IF2130 – Organisasi dan Arsitektur Komputer

sumber: Greg Kesden, CMU 15-213, 2012

Machine-Level Programming: Procedure (x86-64)

Achmad Imam Kistijantoro (imam@informatika.org)

Kusprasapta Mutijarsa (soni@stei.itb.ac.id)

Dicky Prima Satya (dicky@informatika.org)

Andreas Bara Timur (bara@informatika.org)

Today

- **Procedures (x86-64)**
- Arrays
    - One-dimensional
    - Multi-dimensional (nested)
    - Multi-level
- Structures
    - Allocation
    - Access

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

▸ Twice the number of registers

▸ Accessible as 8, 16, 32, 64 bits

x86-64 Integer Registers: Usage Conventions

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

x86-64 Registers

▸ **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**
  ▸ Eliminates need to update `%ebp/%rbp`

▸ **Other Registers**
  ▸ 6 callee saved
  ▸ 2 caller saved
  ▸ 1 return value (also usable as caller saved)
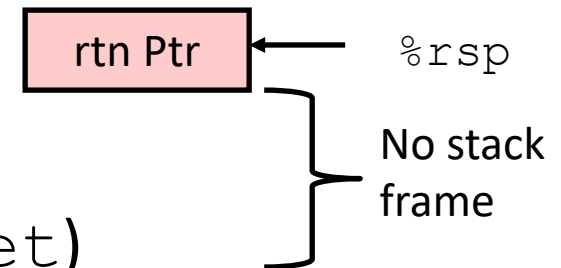 ▸ 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

▸ **Operands passed in registers**
  ▸ First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  ▸ 64-bit pointers
▸ **No stack operations required (except** `ret`**)**
▸ **Avoiding stack**
  ▸ Can hold all local information in registers

| rtn Ptr | ← %rsp
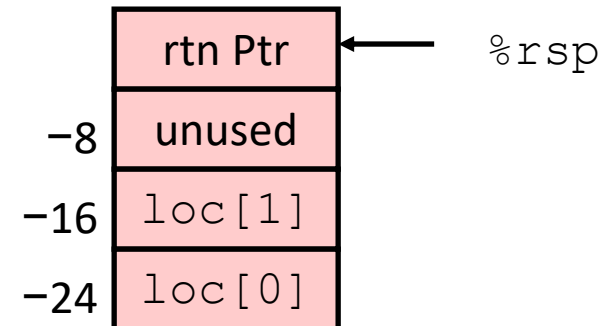
No stack frame

x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq  (%rdi), %rax
  movq  %rax, -24(%rsp)
  movq  (%rsi), %rax
  movq  %rax, -16(%rsp)
  movq  -16(%rsp), %rax
  movq  %rax, (%rdi)
  movq  -24(%rsp), %rax
  movq  %rax, (%rsi)
  ret
```

▸ **Avoiding Stack Pointer Change**
  ▸ Can hold all information within small window beyond stack pointer

rtn Ptr	← %rsp
−8 unused	
−16 loc[1]	
−24 loc[0]	

x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

▶ No values held while swap being invoked

▶ No callee save registers needed

▶ `rep` instruction inserted as no-op
  ▶ Based on recommendation from AMD

```
swap_ele:
    movslq %esi,%rsi              # Sign extend i
    leaq   8(%rdi,%rsi,8), %rax  # &a[i+1]
    leaq   (%rdi,%rsi,8), %rdi   # &a[i] (1st arg)
    movq   %rax, %rsi            # (2nd arg)
    call   swap
    rep                          # No-op
    ret
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

▸ Keeps values of `&a[i]` and `&a[i+1]` in callee save registers

▸ Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)         # Save %rbx
    movq    %rbp, -8(%rsp)          # Save %rbp
    subq    $16, %rsp               # Allocate stack frame
    movslq  %esi,%rax               # Extend i
    leaq    8(%rdi,%rax,8), %rbx    # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp     # &a[i]    (callee save)
    movq    %rbx, %rsi              # 2nd argument
    movq    %rbp, %rdi              # 1st argument
    call    swap
    movq    (%rbx), %rax            # Get a[i+1]
    imulq   (%rbp), %rax            # Multiply by a[i]
    addq    %rax, sum(%rip)         # Add to sum
    movq    (%rsp), %rbx            # Restore %rbx
    movq    8(%rsp), %rbp           # Restore %rbp
    addq    $16, %rsp               # Deallocate frame
    ret
```
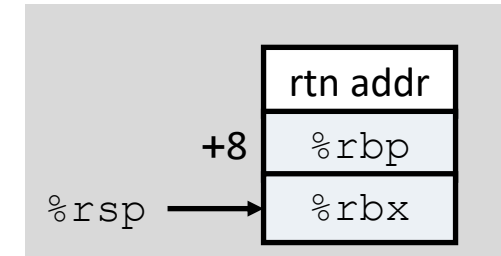
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)          # Save %rbx
movq    %rbp, -8(%rsp)           # Save %rbp
```
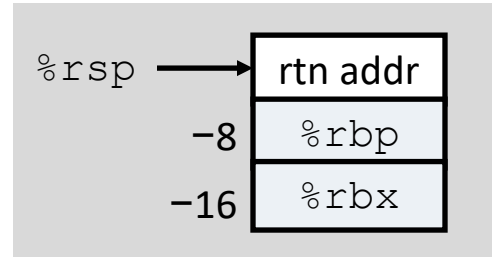
```
subq    $16, %rsp                # Allocate stack frame
```

● ● ●

```
movq    (%rsp), %rbx             # Restore %rbx
movq    8(%rsp), %rbp            # Restore %rbp
```

```
addq    $16, %rsp                # Deallocate frame
```

Interesting Features of Stack Frame

▶ Allocate entire frame at once

  ▶ All stack accesses can be relative to `%rsp`

  ▶ Do by decrementing stack pointer

  ▶ Can delay allocation, since safe to temporarily use red zone

▶ Simple deallocation

  ▶ Increment stack pointer

  ▶ No base/frame pointer needed

x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block

- Many tricky optimizations
  - What kind of stack frame to use
  - Various allocation techniques

Today

- Procedures (x86-64)
- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures

Basic Data Types

▶ Integral

  ▶ Stored & operated on in general (integer) registers
  ▶ Signed vs. unsigned depends on instructions used

Intel	ASM	Bytes	C
byte	**b**	1	**[unsigned] char**
word	**w**	2	**[unsigned] short**
double word	**l**	4	**[unsigned] int**
quad word	**q**	8	**[unsigned] long int** (x86-64)

▶ Floating Point

  ▶ Stored & operated on in floating point registers

Intel	ASM	Bytes	C
Single	**s**	4	**float**
Double	**l**	8	**double**
Extended	**t**	10/12/16	**long double**

▶

Array Allocation

▸ **Basic Principle**

T `A[`*L*`]`;

  ▸ Array of data type *T* and length *L*
  ▸ Contiguously allocated region of *L* * `sizeof`(*T*) bytes

`char string[12];`

x $x + 12$

`int val[5];`

x $x + 4$ $x + 8$ $x + 12$ $x + 16$ $x + 20$

`double a[3];`

x $x + 8$ $x + 16$ $x + 24$

`char *p[3];`   IA32

x $x + 4$ $x + 8$ $x + 12$

x86-64

x $x + 8$ $x + 16$ $x + 24$
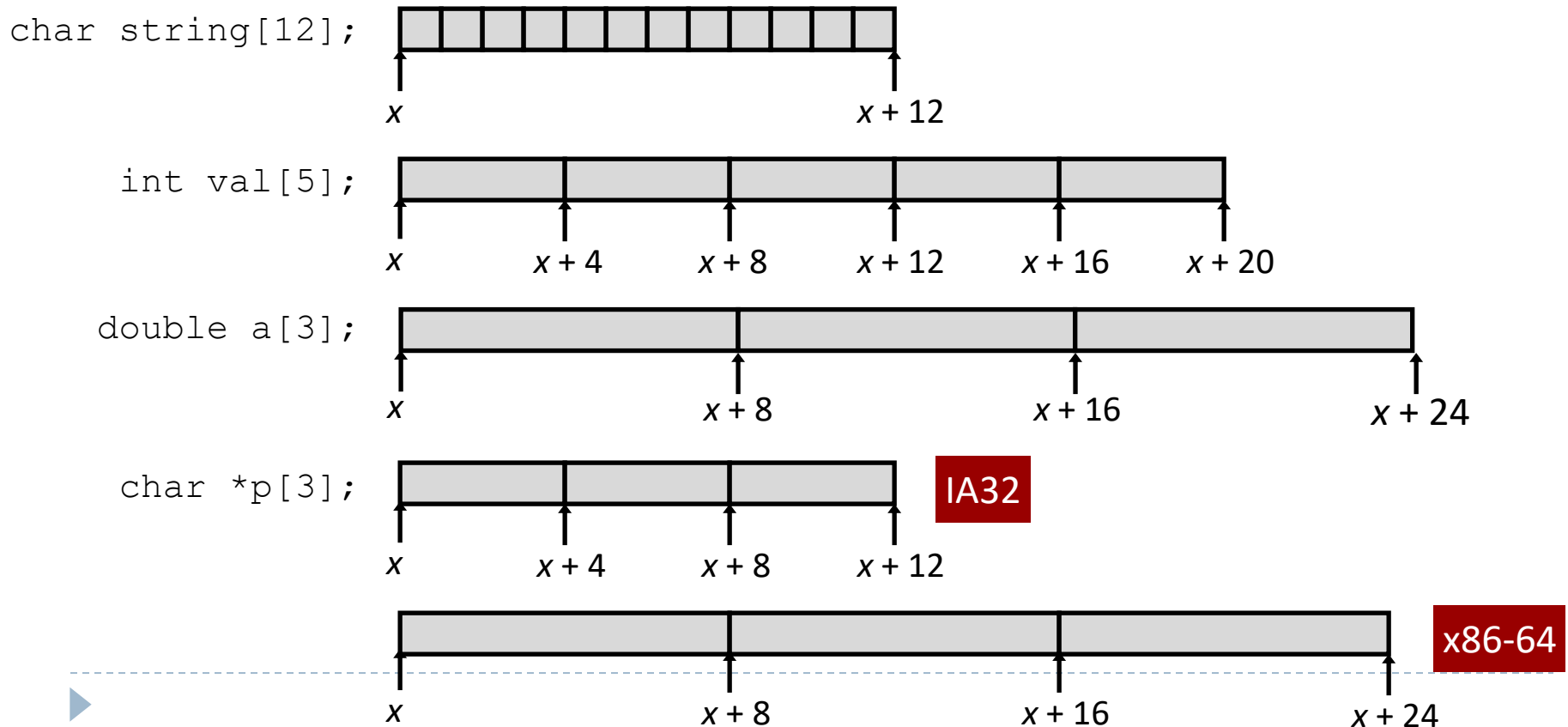
Array Access

▸ Basic Principle

T **A**[L];

  ▸ Array of data type T and length L

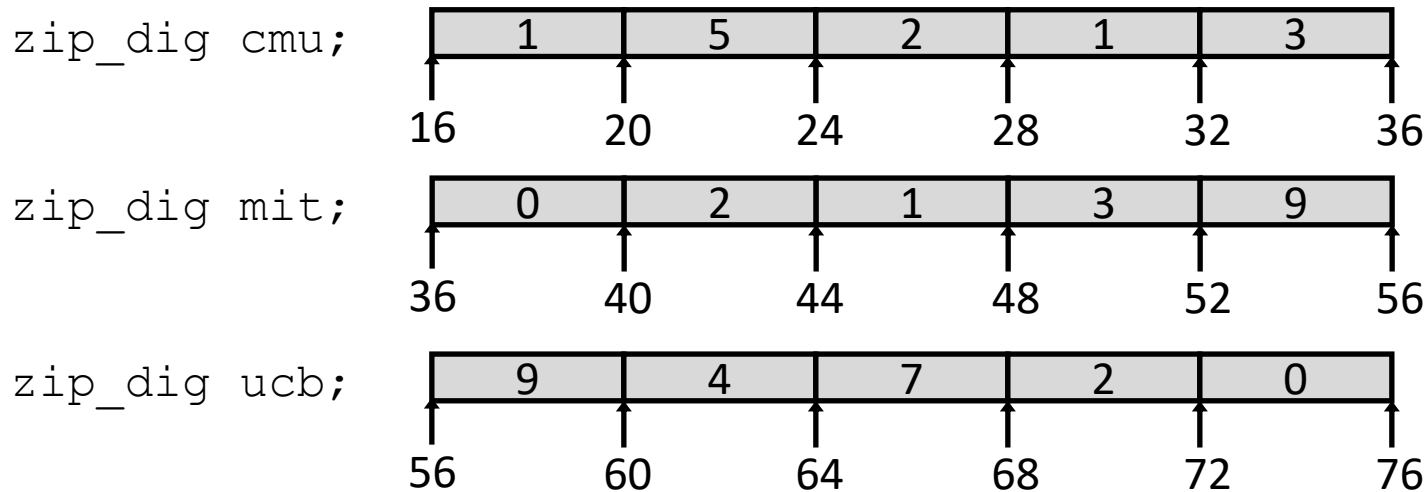 ▸ Identifier **A** can be used as a pointer to array element 0: Type T*

```
int val[5];
```

1	5	2	1	3

x $x + 4$ $x + 8$ $x + 12$ $x + 16$ $x + 20$
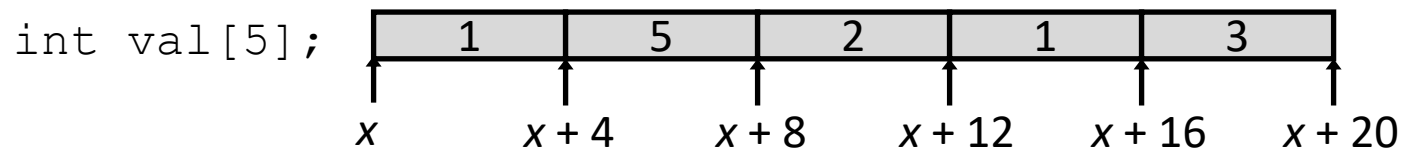
▸
| Reference	Type	Value
val[4]	**int**	3
val	**int ***	x
val+1	**int ***	$x + 4$
&val[2]	**int ***	$x + 8$
val[5]	**int**	??
*(val+1)	**int**	5
val + i	**int ***	$x + 4\,i$

▸

Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

`zip_dig cmu;`

1	5	2	1	3

16   20   24   28   32   36

`zip_dig mit;`

0	2	1	3	9

36   40   44   48   52   56

`zip_dig ucb;`

9	4	7	2	0

56   60   64   68   72   76

▸ Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"

▸ Example arrays were allocated in successive 20 byte blocks

▸   ▸ Not guaranteed to happen in general

Array Accessing Example

```
zip_dig cmu;
```

1	5	2	1	3

16   20   24   28   32   36
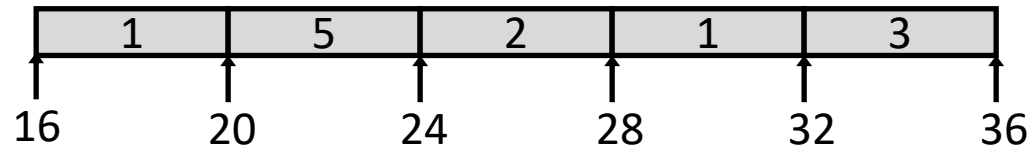
```
int get_digit
   (zip_dig z, int dig)
{
   return z[dig];
}
```

IA32

```
   # %edx = z
   # %eax = dig
    movl (%edx,%eax,4),%eax   # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

Array Loop Example (IA32)

```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # edx = z
  movl  $0, %eax              #    %eax = i
.L4:                          # loop:
  addl  $1, (%edx,%eax,4)  #    z[i]++
  addl  $1, %eax              #    i++
  cmpl  $5, %eax              #    i:5
  jne   .L4                   #    if !=, goto loop
```

Pointer Loop Example (IA32)

```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
    # edx = z = vz
    movl  $0, %eax           #   i = 0
.L8:                         # loop:
    addl  $1, (%edx,%eax)     #   Increment *(vz+i)
    addl  $4, %eax           #   i +=  4
    cmpl  $20, %eax          #   Compare i:20
    jne   .L8                #   if !=, goto loop
```

Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
   {{1, 5, 2, 0, 6},
    {1, 5, 2, 1, 3 },
    {1, 5, 2, 1, 7 },
    {1, 5, 2, 2, 1 }};
```

zip_dig
pgh[4];

| 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

76   96   116   136   156
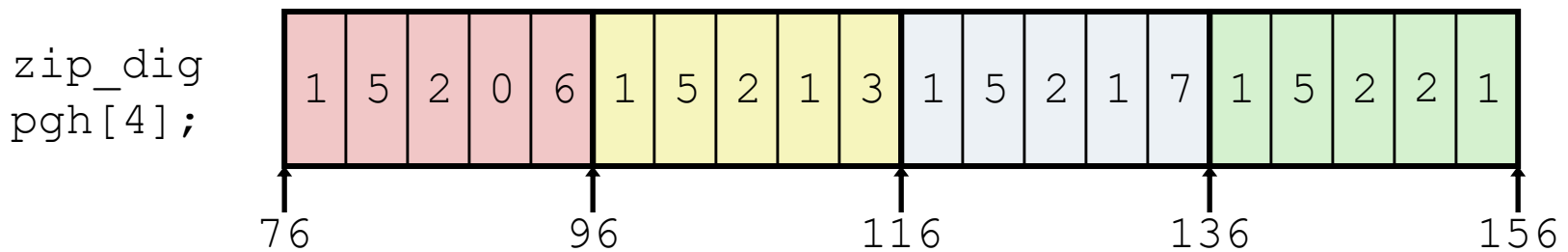
▸ "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  ▸ Variable **pgh**: array of 4 elements, allocated contiguously
  ▸ Each element is an array of 5 **int**'s, allocated contiguously
▸ "Row-Major" ordering of all elements guaranteed

Multidimensional (Nested) Arrays

- Declaration

  T `A`$[R][C]$;

  - 2D array of data type T
  - R rows, C columns
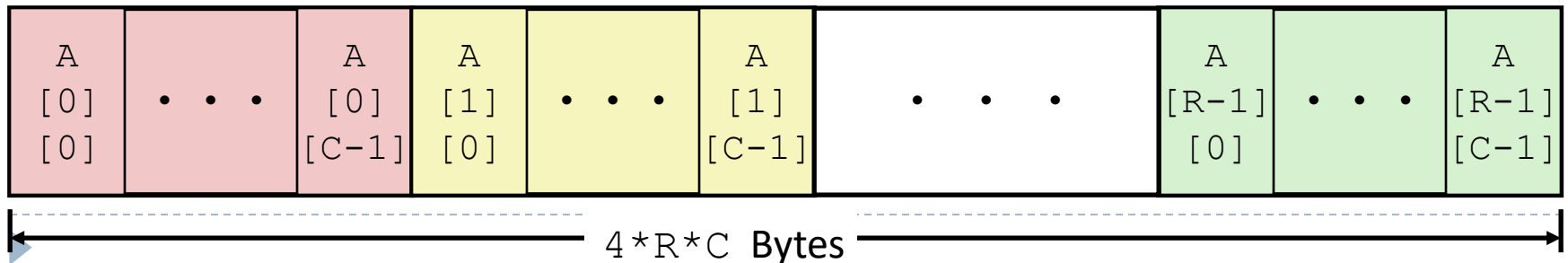 - Type T element requires K bytes

- Array Size

  - $R * C * K$ bytes

- Arrangement

  - Row-Major Ordering
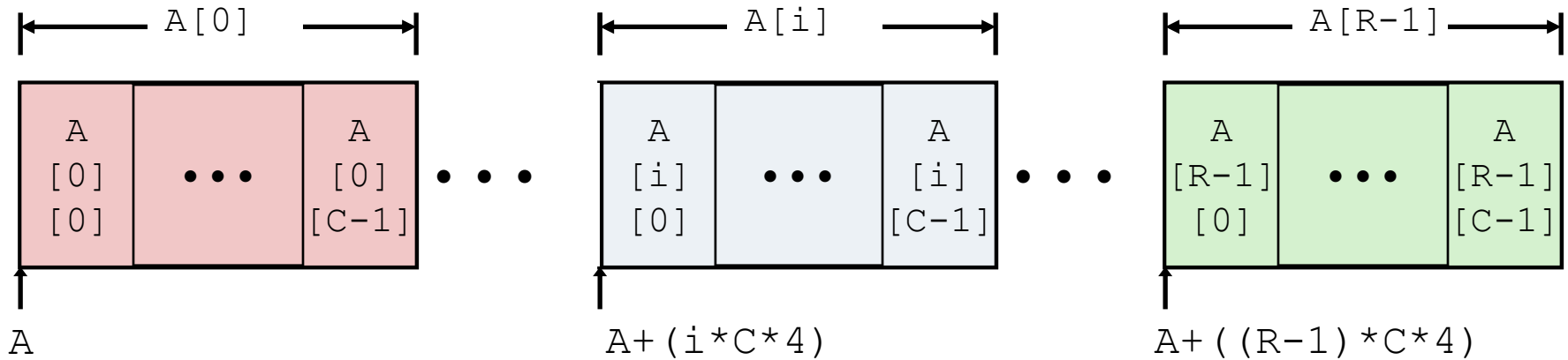
$$
\begin{bmatrix}
A[0][0] & \cdots & A[0][C-1] \\
& \vdots & \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{bmatrix}
$$

`int A[R][C];`

A[0][0]	...	A[0][C-1]	A[1][0]	...	A[1][C-1]	...	A[R-1][0]	...	A[R-1][C-1]

$\longleftarrow$ `4*R*C` Bytes $\longrightarrow$
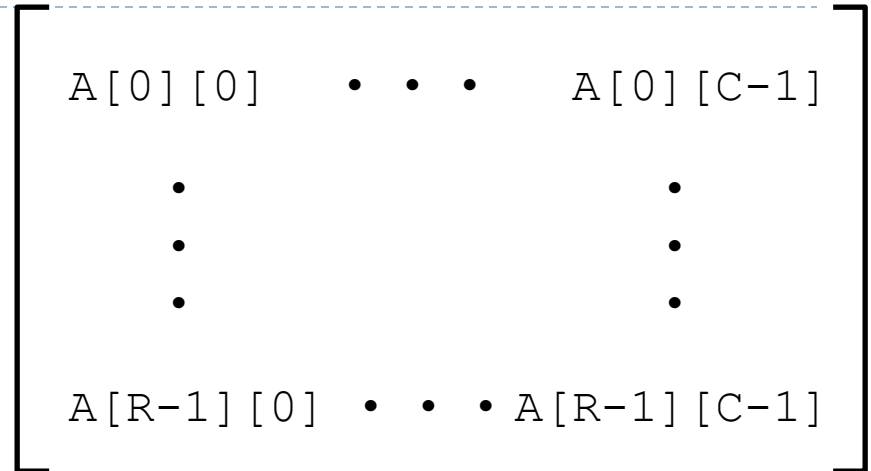
Nested Array Row Access

▶ Row Vectors
  ▶ **A[i]** is array of *C* elements
  ▶ Each element of type *T* requires *K* bytes
  ▶ Starting address **A** **+** *i* * (*C* * *K*)

```
int A[R][C];
```

Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
  # %eax = index
  leal (%eax,%eax,4),%eax  # 5 * index
  leal pgh(,%eax,4),%eax   # pgh + (20 * index)
```

- Row Vector
  - **pgh[index]** is array of 5 **int**'s
  - Starting address **pgh+(20*index)**
- IA32 Code
  - Computes and returns address
  - Compute as **pgh + 4*(index+4*index)**

Nested Array Row Access

▸ Array Elements

   ▸ **A[i][j]** is element of type *T,* which requires *K* bytes

   ▸ Address **A** + $i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```
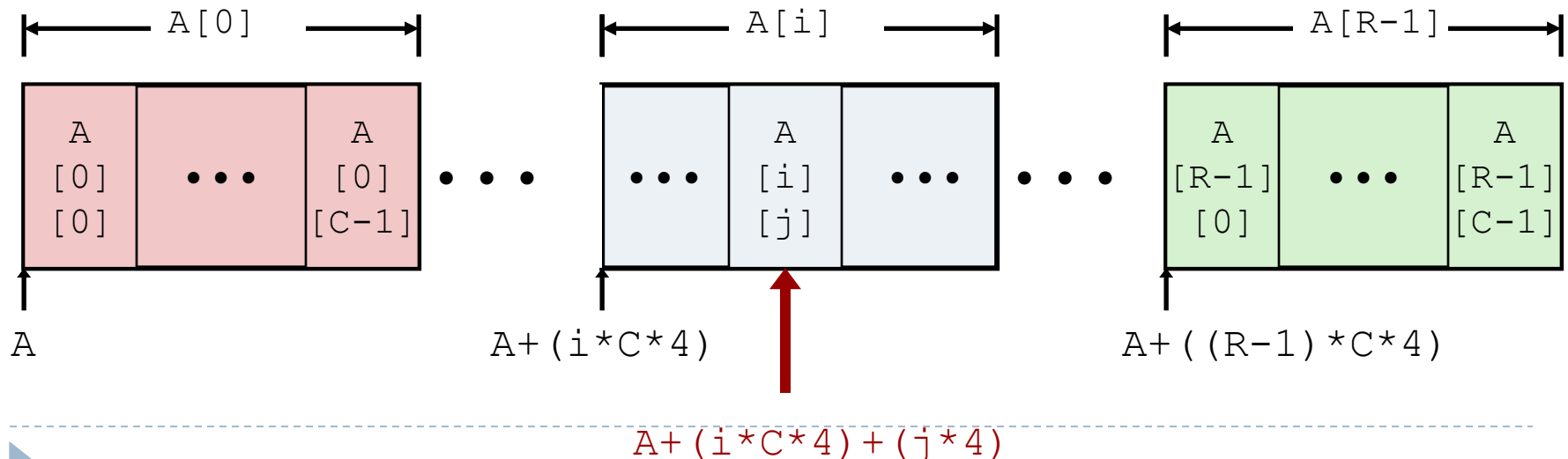
Nested Array Element Access Code

```
int get_pgh_digit
    (int index, int dig)
{
    return pgh[index][dig];
}
```

```
movl   8(%ebp), %eax          # index
leal   (%eax,%eax,4), %eax    # 5*index
addl   12(%ebp), %eax         # 5*index+dig
movl   pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

▶ Array Elements
  ▶ **pgh[index][dig]** is **int**
  ▶ Address: **pgh + 20*index + 4*dig**
    ▶ = **pgh + 4*(5*index + dig)**
▶ IA32 Code
  ▶ ▶ Computes address **pgh + 4*((index+4*index)+dig)**

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s
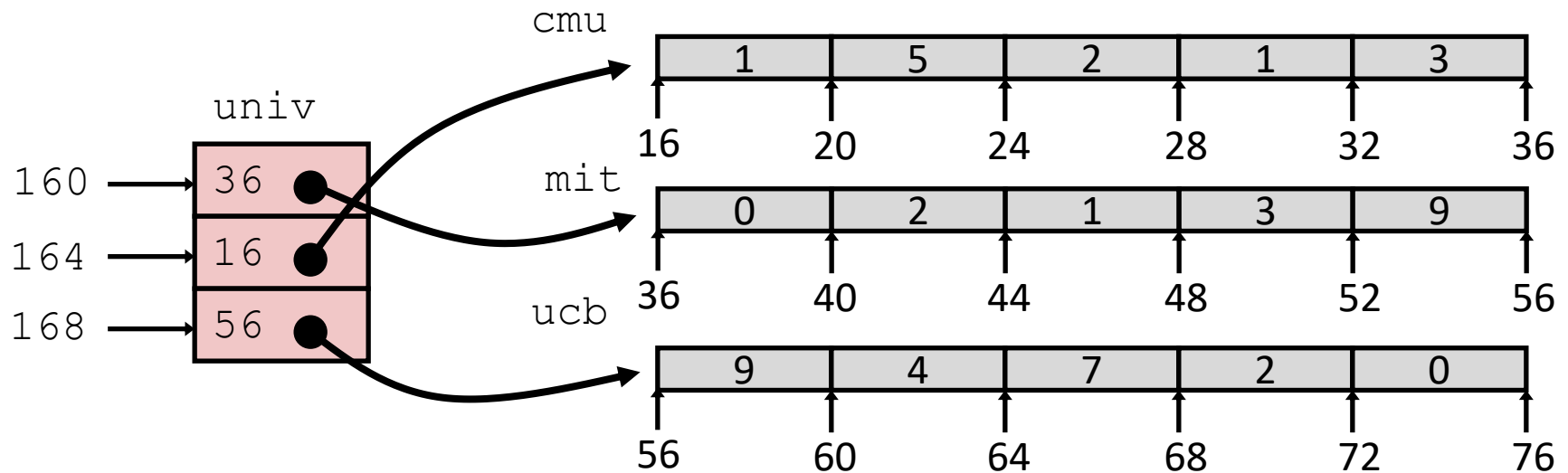
Element Access in Multi-Level Array

```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```

```
   movl   8(%ebp), %eax          # index
   movl   univ(,%eax,4), %edx     # p = univ[index]
   movl   12(%ebp), %eax          # dig
   movl   (%edx,%eax,4), %eax     # p[dig]
```
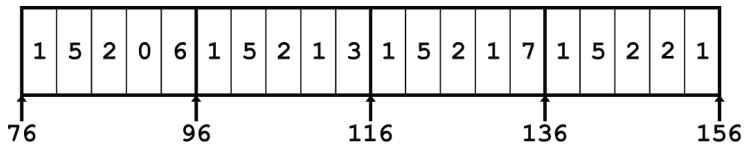
▸ Computation (IA32)
  ▸ Element access **Mem[Mem[univ+4*index]+4*dig]**
  ▸ Must do two memory reads
    ▸ First get pointer to row array
    ▸ Then access element within array
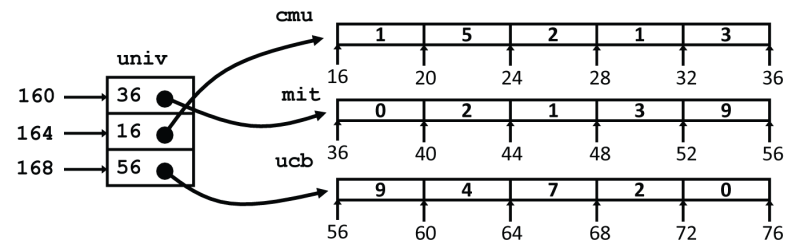
Array Element Accesses

Nested array

```
int get_pgh_digit
    (int index, int dig)
{
    return pgh[index][dig];
}
```

Multi-level array

```
int get_univ_digit
    (int index, int dig)
{
    return univ[index][dig];
}
```


Accesses looks similar in C, but addresses very different:

`Mem[pgh+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

N X N Matrix Code

- Fixed dimensions
  - Know value of N at compile time

- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays

- Variable dimensions, implicit indexing
  - Now supported by gcc

```c
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
   (fix_matrix a, int i, int j)
{
   return a[i][j];
}
```

```c
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
  (int n, int *a, int i, int j)
{
   return a[IDX(n,i,j)];
}
```

```c
/* Get element a[i][j] */
int var_ele
  (int n, int a[n][n], int i, int j)
{
   return a[i][j];
}
```

16 X 16 Matrix Access

- **Array Elements**
  - Address $A + i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
  movl   12(%ebp), %edx     # i
  sall   $6, %edx           # i*64
  movl   16(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   8(%ebp), %eax      # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*64)
```
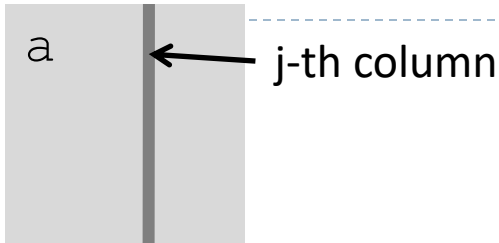
n X n Matrix Access

- **Array Elements**
  - Address $A + i * (C * K) + j * K$
  - C = n, K = 4

```c
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

Optimizing Fixed Array Access

a   ← j-th column

- ## Computation
  - Step through all elements in column j
- ## Optimization
  - Retrieving successive elements from single column

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */
void fix_column
   (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Optimizing Fixed Array Access

▸ Optimization

  ▸ Compute ajp = &a[i][j]

    ▸ Initially = a + 4*j

    ▸ Increment by 4*N

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
   (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

```
.L8:                               # loop:
  movl  (%ecx), %eax         #   Read *ajp
  movl  %eax, (%ebx,%edx,4) #   Save in dest[i]
  addl  $1, %edx             #   i++
  addl  $64, %ecx            #   ajp += 4*N
  cmpl  $16, %edx            #   i:N
  jne   .L8                  #   if !=, goto loop
```

Optimizing Variable Array Access

▸ Compute ajp = &a[i][j]

▸ Initially = a + 4*j

▸ Increment by 4*n

Register	Value
%ecx	**ajp**
%edi	**dest**
%edx	**i**
%ebx	**4*n**
%esi	**n**

```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

```
.L18:                          #  loop:
  movl  (%ecx), %eax           #    Read *ajp
  movl  %eax, (%edi,%edx,4) #    Save in dest[i]
  addl  $1, %edx               #    i++
  addl  $ebx, %ecx             #    ajp += 4*n
  cmpl  $edx, %esi             #    n:i
  jg    .L18                   #    if >, goto loop
```

Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
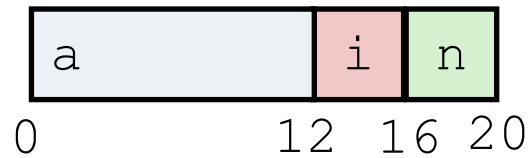 - Multi-level
- **Structures**
  - Allocation
  - Access

Structure Allocation

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

Memory Layout

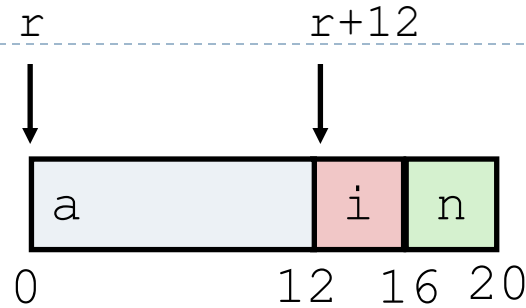| a | | i | n |

0   12   16 20

▶ Concept

  ▶ Contiguously-allocated region of memory

  ▶ Refer to members within structure by names

  ▶ Members may be of different types

Structure Access

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

r   r+12

| a | i | n |

0   12 16 20

▶ **Accessing Structure Member**

- ▶ Pointer indicates first byte of structure
- ▶ Access elements with offsets

```
void
set_i(struct rec *r,
      int val)
{
   r->i = val;
}
```
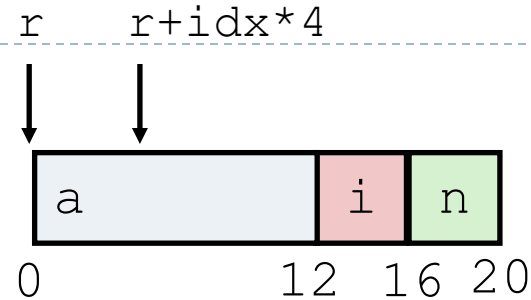
IA32 Assembly

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

Generating Pointer to Structure Member

r   r+idx*4

```
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
  - ▸ Mem[%ebp+8]: **r**
  - ▸ Mem[%ebp+12]: **idx**

```
int *get_ap
  (struct rec *r, int idx)
{
    return &r->a[idx];
}
```
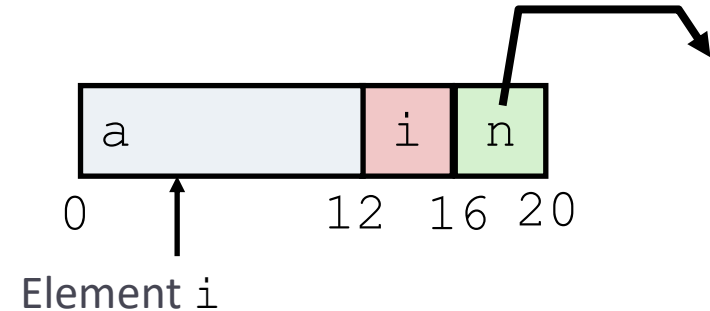
```
movl    12(%ebp), %eax   # Get idx
sall    $2, %eax         # idx*4
addl    8(%ebp), %eax    # r+idx*4
```

Following Linked List

▶ C Code

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

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

a | i | n

0   12 16 20

Element i

Register	Value
%edx	r
%ecx	val

```
.L17:                            # loop:
  movl  12(%edx), %eax       # r->i
  movl  %ecx, (%edx,%eax,4) # r->a[i] = val
  movl  16(%edx), %edx       # r = r->n
  testl %edx, %edx           # Test r
  jne   .L17                 # If != 0 goto loop
```

Summary

▶ Procedures in x86-64
  ▶ Stack frame is relative to stack pointer
  ▶ Parameters passed in registers

▶ Arrays
  ▶ One-dimensional
  ▶ Multi-dimensional (nested)
  ▶ Multi-level

▶ Structures
  ▶ Allocation
  ▶ Access

▶