    movq   %rsi, %rax
    subq   %rdi, %rax
    jmp    .L7
.L4:
    movq   %rdi, %rax
    subq   %rsi, %rax
.L7:
    ret
```

Module Outline

- **Assembly Programming**
  - Program Structure
  - Language Elements
- **Altering Control Flow**
  - If-then-else Constructs
  - Loop Constructs
  - Switch Statements
- **Composite Data Structures**
  - Arrays
  - Structures
  - Unions
- **Calling Convention**
- **Module Summary**

```
    .text
    .align 16

    .globl main

main:
    movl  $1, %eax
    movl  $1, %edi
    leaq  hstr(%rip), %rsi
    movl  $14, %edx

    syscall

    xor   %eax, %eax
    ret

    .data
    .align 16

hstr:
    .ascii "Hello, world!\n"
```

Assembly Programming

Program Structure

HOCHSCHULE LUZERN

Hello, world!

- **Our first assembly program**

```
    .text
    .align 16

    .globl main

main:
    movl  $1, %eax
    movl  $1, %edi
    leaq  hstr(%rip), %rsi
    movl  $14, %edx

    syscall

    xor   %eax, %eax
    ret

    .data
    .align 16

hstr:
    .ascii "Hello, world!\n"
```

Hello, world!

- **Our first assembly program**

```
$ gcc -o hello hello.S
$ ./hello
Hello, world!
$
```

General Structure

```
    .text
    .align 16

    .globl main

main:
    movl  $1, %eax
    …
    ret


    .data
    .align 16

hstr:
    .ascii "Hello, world!\n"
```

section specifier (.text = code)

alignment specification

exported symbols

function definitions

section specifier (.data = global variables)

alignment specification

label (memory address alias, i.e., variable)

data (value of initialized variable)

Assembly Programming

Language Elements

The GNU Assembler (GAS)

- **The GNU Assembler**
  - gas, as
  - assembler maintained by the GNU project
  - supports many, many different architectures
- **Syntax varies by architecture**
  - to match SOP of those architectures
  - last line must be terminated
    by a newline
- **Many, many options and directives**
  - refer to the excellent GAS manual

HOCHSCHULE LUZERN

Language Elements

- **Block delimiter**
  - no blocks in assembly
- **Whitespace**
  - spaces, tabs, newlines
  - is ignored
  - no forced indentation
- **Command end marker**
  - newline
  - last line terminated by a newline
- **Comments**
  - single line: #
  - multi-line: /* … */
  - multi-line > single line
  - nesting of same-level comments not supported

```
    .text
    .align 16

    .globl main

/*
 * main function
 */
main:
    movl  $1, %eax   # set %rax to 1
    …
    ret              # return to caller


    .data
    .align 16

hstr:
    .ascii "Hello, world!\n"
```

Language Elements

- **Symbols**
  - used to name things
    - code locations
      - functions
      - jump targets
    - data
      - variables
  - the dot symbol
    - refers to the current location
    - very handy to compute static string length
  - [syntax]()

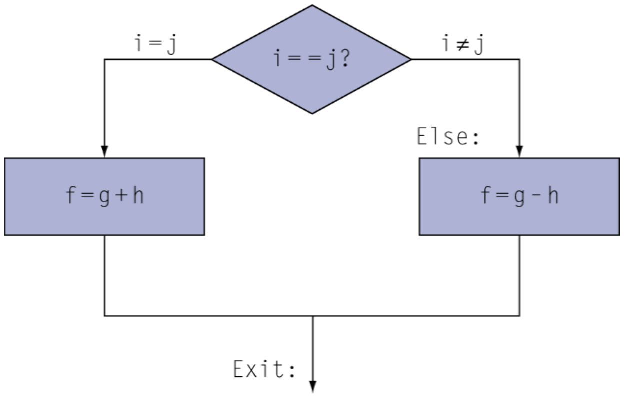
```
    .text
    .align 16

    .globl main

/*
 * main function
 */
main:
    movl  $1, %eax   # set %rax to 1
    …
    leaq  hstr(%rip), %rsi
    movl  $hlen, %edx
    …
    ret              # return to caller


    .data
    .align 16

hstr:
    .ascii "Hello, world!\n“
    hlen = . - hstr
```

Language Elements

- **Directives**
  - large number of directives
  - differ by architecture
  - **most commonly used directives**

.text	.ascii
.data	.asciz
.align	.byte
.extern	.int
.globl	.long
	.quad
	.short
	.size
	.skip
	.string
	.word
	.zero

HOCHSCHULE LUZERN



Altering Control Flow

Altering Control Flow

- Higher-level programming languages offer control-flow altering constructs

```c
int foo(int x, int y)
{
  int res = 0;

  if (x > y) x = x-y;

  while (x > 0) {
    res = res + y;
    x--;
  }

  return res;
}
```

- How can we achieve that with assembly code?

Branch Operations

- Processor ISAs offer branch operations to alter the sequential control flow
  - generic form

    ```
    branch <label>
    ```

  - instructs processor to continue execution at <label>
    - same as goto in higher-level programming languages
  - branch <label> implemented as
    PC = &label
  - **unconditional branch**
    - branch is always executed
  - differences by architectures
    - Intel: `jmp <label>`
    - RISC-V: `branch <label>`

```
    …
    subl   %eax, %edx
    movl   %edx, %eax
    jmp .L7
    …

.L7:
    …
```

Branch Operations

- **Conditional branches**
  - *conditionally* alter control flow
  - generic form

```
branch <condition>, <label>
```

  - instructs processor to continue execution at <label> if <condition> is true
    - if (condition) goto label
  - different ways to implement conditional branches

```
if (operand1 <cond> operand2) goto <label>
```

    - Intel:

```
cmp     operand2, operand1
j<cond> <label>
```

    - RISC-V:

```
b<cond> operand1, operand2, <label>
```

Recap: x86 Condition Codes

- **Single bit registers**

  CF Carry Flag (for unsigned)
  SF Sign Flag (for signed)
  ZF Zero Flag
  OF Overflow Flag (for signed)

- **Implicitly set by arithmetic operations**

  Example: **`addq/addl`** ***`src,dest`*** **↔ `t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

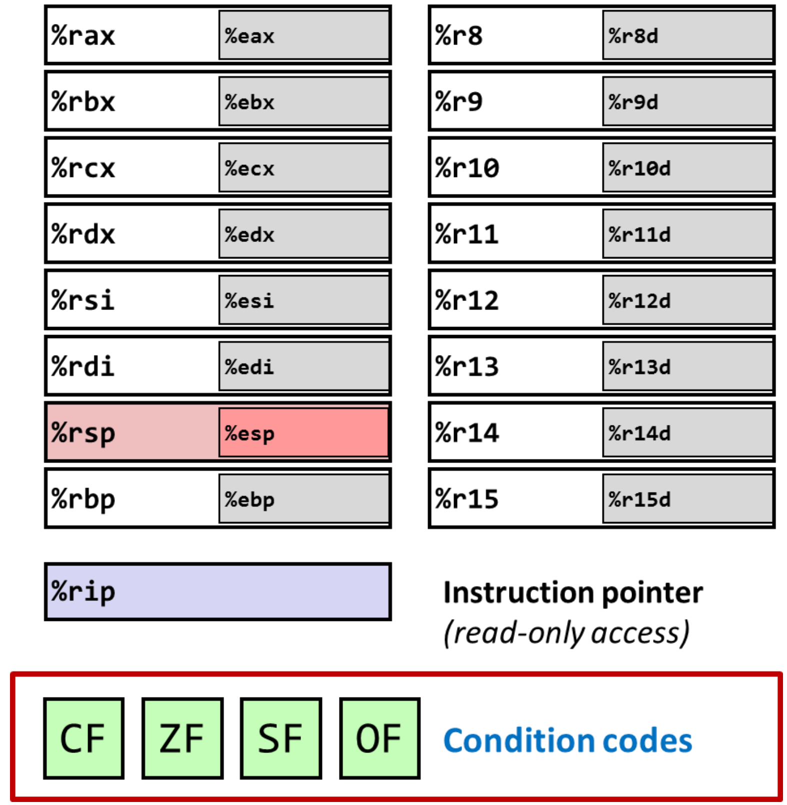
  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

  - *Not* set by **`lea`** instruction

- **Explicitly set by**
  - **`cmp S2, S1`** set condition codes based on **`S1-S2`**
  - **`test S2, S1`** set condition codes based on **`S1 & S2`**



Recap: x86 Condition Codes/Jumps

- setX: reading condition codes
  - set single byte, does not alter remaining 3 bytes
- jX: jump to different part of code depending on condition codes
- cmovX: conditional move

setX dest	jX target	cmovX S, reg	Condition	Description
	jmp		1	
sete	je	cmove	ZF	Equal / Zero
setne	jne	cmovne	~ZF	Not Equal / Not Zero
sets	js	cmovs	SF	Negative
setns	jns	cmovns	~SF	Nonnegative
setg	jg	cmovg	~(SF^OF)&~ZF	Greater (Signed)
setge	jge	cmovge	~(SF^OF)	Greater or Equal (Signed)
setl	jl	cmovl	(SF^OF)	Less (Signed)
setle	jle	cmovle	(SF^OF)|ZF	Less or Equal (Signed)
seta	ja	cmova	~CF&~ZF	Above (unsigned)
setb	jb	cmovg	CF	Below (unsigned)

HOCHSCHULE LUZERN

```
…
 movq    %rsi, %rdi
 cmpq    %rsi, %r8
 jl      .L2
 movq    %r8, %rdi
 call    foo@PLT
jmp    .L3
…
```

Altering Control Flow

If-then-else Constructs

Compiling If Statements

C code:

```
int cf_if(long a, long b) {
  int res;

  if (a >= b) res = foo(a, b);
  else res = bar(b, a);

  return res*2;
}
```

C code (goto version):

```
  if (a < b) goto L_false;
  res = foo(a, b)
  goto Exit;

L_false:
  res = bar(b, a)

Exit:
  return res*2;
```

Compiling If Statements

C code:

```
int cf_if(long a, long b) {
  int res;

  if (a >= b) res = foo(a, b);
  else res = bar(b, a);

  return res*2;
}
```

C code (goto version):

```
  if (a < b) goto L_false;
  res = foo(a, b)
  goto Exit;

L_false:
  res = bar(b, a)

Exit:
  return res*2;
```

Direct translation to x86 code:

```
cf_if:
   movq    %rdi, %r8
   subq    $8, %rsp
   movq    %rsi, %rdi
   cmpq    %rsi, %r8
   jl      .L_false
   movq    %r8, %rdi
   call    foo@PLT
   jmp     .L_exit

.L_false:
   movq    %r8, %rsi
   call    bar@PLT

.L_exit:
   addq    $8, %rsp
   addl    %eax, %eax
   ret
```

Compiling If Statements

C code:

```
int cf_if(long a, long b) {
  int res;

  if (a >= b) res = foo(a, b);
  else res = bar(b, a);

  return res*2;
}
```

C code (goto version):

```
  if (a < b) goto L_false;
  res = foo(a, b)
  goto Exit;

L_false:
  res = bar(b, a)

Exit:
  return res*2;
```

Direct translation to x86 code:

```
cf_if:
   movq   %rdi, %r8
   subq   $8, %rsp
   movq   %rsi, %rdi
   cmpq   %rsi, %r8
   jl     .L_false
   movq   %r8, %rdi
   call   foo@PLT
   jmp    .L_exit

.L_false:
   movq   %r8, %rsi
   call   bar@PLT

.L_exit:
   addq   $8, %rsp
   addl   %eax, %eax
   ret
```

Compiling If Statements

Code generated by GCC 9.3

```
cf_if:
   movq   %rdi, %r8
   subq   $8, %rsp
   movq   %rsi, %rdi
   cmpq   %rsi, %r8
   jl     .L2
   movq   %r8, %rdi
   call   foo@PLT
   addq   $8, %rsp
   addl   %eax, %eax
   ret
.L2:
   movq   %r8, %rsi
   call   bar@PLT
   addq   $8, %rsp
   addl   %eax, %eax
   ret
```

- Observations
  - minimize control flow instructions
  - minimize instructions in general
  - aligns stack points at 16-byte boundaries
  - condition sometimes reversed
    - and fix jumps to if/else bodies

```
for (Init; Test; Update)
    Body
```

```
Init;
while (Test ) {
    Body
    Update;
}
```

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while (Test);
done:
```

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

Altering Control Flow

Loop Constructs

HOCHSCHULE LUZERN

Loop Statements

- **Basic loop constructs**
  - do { body } while (cond);

```
do {
  i += 1;
} while (A[i] < i);
```

  - while (cond) { body }

```
while (A[i] == k) {
  i += 1;
};
```

  - for (init; cond; update) { body }

```
for (i=0; i<N; i++) {
  sum += A[i];
}
```


HOCHSCHULE LUZERN

General “Do-While” Translation

C Code

```
do
  Body
while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop;
```

- Body:
```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

General "While" Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
  while (Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Loop Statements

- **While as Do-While**
  - initial check
  - move condition to end

```
  while (A[i] == k) {
    i += 1;
  };
```

```
  if (A[i] != k) goto Exit;
  do {
    i += 1;
  } while (A[i] == k);
Exit:
```

"For" Loop → While → Do While → Goto

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while (Test);
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

“For” Loop Conversion Example

C Code

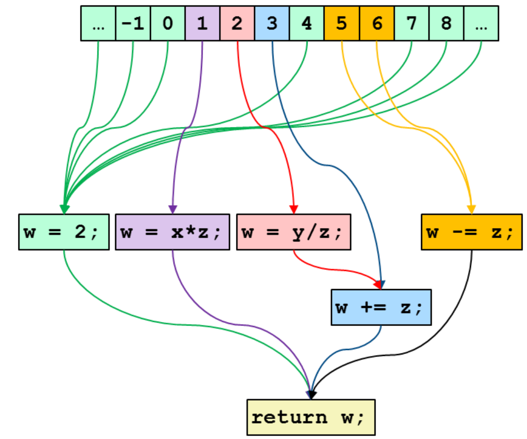
```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- In this case, the initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;                            Update
  if (i < WSIZE)
    goto loop;                    Test
 done:
  return result;
}
```


HOCHSCHULE LUZERN



Altering Control Flow

Switch Statements

```
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        // fall through
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

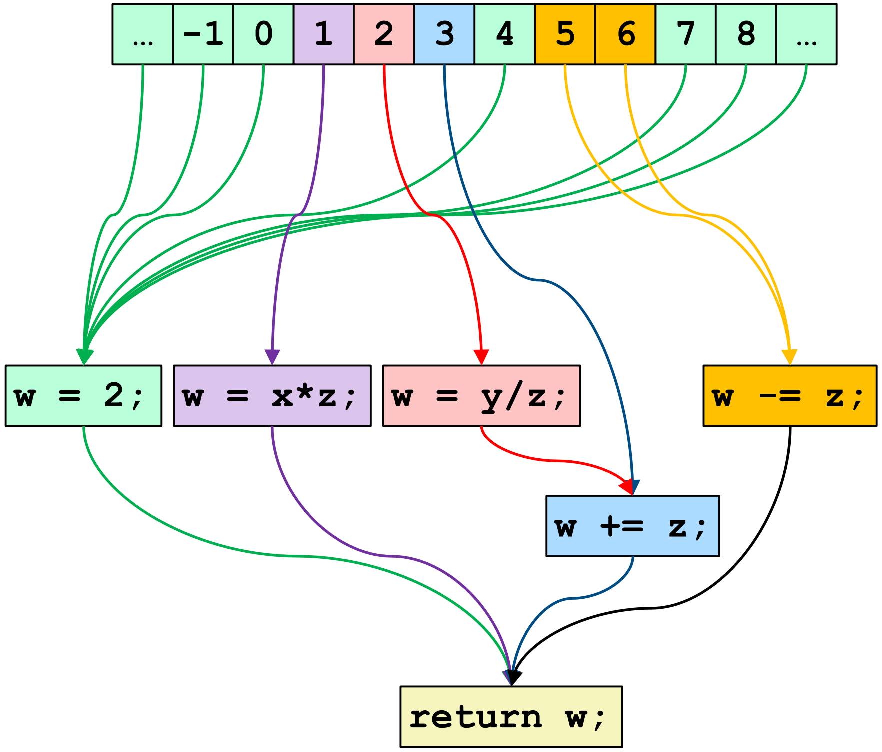
Switch Statement Example

- Multiple case labels (here: 5 & 6)
- Fall through cases (here: 2)
- Missing cases (here: 4)

```
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        // fall through
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

Switch Statement Example

- Execution flow in dependence of **x**



HOCHSCHULE LUZERN

```c
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        // fall through
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

(Inefficient) Implementation using nested if statements

```c
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  if (x == 1) {
    w = y*z;
  else {
    if (x == 2) {
      w = y/z;
      w += z;
    } else {
      if (x == 3) {
        w += z;
      } else {
        if ((x == 5) || (x == 6)) {
          w -= z;
        } else {
          w = 2;
        }
      }
    }
  }
  return w;
}
```

Efficient implementation using a jump table

```
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        // fall through
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

```
long switch_eg
   (long x, long y, long z)
{
  long w = 1;

  if ((x < 1) || (x > 6)) goto default;
  goto jtab[x];

  case1:
    w = y*z;
    goto end;
  case2:
    w = y/z;
  case3:
    w += z;
    goto end;
  case56:
    w -= z;
    goto end;
  default:
    w = 2;
  end:
  return w;
}
```

jtab

index	*target*
0	default
1	case1
2	case2
3	case3
4	default
5	case56
6	case56

Efficient Implementation using a Jump Table

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

```
jtab: Targ0
      Targ1
      Targ2
      •
      •
      •
      Targn-1
```

Jump Targets

```
Targ0:   Code Block 0

Targ1:   Code Block 1

Targ2:   Code Block 2

•
•
•

Targn-1: Code Block n–1
```

Approximate Translation

```
target = jtab[x];
goto *target;
```

Switch Statement Example (x86_64)

```
long switch_eg(long x, long y, long z)
{                          rdi     rsi      rdx
  long w = 1;
  switch(x) {
      . . .
  }
  return w;
}
```

Setup:

```
 switch_eg:
   movq    %rdx, %rcx         # rcx = copy of z
   cmpq    $6, %rdi           # compare x and 6
   ja      .L8                # if above (unsigned >) goto L8
```

What range of values takes default?

HOCHSCHULE LUZERN

Switch Statement Example (x86_64)

```
long switch_eg(long x, long y, long z)
{                          rdi      rsi      rdx
  long w = 1;
  switch(x) {
      . . .
  }
  return w;
}
```

Jump table

```
.section      .rodata
  .align 8
.L4:
  .quad       .L8  # x = 0
  .quad       .L3  # x = 1
  .quad       .L5  # x = 2
  .quad       .L9  # x = 3
  .quad       .L8  # x = 4
  .quad       .L7  # x = 5
  .quad       .L7  # x = 6
```

Setup:

```
switch_eg:
   movq    %rdx, %rcx          # rcx = copy of z
   cmpq    $6, %rdi            # compare x and 6
   ja      .L8                 # if above (unsigned >) goto L8
   jmp     *.L4(,%rdi,8)       # goto *jtab[x]
```

Notes: (1) **w** not initialized
(2) `jmp *<address>` executes an indirect jump

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at `.L4`
- Jumping
  - **Direct:** `jmp .L2`
  - Jump target is denoted by label `.L2`

  - **Indirect:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (labels have 64-bits = 8 bytes addresses on x86_64)
  - Fetch target from effective Address `.L4 + rdi*8`
    - Only for $0 \leq x \leq 6$

Jump table

```
.section      .rodata
  .align 8
.L4:
  .quad       .L8  # x = 0
  .quad       .L3  # x = 1
  .quad       .L5  # x = 2
  .quad       .L9  # x = 3
  .quad       .L8  # x = 4
  .quad       .L7  # x = 5
  .quad       .L7  # x = 6
```

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L5 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L7 # x = 5
  .quad     .L7 # x = 6
```

```
switch(x) {
  case 1:                 // .L3
      w = y*z;
      break;
  case 2:                 // .L5
      w = y/z;
      // fall through
  case 3:                 // .L9
      w += z;
      break;
  case 5: case 6:         // .L7
      w -= z;
      break;
  default:                // .L8
      w = 2;
}
```

Handling Fall-Through

```
long w = 1;
…
switch(x) {
  …
  case 2:
      w = y/z;
      // fall through
  case 3:
      w += z;
      break;
  …
}
```

```
case 3:
        w = 1;
        goto merge;
```

```
    case 2:
        w = y/z;
merge:
        w += z;
```

Full Example

```c
long switch_eg
   (long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        // fall through
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi
    ja      .L8
    jmp     *.L4(,%rdi,8)
.L3:
    movq    %rsi, %rax
    imulq   %rdx, %rax
    ret
.L5:
    movq    %rsi, %rax
    cqto
    idivq   %rcx
    jmp     .L6
.L9:
    movl    $1, %eax
.L6:
    addq    %rcx, %rax
    ret
.L7:
    movl    $1, %eax
    subq    %rdx, %rax
    ret
.L8:
    movl    $2, %eax
    ret
```

```
.L4:
 .quad   .L8
 .quad   .L3
 .quad   .L5
 .quad   .L9
 .quad   .L8
 .quad   .L7
 .quad   .L7
```

Understanding the Assembly Code

```
long switch_eg
   (long x, long y, long z)
{        rdi       rsi       rdx
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi
    ja      .L8
    jmp     *.L4(,%rdi,8)
.L3:
    movq    %rsi, %rax
    imulq   %rdx, %rax
    ret
.L5:
    movq    %rsi, %rax
    cqto
    idivq   %rcx
    jmp     .L6
.L9:
    movl    $1, %eax
.L6:
    addq    %rcx, %rax
    ret
.L7:
    movl    $1, %eax
    subq    %rdx, %rax
    ret
.L8:
    movl    $2, %eax
    ret
```

```
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

Understanding the Assembly Code

- `switch.s`: complete output of `gcc -m64 -O -S switch.c`
  - jump table embedded in code
  - separated into code and data section by assembler

```
	.file	"switch.c"
	.text
	.globl	switch_eg
	.type	switch_eg, @funct…
switch_eg:
.LFB0:
	.cfi_startproc
	movq	%rdx, %rcx
	cmpq	$6, %rdi
	ja	.L8
	jmp	*.L4(,%rdi,8)
	.section	.rodata
	.align 8
	.align 4
.L4:
	.quad	.L8
	.quad	.L3
	.quad	.L5
	.quad	.L9
	.quad	.L8
	.quad	.L7
	.quad	.L7
	.text
.L3:
	movq	%rsi, %rax
	imulq	%rdx, %rax
	ret
```

```
.L5:
	movq	%rsi, %rax
	cqto
	idivq	%rcx
	jmp	.L6
.L9:
	movl	$1, %eax
.L6:
	addq	%rcx, %rax
	ret
.L7:
	movl	$1, %eax
	subq	%rdx, %rax
	ret
.L8:
	movl	$2, %eax
	ret
	.cfi_endproc
.LFE0:
	.size	switch_eg, .
	.ident	"GCC: (Gentoo
		4.9.4 p1.0,
		pie-0.6.4) 4.9.4"
	.section	.note.GNU-stack,
		"",@progbits
```

Understanding the Assembly Code

- Noteworthy Features
  - Jump table avoids sequencing through cases
    - constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Compiler decides low-level implementation of switch
    (jump table, if-elsif-elsif-else, decision tree)

Object Code

Assembly Code

```
switch_eg:
  …
  ja    .L8                # if above (unsigned >) goto default
  jmp   *.L4(,%rdi,8)      # goto *jtab[x]
```

Disassembled Object Code

```
0000000000400508 <switch_eg>:
  …
  40050f:  77 07       ja     40053c <switch_eg+0x34>
  400511:  ff 24 fd …  jmp    *0x4005c8(,%rdi,8)
```

- Matching assembly to object code
  - code of default (**`.L8`**) located at address **`0x000000000040053c`**
  - jump table (**`.L4`**) located at address **`0x00000000004005c8`**

Object Code: Jump Table

- jump table does not show up in disassembled code
  - located in *data* section
  - can use objdump -d**D** <file> to also disassemble data sections, but result is not very useful

- better using GDB (GNU debugger)

```
$ gdb switch
(gdb) x/7xg 0x4005c8
0x4005c8:    0x000000000040053c    0x0000000000400518
0x4005d8:    0x0000000000400520    0x000000000040052a
0x4005e8:    0x000000000040053c    0x0000000000400533
0x4005f8:    0x0000000000400533
(gdb)
```

  - `x/7xg`: e*x*amine *7* he*x*adecimal format "*g*iant words" (8 bytes)
  - Use command "`help x`" to get format documentation

HOCHSCHULE LUZERN

Object Code: Deciphering the Jump Table

```
(gdb) x/7xg 0x4005c8
0x4005c8:    0x000000000040053c    0x0000000000400518
0x4005d8:    0x0000000000400520    0x000000000040052a
0x4005e8:    0x000000000040053c    0x0000000000400533
0x4005f8:    0x0000000000400533
```

Address	Value	x
0x4005c8	0x40053c	0
0x4005d0	0x400518	1
0x4005d8	0x400520	2
0x4005e0	0x40052a	3
0x4005e8	0x40053c	4
0x4005f0	0x400533	5
0x4005f8	0x400533	6

Code and Disassembled Executable

```c
long switch_eg
   (long x,
    long y,
    long z)
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

```
0000000000400508 <switch_eg>:
  400508:    48 89 d1                mov    %rdx,%rcx
  40050b:    48 83 ff 06             cmp    $0x6,%rdi
  40050f:    77 2b                   ja     40053c
  400511:    ff 24 fd c8 05 40 00    jmpq   *0x4005c8(,%rdi,8)
  400518:    48 89 f0                mov    %rsi,%rax
  40051b:    48 0f af c2             imul   %rdx,%rax
  40051f:    c3                      retq
  400520:    48 89 f0                mov    %rsi,%rax
  400523:    48 99                   cqto
  400525:    48 f7 f9                idiv   %rcx
  400528:    eb 05                   jmp    40052f
  40052a:    b8 01 00 00 00          mov    $0x1,%eax
  40052f:    48 01 c8                add    %rcx,%rax
  400532:    c3                      retq
  400533:    b8 01 00 00 00          mov    $0x1,%eax
  400538:    48 29 d0                sub    %rdx,%rax
  40053b:    c3                      retq
  40053c:    b8 02 00 00 00          mov    $0x2,%eax
  400541:    c3                      retq
  400542:    66 2e 0f 1f 84 00 00    nopw   %cs:0x0(…
  400549:    00 00 00
  40054c:    0f 1f 40 00             nopl   0x0(%rax)
```

Disassembled Executable: Match w/ Code

```
long switch_eg
   (long x, in rdi
    long y, in rsi
    long z) in rdx
{
  long w = 1;
  switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
  }
  return w;
}
```

```
0000000000400508 <switch_eg>:
  400508:    48 89 d1              mov    %rdx,%rcx
  40050b:    48 83 ff 06           cmp    $0x6,%rdi
  40050f:    77 2b                 ja     40053c
  400511:    ff 24 fd c8 05 40 00  jmpq   *0x4005c8(,%rdi,8)
  400518:    48 89 f0              mov    %rsi,%rax
  40051b:    48 0f af c2           imul   %rdx,%rax
  40051f:    c3                    retq
  400520:    48 89 f0              mov    %rsi,%rax
  400523:    48 99                 cqto
  400525:    48 f7 f9              idiv   %rcx
  400528:    eb 05                 jmp    40052f
  40052a:    b8 01 00 00 00        mov    $0x1,%eax
  40052f:    48 01 c8              add    %rcx,%rax
  400532:    c3                    retq
  400533:    b8 01 00 00 00        mov    $0x1,%eax
  400538:    48 29 d0              sub    %rdx,%rax
  40053b:    c3                    retq
  40053c:    b8 02 00 00 00        mov    $0x2,%eax
  400541:    c3                    retq
  400542:    66 2e 0f 1f 84 00 00  nopw   %cs:0x0(…
  400549:    00 00 00
  40054c:    0f 1f 40 00           nopl   0x0(%rax)
```

Disassembled Executable: Match w/ Table

Jump table

Value	x
40053c	0
400518	1
400520	2
40052a	3
40053c	4
400533	5
400533	6

```
0000000000400508 <switch_eg>:
  400508:    48 89 d1                 mov    %rdx,%rcx
  40050b:    48 83 ff 06              cmp    $0x6,%rdi
  40050f:    77 2b                    ja     40053c
  400511:    ff 24 fd c8 05 40 00     jmpq   *0x4005c8(,%rdi,8)
  400518:    48 89 f0                 mov    %rsi,%rax
  40051b:    48 0f af c2              imul   %rdx,%rax
  40051f:    c3                       retq
  400520:    48 89 f0                 mov    %rsi,%rax
  400523:    48 99                    cqto
  400525:    48 f7 f9                 idiv   %rcx
  400528:    eb 05                    jmp    40052f
  40052a:    b8 01 00 00 00           mov    $0x1,%eax
  40052f:    48 01 c8                 add    %rcx,%rax
  400532:    c3                       retq
  400533:    b8 01 00 00 00           mov    $0x1,%eax
  400538:    48 29 d0                 sub    %rdx,%rax
  40053b:    c3                       retq
  40053c:    b8 02 00 00 00           mov    $0x2,%eax
  400541:    c3                       retq
  400542:    66 2e 0f 1f 84 00 00     nopw   %cs:0x0(…
  400549:    00 00 00
  40054c:    0f 1f 40 00              nopl   0x0(%rax)
```


HOCHSCHULE LUZERN

IA32 Switch Implementation

- Same general idea using 32-bit code
- Table entries 32 bits long (pointers)

```
long switch_eg(long x,   long y,    long z)
{                     ebp+8       ebp+12     ebp+16
  long w = 1;
  switch(x) {
    …
  }
  return w;
}
```

Jump table

```
.section      .rodata
  .align 4
.L7:
  .long       .L2  # x = 0
  .long       .L3  # x = 1
  .long       .L4  # x = 2
  .long       .L5  # x = 3
  .long       .L2  # x = 4
  .long       .L6  # x = 5
  .long       .L6  # x = 6
```

Setup:

```
switch_eg:
    pushl    %ebp              # setup
    movl     %esp, %ebp        # setup
    movl     8(%ebp), %eax     # eax = x
    cmpl     $6, %eax          # compare x and 6
    ja       .L2               # if above (unsigned >) goto default
    jmp      *.L7(,%eax,4)     # goto *jtab[x]
```

Disassembled Executable (IA32)

```
8048422:    b8 02 00 00 00        mov    $0x2,%eax
8048427:    eb 2a                 jmp    8048453 <switch_eg+0x43>
8048429:    b8 01 00 00 00        mov    $0x1,%eax
804842e:    66 90                 xchg   %ax,%ax
8048430:    eb 14                 jmp    8048446 <switch_eg+0x36>
8048432:    8b 45 10              mov    0x10(%ebp),%eax
8048435:    0f af 45 0c           imul   0xc(%ebp),%eax
8048439:    eb 18                 jmp    8048453 <switch_eg+0x43>
804843b:    8b 55 0c              mov    0xc(%ebp),%edx
804843e:    89 d0                 mov    %edx,%eax
8048440:    c1 fa 1f              sar    $0x1f,%edx
8048443:    f7 7d 10              idivl  0x10(%ebp)
8048446:    03 45 10              add    0x10(%ebp),%eax
8048449:    eb 08                 jmp    8048453 <switch_eg+0x43>
804844b:    b8 01 00 00 00        mov    $0x1,%eax
8048450:    2b 45 10              sub    0x10(%ebp),%eax
8048453:    5d                    pop    %ebp
8048454:    c3                    ret
```

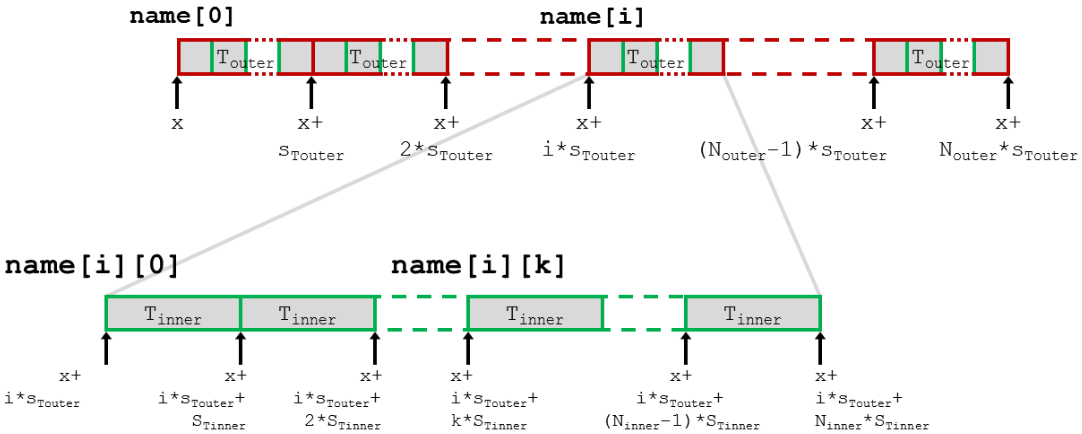
- single exit at bottom, code snippets jump to exit
- purpose of instruction at 0x804842e?

Control Transfer Structures: Summary

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch

- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control

- Standard Techniques
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees



Composite Data Structures

HOCHSCHULE LUZERN

Composite Data Structures

- **Arrays**
- Structures
- Unions

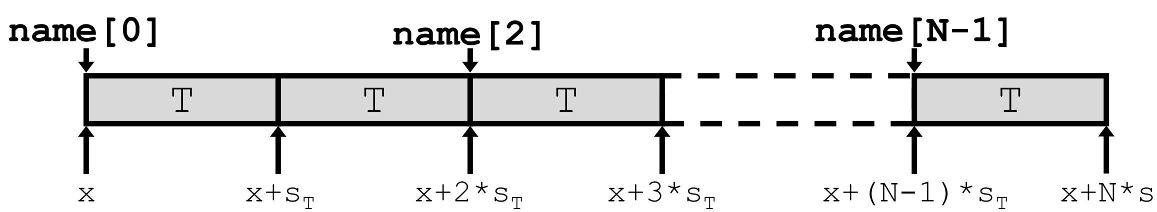
Recap: Arrays

- **Declaration**

```
<T> name[<N>]
```

- **Size**
  - one element: $s_T = \mathtt{sizeof(T)}$
  - entire array: $s_A = N * s_T$

- **Memory layout**

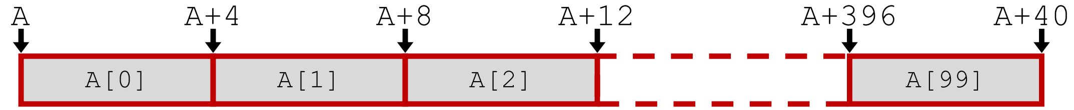


- **Address of i-th element**

$$adr_i = name + i*s_T$$

- **Alignment**
  - array alignment = alignment of base type T

Example: 1-d Array



One-dimensional array

```c
#define N 100

int A[N];

int get(int i)
{
  return A[i];
}

int sum(void)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i];
  }

  return sum;
}
```

array1.c

```
$ gcc -m64 -O2 -S array1.c
```

```
get:
  movslq %edi, %rdi
  movl   A(,%rdi,4), %eax
  ret

sum:
  movl   $A, %edx
  xorl   %eax, %eax
.L4:
  addl   (%rdx), %eax
  addq   $4, %rdx
  cmpq   $A+400, %rdx
  jne    .L4
  rep ret
```

array1.s

`sum` compiled as:

```c
int sum(void)
{
  int *ap = A;
  int sum = 0;

  do {
    sum += *ap++;
  while (ap != A + 100);

  return sum;
}
```


HOCHSCHULE LUZERN

Recap: Multidimensional Arrays

- **formed when <type> is an array type**

$$\texttt{<}T_{outer}\texttt{>}\ \texttt{name[<}N_{outer}\texttt{>]}$$

$$\texttt{<}T_{outer}\texttt{>} = \texttt{<}T_{inner}\texttt{>}\ \ldots\texttt{[<}N_{inner}\texttt{>]}$$

combined notation

$$\texttt{<}T_{inner}\texttt{>}\ \texttt{<name>[}N_{outer}\texttt{][}N_{inner}\texttt{]}$$

- **Size**
  - one element: $s_{T_{outer}} = \texttt{sizeof}(T_{outer}) = N_{inner} * s_{T_{inner}}$
  - entire array: $s_{A_{outer}} = N_{outer} * s_{T_{outer}} = N_{outer} * N_{inner} * s_{T_{inner}}$

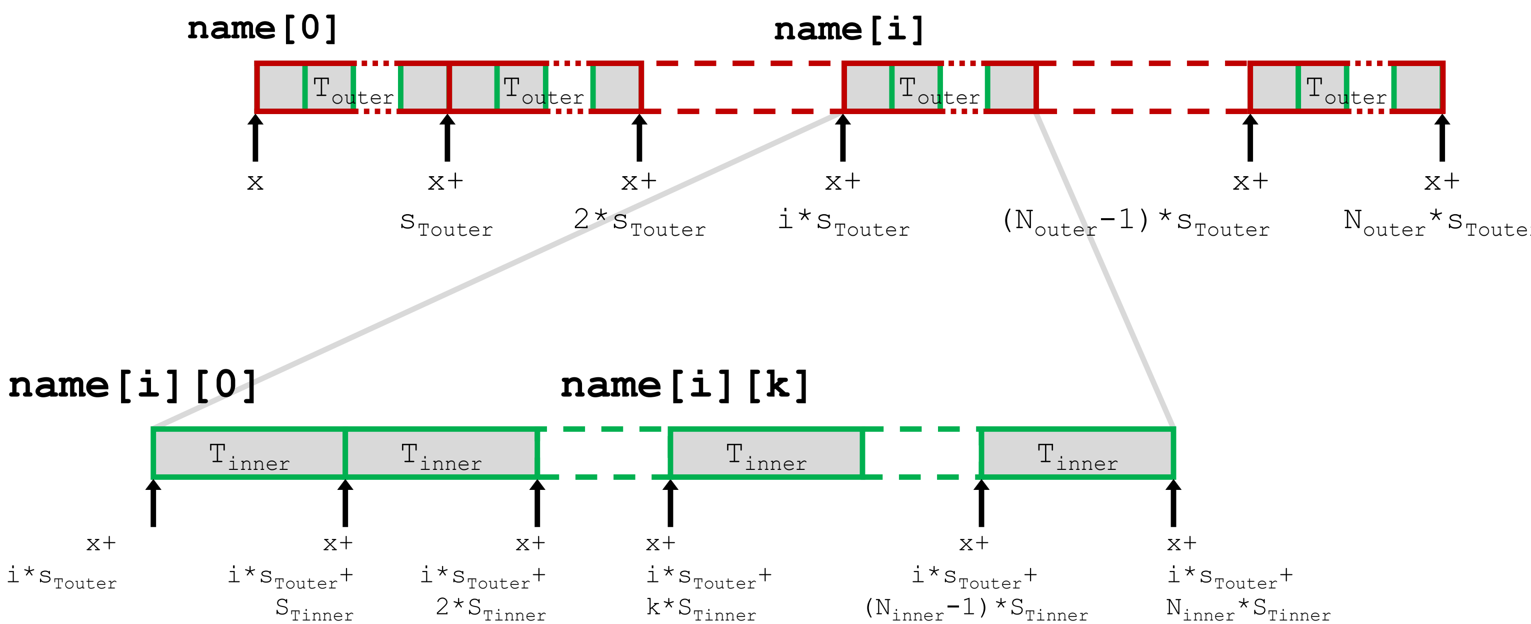
Recap: Multidimensional Arrays

- **Memory layout ("row-major" layout)**

`<T_outer> name[<N_outer>]`

`<T_outer> = <T_inner> …[<N_inner>]`

`<T_inner> <name>[N_outer][N_inner]`



$\Downarrow$

x+ $i*N_{inner}*S_{Tinner}$+ $k*S_{Tinner}$ $\Rightarrow$ **name + S_{Tinner}*(k + N_{inner}*i)**

- **Address of element [i,k]**
  - note how N_{outer} is not needed to compute name[i][k]

HOCHSCHULE LUZERN

Recap: Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\texttt{<}T_{base}\texttt{>}\ \texttt{<name>}[N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Size**
  - entire array

$$s_{D_n} = N_{D_n} * s_{T_{Dn-1}} = N_{D_n} * N_{D_{n-1}} * s_{T_{Dn-2}} = \ldots = \mathbf{N_{D_n} * N_{D_{n-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

  - subdimension k (n ≥ k ≥ 1)

$$s_{D_k} = N_{D_k} * s_{T_{Dk-1}} = \ldots = \mathbf{N_{D_k} * N_{D_{k-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

Recap: Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\texttt{<}T_{base}\texttt{>}\ \texttt{<name>}[N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Address**
  - $\texttt{name}[i_{D_n}][i_{D_{n-1}}]\ldots[i_{D_2}][i_{D_1}]$

$$\textbf{name} + s_{T_{base}} * (i_{D_1} + N_{D_1} * (i_{D_2} + N_{D_2} * (\ldots + N_{D_{n-1}} * i_{D_n})\ldots))$$

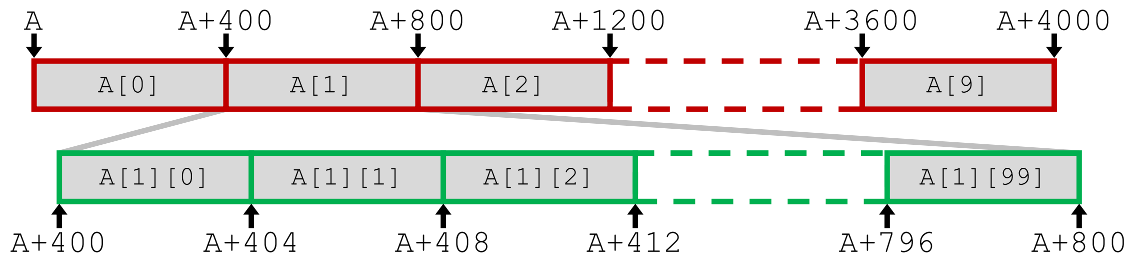
  - again, note how $\mathbf{N_{D_n}}$ is not required for the address computation

    → this is why you can declare arrays with an open outermost dimension such as

```
int A[][N][K];
```


HOCHSCHULE LUZERN

Example: 2-d Array



Two-dimensional array

```c
#define N 10
#define M 100

int A[N][M];

int get(int i, int j)
{
  return A[i][j];
}

int sum(void)
{
  int i, j, sum = 0;

  for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
      sum += A[i][j];
    }
  }

  return sum;
}
```

array2.c

```
get:
  movslq %edi, %rdi
  movslq %esi, %rsi
  leaq   (%rdi,%rdi,4), %rax
  leaq   (%rax,%rax,4), %rax
  leaq   (%rsi,%rax,4), %rax
  movl   A(,%rax,4), %eax
  ret

sum:
  movl    $A, %edx
  movl    $A+4000, %esi
  xorl    %eax, %eax
.L3:
  leaq    400(%rdx), %rcx
.L4:
  addl    (%rdx), %eax
  addq    $4, %rdx
  cmpq    %rcx, %rdx
  jne     .L4
  cmpq    %rdx, %rsi
  jne     .L3
  rep ret
```

array2.s

```
$ gcc -m64 -O2 -S array2.c
```


HOCHSCHULE LUZERN

Example: 2-d Array

Two-dimensional array (cont'd)

```
#define N 10
#define M 100

int A[N][M];

int sum(void)
{
  int i, j, sum = 0;

  for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
      sum += A[i][j];
    }
  }

  return sum;
}
```

array2.c

```
sum:
  movl    $A, %edx
  movl    $A+4000, %esi
  xorl    %eax, %eax
.L3:
  leaq    400(%rdx), %rcx
.L4:
  addl    (%rdx), %eax
  addq    $4, %rdx
  cmpq    %rcx, %rdx
  jne     .L4
  cmpq    %rdx, %rsi
  jne     .L3
  rep ret
```

array2.s

`sum` compiled as:

```
int sum(void)
{
  int *Aptr = &A[0][0];
  int *endptr = &A[N][0];
  int sum = 0;

  do {
    int *endrow = Aptr + M;
    do {
      sum += *Aptr++;
    } while (Aptr < endrow);
  } while (Aptr < endptr);

  return sum;
}
```


HOCHSCHULE LUZERN

Example: Sum of a Column in a 2-D Array

Adding up a column in a 2-d array

```
#define N 100

int A[N][N];

int colsum(int c)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i][c];
  }

  return sum;
}
```

array3.c

```
$ gcc -m64 -S -O2 array3.c
```

```
colsum:
  movslq %edi, %rcx
  xorl   %eax, %eax
  salq   $2, %rcx
  leaq   A(%rcx), %rdx
  addq   $A+40000, %rcx
.L2:
  addl   (%rdx), %eax
  addq   $400, %rdx
  cmpq   %rcx, %rdx
  jne    .L2
  rep ret
```

array3.s

`colsum` compiled as:

```
int colsum(int c)
{
  int sum = 0;
  int *ap = A+c;
  int *endp = A+10000+c;

  do {
    sum += *ap;
    ap = ap + 100;
  while (ap != endp)

  return sum;
}
```


HOCHSCHULE LUZERN

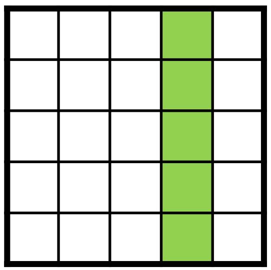
Example: Sum of a Column in a 2-D Array

Adding up a column in a 2-d array

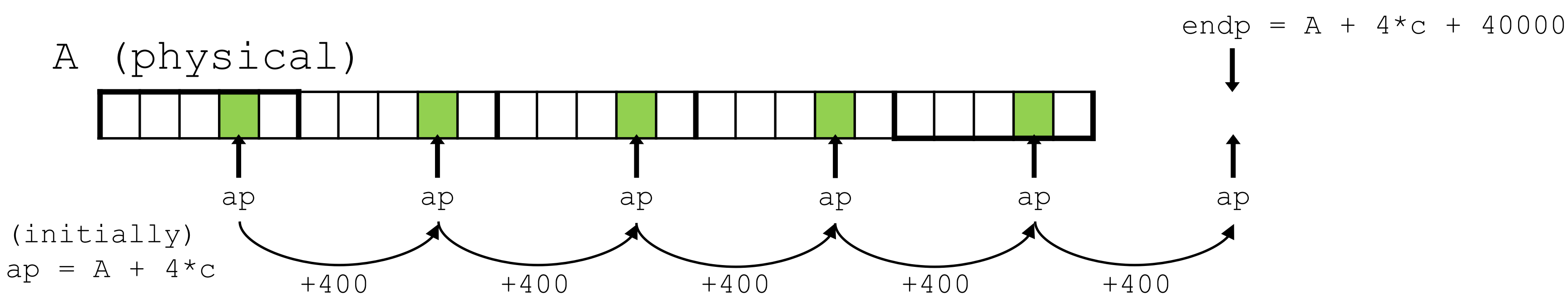
```
colsum:
  movslq %edi, %rcx
  xorl   %eax, %eax
  salq   $2, %rcx
  leaq   A(%rcx), %rdx
  addq   $A+40000, %rcx
.L2:
  addl   (%rdx), %eax
  addq   $400, %rdx
  cmpq   %rcx, %rdx
  jne    .L2
  rep ret
```

array3.s

A (logical)

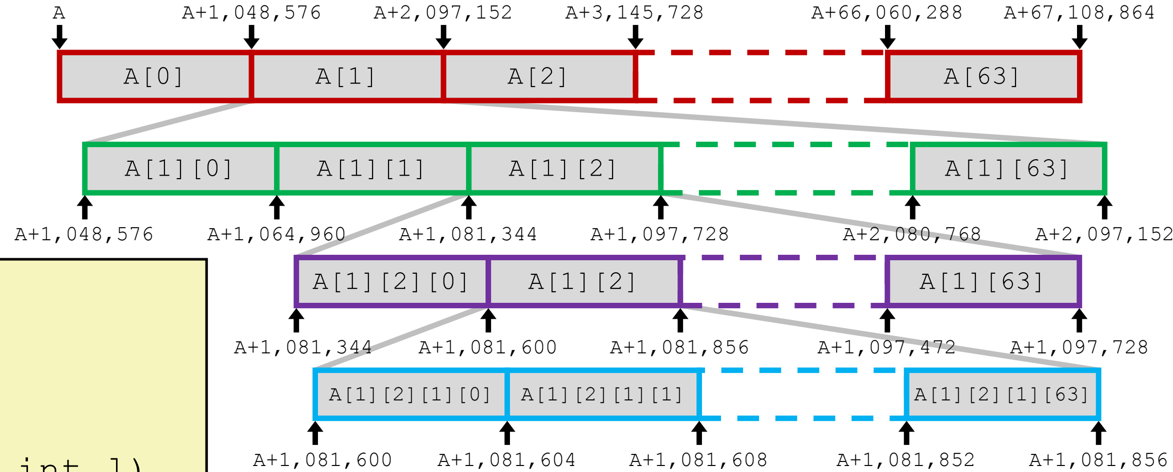


A (physical)



HOCHSCHULE LUZERN

Example: 4-d Array



4-dimensional array

```
#define N 64

int A[N][N][N][N];

int get(int i, int j, int k, int l)
{
  return A[i][j][k][l];
}

int sum(void)
{
  int i, j, k, l, sum = 0;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        for (l=0; l<N; l++)
          sum += A[i][j][k][l];

  return sum;
}
                                    array4.c
```

```
$ gcc -m64 -S -O2 array4.c
```

size of one element in:

1st dimension	1,048,576	bytes
2nd dimension	16,384	bytes
3rd dimension	256	bytes
4th dimension	4	bytes

```
get:
  movslq %edi, %rdi
  movslq %esi, %rsi
  movslq %edx, %rdx
  salq   $6, %rdi
  movslq %ecx, %rcx
  addq   %rdi, %rsi
  salq   $6, %rsi
  addq   %rsi, %rdx
  salq   $6, %rdx
  addq   %rcx, %rdx
  movl   A(,%rdx,4), %eax
  ret
                          array4.s
```


HOCHSCHULE LUZERN

Example: 4-d Array

size of entire array	67,108,864	bytes
size of one element in:		
1^{st} dimension	1,048,576	bytes
2^{nd} dimension	16,384	bytes
3^{rd} dimension	256	bytes
4^{th} dimension	4	bytes

4-dimensional array

```
#define N 64
int A[N][N][N][N];
int sum(void)
{ int i, j, k, l, sum = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        for (l=0; l<N; l++)
          sum += A[i][j][k][l];
  return sum;
}
```
array4.c

gcc

```
sum:
  movl    $A+1064960, %edi
  movl    $A+68173824, %r8d
  xorl    %eax, %eax
.L3:
  leaq    -1048576(%rdi), %rsi
.L10:
  leaq    -16384(%rsi), %rdx
.L8:
  leaq    256(%rdx), %rcx
.L6:
  addl    (%rdx), %eax
  addq    $4, %rdx
  cmpq    %rcx, %rdx
  jne     .L6
  cmpq    %rdx, %rsi
  jne     .L8
  addq    $16384, %rsi
  cmpq    %rdi, %rsi
  jne     .L10
  leaq    1048576(%rsi), %rdi
  cmpq    %r8, %rdi
  jne     .L3
  rep ret
```
array4.s

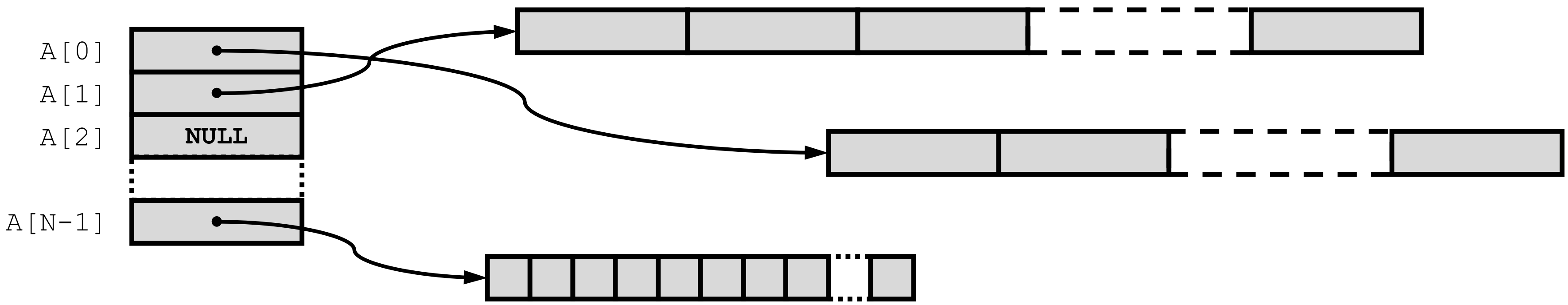
asm-to-C

```
int sum(void)
{     rdi = A+1,048,576+16,384;
      r8  = A+67,108,864+1,048,576+16384;
      sum (=eax) = 0;

l3:   rsi = rdi - 1,048,576;
l10:  rdx = rsi - 16,384;
l8:   rcx = rdx + 256;
l6:   sum += *rdx++;
      if (rdx != rcx) goto l6;
      if (rdx != rsi) goto l8;
      rsi = rsi + 16,384;
      if (rsi != rdi) goto l10;
      rdi = rsi + 1,048,576;
      if (rdi != r8) goto l3;

      return sum;
}
```

Recap: Multilevel Arrays

- A multilevel array is simply an array of pointers to another array

```
int *A[N]
```



- pointed-to arrays can have different sizes
- elements in the pointer array can be NULL
- one extra memory access per pointer indirection
- both arrays, pointer array and pointed-to arrays can be multidimensional
- address calculation separate
  pointer array: element = pointer; pointed-to array: any type

Example: Multidimensional vs. Multilevel Array

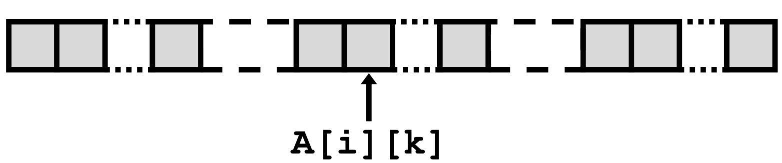
Multidimensional (nested) array

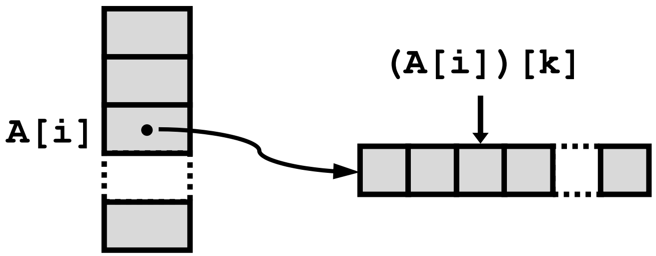
```
int A[N][M];

int get(int i, int k)
{
  return A[i][k];
}
```

Multi-level array

```
int *A[N];

int get(int i, int k)
{
  return A[i][k];
}
```

C code looks identical, but access very different:





MEM[A+4*(k + M*i)]

MEM[MEM[A+4*i] + 4*k]

Example: Multidimensional vs. Multilevel Array

Multidimensional (nested) array

```
#define N  5
#define M 10

long int A[N][M];

long int get(long int i, long int k)
{
  return A[i][k];
}
```

```
# parameters: %rdi = i, %rsi = k

<get>:
  0:  lea    (%rdi,%rdi,4),%rax
  4:  lea    (%rsi,%rax,2),%rax
  8:  mov    A(,%rax,8),%rax
 10:  retq
```

MEM[A+8*(k + 10*i)]

Multi-level array

```
#define N 5


long int *A[N];

long int get(long int i, long int k)
{
  return A[i][k];
}
```

```
# parameters: %rdi = i, %rsi = k

<get>:
  0: mov    A(,%rdi,8),%rax
  8: mov    (%rax,%rsi,8),%rax
  c: retq
```

MEM[MEM[A+8*i] + 8*k]

HOCHSCHULE LUZERN

Example: Multidimensional vs. Multilevel Array

Multidimensional (nested) array

```
#define N  5

long int A[N][N][N];

long int get(long int i, long int k,
             long int l)
{
  return A[i][k][l];
}
```

```
# parameters: %rdi=i, %rsi=k, %rdx=l

<get>:
  0: lea    (%rdi,%rdi,4),%rax
  4: lea    (%rsi,%rsi,4),%rcx
  8: lea    (%rax,%rax,4),%rax
  c: add    %rcx,%rax
  f: add    %rax,%rdx
 12: mov    A(,%rdx,8),%rax
 1a: retq
```

MEM[A+8*(l + 5*(k + 5*i))]

Multi-level array

```
#define N 5

long int **A[N];

long int get(long int i, long int k,
             long int l)
{
  return A[i][k][l];
}
```

```
# parameters: %rdi=i, %rsi=k, %rdx=l

<get>:
  0: mov    A(,%rdi,8),%rax
  8: mov    (%rax,%rsi,8),%rax
  c: mov    (%rax,%rdx,8),%rax
 10: retq
```

MEM[MEM[MEM[A+8*i] + 8*k] + 8*l]

Composite Data Structures

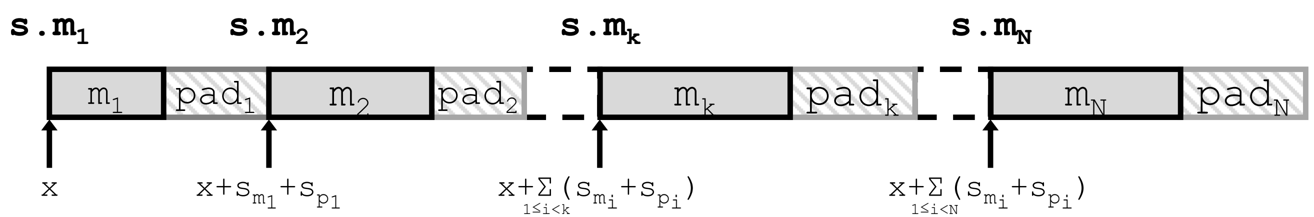
- Arrays
- **Structures**
- Unions

Recap: Structures

- **Declaration**

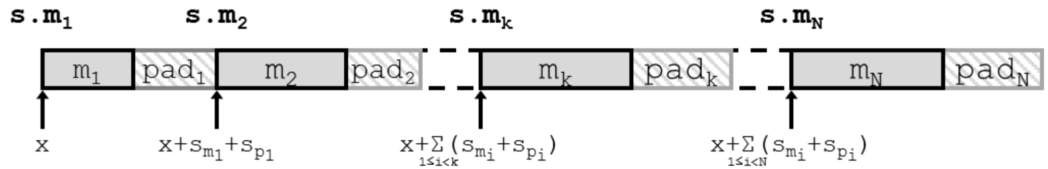
```
struct name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i in-order, non-overlapping, and properly aligned



- **Alignment**
  - struct alignment = maximum alignment requirement of any of its members
  - member alignment = alignment requirement of member type
    - padding: “holes” in the memory layout to maintain alignment requirements (denoted pad_i above)

Recap: Structures



- **Address of k-th member**
  - start of struct plus sum of sizes of all 1..k-1 members and paddings

$$\mathtt{adr}_{m_k} = x + \sum_{1 \le i < k} (s_{m_i} + s_{p_i})$$

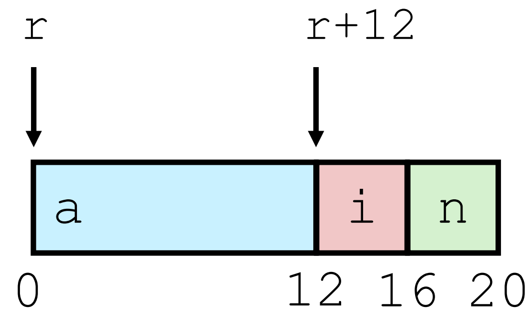
- **Size of struct**

$$s_s = \sum_{1 \le i \le N} (s_{m_i} + s_{p_i})$$

  - the last padding ($\mathtt{pad}_N$) is chosen such that the size of the struct is the next multiple of the biggest alignment requirement of any of its members
    - this comes in handy when declaring arrays of structs (all elements of the array will be automatically aligned)

Example: Structure Access

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

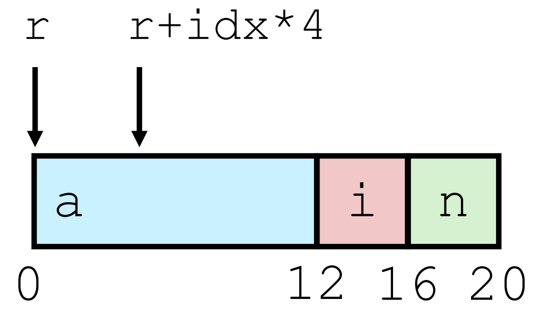
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

x86_64 assembly

```
 # %edx = val
 # %rax = r
movl %edx, 12(%rax)    # Mem[r+12] = val
```

Example: Generating Pointer to Member

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Arguments
    - `%rdi: r`
    - `%rsi: idx`

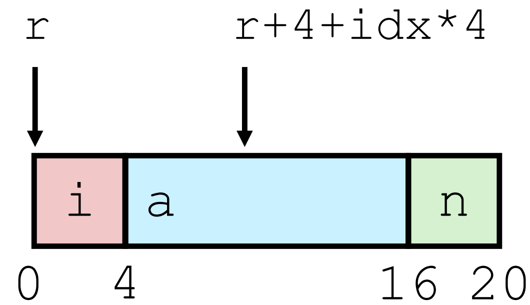
```
int *get_ap
 (struct rec *r, long int idx)
{
  return &r->a[idx];
}
```

x86-64 assembly

```
get_ap:
  leaq   (%rdi,%rsi,4), %rax
  ret
```

Example: Generating Pointer to Member

```
struct rec {
  int i;
  int a[3];
  struct rec *n;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Arguments
    - `%rdi: r`
    - `%rsi: idx`

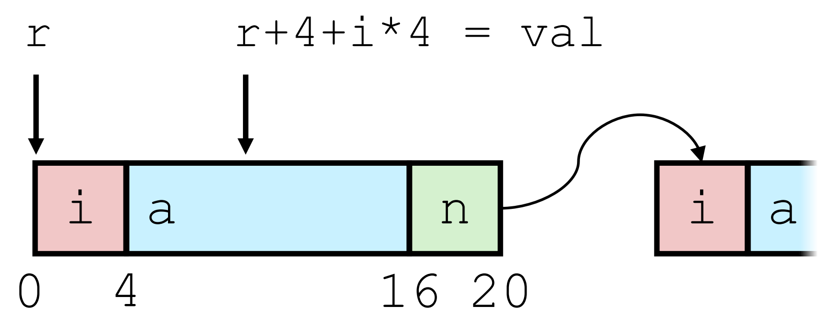
```
int *get_ap
 (struct rec *r, long int idx)
{
  return &r->a[idx];
}
```

x86-64 assembly

```
get_ap:
  leaq   4(%rdi,%rsi,4), %rax
  ret
```

Example: Following a Linked List

```
#include <stdio.h>

struct rec {
  int i;
  int a[3];
  struct rec *n;
};

void set_val(struct rec *r, int val)
{
  while (r != NULL) {
    int i = r->i;
    r->a[i] = val;
    r = r->n;
  }
}
```



- Set element `r->a[r->i]` to `val` in entire linked list
  - Arguments
    - `%rdi: r`
    - `%rsi(%esi): val`

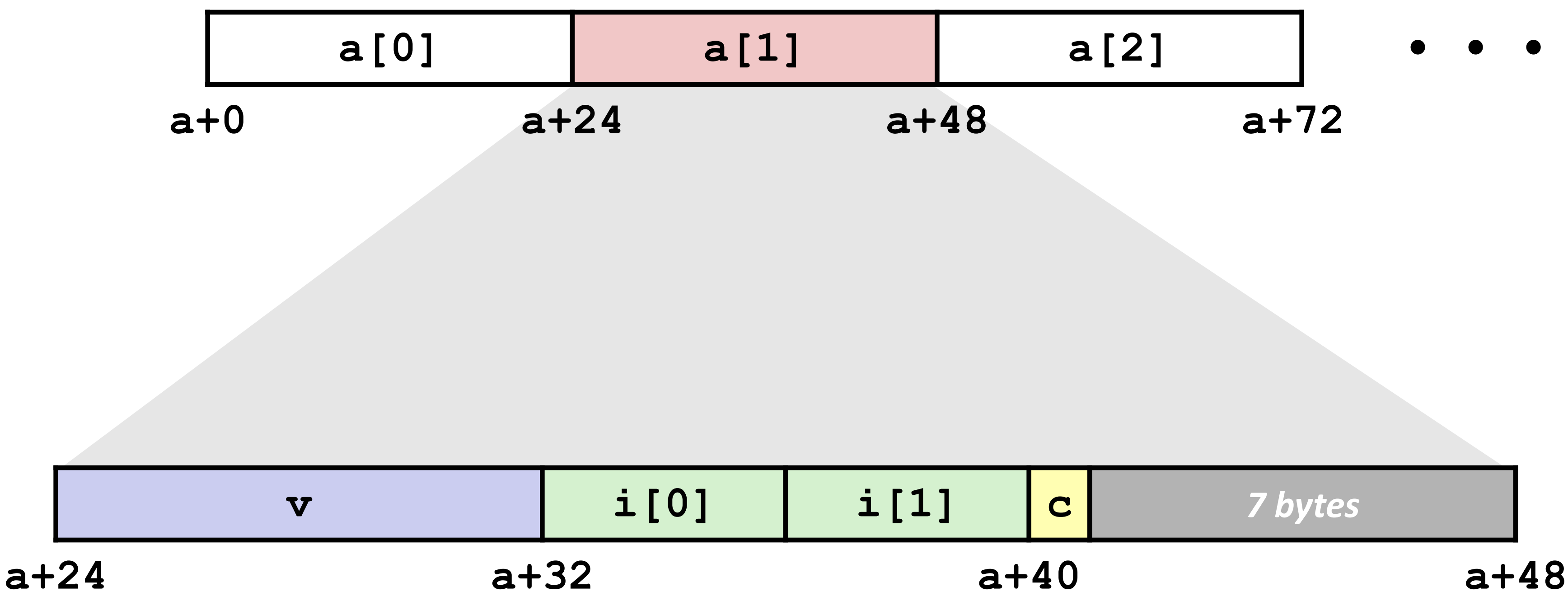
x86-64 assembly

```
set_val:
  jmp     .L7
.L5:
  movslq  (%rdi), %rax           # r->i
  movl    %esi, 4(%rdi,%rax,4)   # r->a[r->i]
  movq    16(%rdi), %rdi         # r->next
.L7:
  testq   %rdi, %rdi             # r
  jne     .L5
  rep ret
```

Example: Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

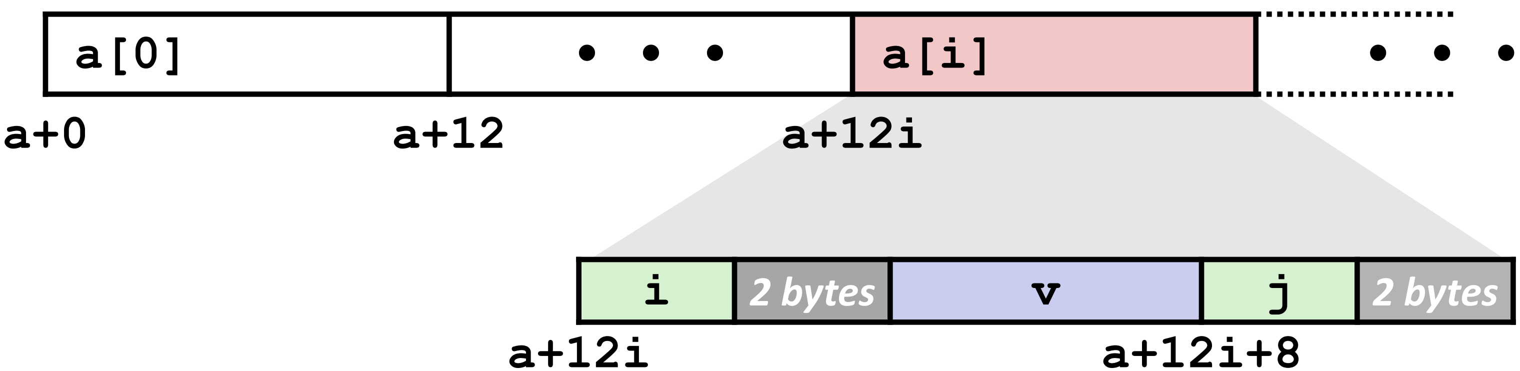


HOCHSCHULE LUZERN

Example: Accessing Array Elements

- Compute array offset 12i
  - sizeof(S3), including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Composite Data Structures

- Data Alignment
- Arrays
- Structures
- **Unions**

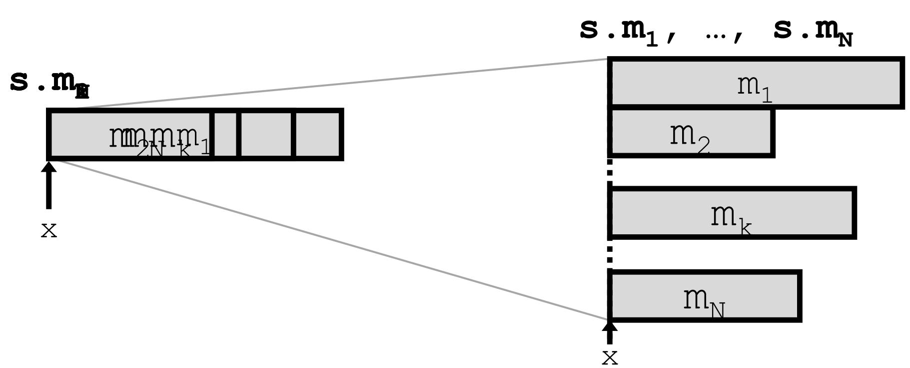
HOCHSCHULE LUZERN

Recap: Unions

- **Declaration**

```
union name {
   <T1> <m1>;
   <T2> <m2>;
   …
   <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i, and properly aligned
  - all members *are located at offset 0 and overlap in memory*



HOCHSCHULE LUZERN

Recap: Unions

- **Alignment**
  - union alignment = maximum alignment requirement of any of its members
  - member alignment = alignment requirement of member type

- **Address of k-th member**
  - start of union

$$\mathtt{adr}_{\mathtt{m}_\mathtt{k}} = \mathtt{x}$$

- **Size of union**

$$\mathtt{s}_\mathtt{u} = \max_{1 \le i \le N}(\mathtt{s}_{\mathtt{m}_\mathtt{i}})$$

HOCHSCHULE LUZERN

Example: Using Unions to Access Bit Patterns

- Task: print bit pattern of a floating point number

- Try 1:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  return (unsigned int)f;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}                                              convert1.c
```

The assembly reveals

```
get_bitpattern:
  subl    $20, %esp
  fnstcw  14(%esp)
  movzwl  14(%esp), %eax
  flds    24(%esp)
  movb    $12, %ah
  movw    %ax, 12(%esp)
  fldcw   12(%esp)
  fistpq  (%esp)      # convert float to int
  fldcw   14(%esp)
  movl    (%esp), %eax
  addl    $20, %esp
  ret
```

indeed, this is a conversion float → int.

Output looks suspiciously wrong:

```
$ gcc -m32 -O3 -o convert1 convert1.c
$ ./convert1
3.1415927410 = 00000000000000000000000000000011b = 0x00000003
```


HOCHSCHULE LUZERN

Example: Using Unions to Access Bit Patterns

- Try 2:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  union {
    float f;
    unsigned int u;
  } fu;
  fu.f = f;
  return fu.u;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
                                          convert2.c
```

This is what we want

```
get_bitpattern:
  movl    4(%esp), %eax
  ret
```

Output:

```
$ gcc -m32 -O3 -o convert2 convert2.c
$ ./convert2
3.1415927410 = 01000000010010010000111111011011b = 0x40490fdb
```

Calling Convention

HOCHSCHULE
LUZERN

Calling Convention on x86_64

- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer `%rsp`**
- **Register saving convention**
  - 6 “callee saved”
  - 2 “caller saved”
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

Calling Convention on x86_64

`%rax`	Caller saved / Return value
`%rbx`	Callee saved
`%rcx`	Caller saved / Argument #4
`%rdx`	Caller saved / Argument #3
`%rsi`	Caller saved / Argument #2
`%rdi`	Caller saved / Argument #1
`%rsp`	Stack pointer
`%rbp`	Callee saved
`%r8`	Caller saved / Argument #5
`%r9`	Caller saved / Argument #6
`%r10`	Caller saved / Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

Calling Convention on IA32

- **Arguments passed to functions via stack**
- **References to stack frame via base pointer `%ebp`**
- **Register saving convention**
  - 3 “callee saved”
  - 3 “caller saved”
  - 1 return value (also usable as caller saved)
  - 2 special (stack pointer)

HOCHSCHULE LUZERN

IA32/Linux+Windows Calling Convention

- **`%eax`, `%ecx`, `%edx`**
  - caller saved prior to call
    (if values are used later)
  - %eax used to return integer value
- **`%ebx`, `%esi`, `%edi`**
  - callee saved (if used)
- **`%esp`, `%ebp`**
  - used to manage the stack frames
  - must restore original values upon exit
    from procedure
    (= special form of callee saved)

`%eax`	Caller saved / Return value
`%ecx`	Caller saved
`%edx`	Caller saved
`%ebx`	Callee saved
`%esi`	Callee saved
`%edi`	Callee saved
`%esp`	Stack pointer
`%ebp`	Frame pointer

Implementation of Linux/x86_64 System Calls

- System calls are invoked via a special `syscall/sysenter` instruction
  - parameters passed through registers (not the stack)
    - `rax`: system call number
    - `rdi, rsi, rdx, rcx, r8, r9`: (up to) 6 arbitrary arguments; interpretation depends on the invoked system call.
      - the kernel uses a different set of registers: `rdi, rsi, rdx, r10, r8, r9`
    - result of systemcall placed in `rax`
      - >0: system call succeeded, result interpretation depends on syscall
      - <0: error

- libc provides convenient wrappers for most system calls

Implementation of Linux/x86_64 System Calls

- Directly invoking Linux/x86_64 System calls

```c
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  write(1, "hello, world\n", 13);
  exit(0);
}                              hello.c
```

```
$ gcc -o hello64 hello.c
$ ./hello64
hello, world
$
```

```c
int main(void)
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}                              myhello.c
```

```
.globl    my_write
.globl    my_exit

.section .text

# ssize_t my_write(int fd, const void *string,
#                  size_t count)
#
# parameters on function entry:
#   rdi: fd
#   rsi: buf
#   rdx: count

my_write:

  mov   $1, %rax       # rax: 1 (write syscall ID)
                       # rdi: fd
                       # rsi: buf
                       # rdx: count

  syscall              # invoke system call
                       # result in rax

  ret
…                                         mylib64.s
```

Implementation of Linux/x86_64 System Calls

- Directly invoking Linux/x86_64 System calls

```
int main(void)
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}                                    myhello.c
```

```
…

# void my_exit(int status)
#
# parameters on function entry:
#   rdi: status

my_exit:

  mov   $60, %eax       # rax: 60 (exit syscall ID)
                        # rdi: status

  syscall               # invoke system call

  # exit does not return, but just to be sure

  ret                                    mylib64.s
```

```
$ gcc -o myhello64 myhello.c mylib64.s
$ ./myhello64
hello, world
$
```

Implementation of Linux/IA32 System Calls

- Application programs use system calls to request a service from the kernel

- System calls are invoked via the `int 0x80` instruction
  - parameters passed through registers (not the stack)
    - `eax`: system call number
    - `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`: (up to) 6 arbitrary arguments (interpretation depends on the invoked system call)
    - `esp` cannot be used (overwritten when the CPU enters kernel mode)

- libc provides convenient wrappers for most system calls

Implementation of Linux/IA32 System Calls

- Directly invoking Linux/IA32 System calls

```
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  write(1, "hello, world\n", 13);
  exit(0);
}                              hello.c
```

```
$ gcc -m32 -o hello32 hello.c
$ ./hello32
hello, world
$
```

```
int main(void)
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}                              myhello.c
```

```
.globl my_write
.globl my_exit

.section .text

# int my_write(int fd, const void *buf,
                unsigned int count);
my_write:

  push  %ebp
  mov   %esp, %ebp
  push  %ebx

  mov   $4, %eax             # eax: 4 (write syscall)
  mov   8(%ebp), %ebx        # ebx: fd    (ebp+8)
  mov   12(%ebp), %ecx       # ecx: buf   (ebp+12)
  mov   16(%ebp), %edx       # edx: count (ebp+16)

  int   $0x80                # invoke system call
                             # result in eax
  pop   %ebx
  pop   %ebp

  ret

…                                        mylib32.s
```

Implementation of Linux/IA32 System Calls

- Directly invoking Linux/IA32 System calls

```
int main(void)
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}                              myhello.c
```

```
…

# void my_exit(int nr)
my_exit:

  push  %ebp
  mov   %esp, %ebp
  push  %ebx

  mov   $1, %eax              # eax: 1 (exit syscall)
  mov   8(%ebp), %ebx         # ebx: nr (ebp+8)

  int   $0x80                 # invoke system call

  pop   %ebx
  pop   %ebp

  ret                                     mylib32.s
```

```
$ gcc -m32 -o myhello32 myhello.c mylib32.s
$ ./myhello32
hello, world
$
```

Linux Kernel System Call Implementation

- Linux kernel system call table
  - in the Linux kernel source directory, see `arch/x86/entry/syscalls/`
    - `syscall_64.tbl` for 64-bit system calls
    - `syscall_32.tbl` for 32-bit system calls

- User-level SYSCALL invocation
  - glibc provides wrappers for each system call ("write" → SYSCALL in assembly)
  - source builds syscalls dynamically; easier to look at compiled libc.a

```
/tmp $ ar t /usr/lib/libc.a
/tmp $ ar x /usr/lib/libc.a write.o
/tmp $ objdump -d write.o
0000000000000000 <__libc_write>:
…
  6d:   b8 01 00 00 00          mov    $0x1,%eax
  72:   0f 0f                   syscall
…
```


HOCHSCHULE LUZERN

Linux Kernel System Call Implementation

- Kernel SYSCALL entry point
  - `ENTRY(entry_SYSCALL_64` in `arch/x86/entry/entry64.S`
  - depending on kernel version less or more complicated

- Kernel implementation of a system call
  - grep kernel tree for “`SYSCALL_DEFINE.\?(<syscall>,`”
  - for example, to find the write system call:

```
/usr/src/linux $ grep -rA3 "SYSCALL_DEFINE.\?(write," *
  SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *,
                  buf, size_t, count)
  {
    struct fd f = fdget_pos(fd);
```


HOCHSCHULE LUZERN

```
absdiff:
    cmpq    %rsi, %rdi
    jg      .L4
    movq    %rsi, %rax
    subq    %rdi, %rax
    jmp     .L7
.L4:
    movq    %rdi, %rax
    subq    %rsi, %rax
.L7:
    ret
```

Module Summary

Translation of C to Assembly

- **Control flow constructs**
  - No high-level control flow constructs at the assembly level
  - All control flow is altered using a combination of comparison and jump instructions
  - for → while → do while → assembly
  - (Larger) switch statements use efficient jump tables
- **Composite data structures**
  - No composite data structures at the assembly level
  - Translated by the compiler into offsets into a linear memory space
- **Calling conventions**
  - Caller vs callee-saved registers
  - Parameter passing
    - via the stack on IA32, in registers on x86_64

HOCHSCHULE LUZERN