Machine-level Programs – Data
Array Allocation

❖ Basic Principle

\[ T \, A[L]; \]
- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes in memory

```
char string[12];
```

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

```
double a[3];
```

```
char *p[3];
```

**Array Access**

❖ **Basic Principle**

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Identifier \( \textbf{A} \) can be used as a pointer to array element 0: Type \( T^* \)

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

❖ **Reference**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{val}[4] )</td>
<td>int</td>
<td>( 3 )</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>( \text{val}+1 )</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;( \text{val}[2] )</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>( * (\text{val}+1) )</td>
<td>int</td>
<td>( 5 )</td>
</tr>
<tr>
<td>( \text{val} + i )</td>
<td>int *</td>
<td>( x + 4 , i )</td>
</tr>
</tbody>
</table>
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig skku = { 1, 6, 4, 1, 9 };
zip_dig assm = { 0, 7, 2, 3, 3 };
zip_dig jeju = { 6, 3, 1, 2, 2 };

❖ Declaration “zip_dig skku” equivalent to “int skku[5]”
❖ Example arrays were allocated in successive 20 byte blocks
  ▪ Not guaranteed to happen in general
Array Accessing Example

zip_dig skku;

```
int get_digit
  (zip_dig z, int digit)
{
    return z[digit];
}
```

IA32
```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)
Array Loop Example

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

```assembly
# %rdi = z
    movl $0, %eax    # i = 0
    jmp .L3           # goto middle
.L4:
    addl $1, (%rdi,%rax,4) # z[i]++
    addq $1, %rax    # i++
.L3:
    cmpq $4, %rax    # i:4
    jbe .L4          # if <=, goto loop
    rep; ret
```
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
- $R \times C \times K$ bytes

❖ Arrangement
- Row-Major Ordering

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

```
\begin{array}{cccc}
 A[0][0] & \cdots & \cdots & A[0][C-1] \\
 \vdots & \ddots & \ddots & \vdots \\
 A[R-1][0] & \cdots & \cdots & A[R-1][C-1] \\
\end{array}
```
Nested Array Example

```
#define PCOUNT 4
zip_dig swn[PCOUNT] =
    {{1, 6, 4, 0, 8},
     {1, 6, 4, 1, 5 },
     {1, 6, 4, 1, 9 },
     {1, 6, 4, 2, 3 }};
```

- Variable `swn`: array of 4 elements, allocated contiguously
- Each element is an array of 5 `int`'s, allocated contiguously

  - Variable `swn`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously

❖ “Row-Major” ordering of all elements in memory
Nested Array Row Access

❖ **Row Vectors**

- \( A[i] \) is array of \( C \) elements
- Each element of type \( T \) requires \( K \) bytes
- Starting address \( A + i \cdot (C \cdot K) \)

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

![Diagram showing row vectors and their addresses](image)
Nested Array Row Access Code

❖ Row Vector
  ▪ \texttt{swn[index]} is array of 5 int’s
  ▪ Starting address \texttt{swn+20*index}

❖ Machine Code
  ▪ Computes and returns address
  ▪ Compute as \texttt{swn + 4*(index+4*index)}
**Nested Array Element Access**

- **Array Elements**
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address $A + i \ast (C \ast K) + j \ast K = A + (i \ast C + j) \ast K$

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

$A+(i\ast C\ast 4)$

$A+(i\ast C\ast 4)+(j\ast 4)$
Nested Array Element Access Code

Array Elements

- \( \text{swn}[\text{index}][\text{dig}] \) is int
- Address: \( \text{swn} + 20\times \text{index} + 4\times \text{dig} \)
  \[ = \text{swn} + 4 \times (5 \times \text{index} + \text{dig}) \]

```c
int get_swn_digit(int index, int dig) {
    return swn[index][dig];
}
```

```
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi          # 5*index+dig
movl swn(%rsi,4), %eax  # M[swn + 4*(5*index+dig)]
```
Multi-Level Array Example

- Variable `place` denotes an array of 3 elements.
- Each element is a pointer:
  - 8 bytes
- Each pointer points to an array of `int`'s.

```c
#define COUNT 3
int *place[COUNT] = { assm, skku, jeju; }
```

```c
zip_dig skku = { 1, 6, 4, 1, 9 };
zip_dig assm = { 0, 7, 2, 3, 3 };
zip_dig jeju = { 6, 3, 1, 2, 2 };;
```
Element Access in Multi-Level Array

```c
int get_place_digit( size_t index, size_t digit )
{
    return place[index][digit];
}
```

**Computation**

- Element access `Mem[Mem[place+8*index]+4*digit]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array
Array Element Accesses

Nested array

```c
int get_swn_digit(size_t index, size_t digit)
{
    return swn[index][digit];
}
```

Multi-level array

```c
int get_place_digit(size_t index, size_t digit)
{
    return place[index][digit];
}
```

Accesses looks similar in C, but address computations very different:

```
Mem[swn+20*index+4*digit]  Mem[Mem[place+8*index]+4*digit]
```
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

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

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

/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
    return a[i][j];
}
```
16 X 16 Matrix Access

- **Array elements**
  - Address $A + i \times (C \times K) + j \times K$
  - $C = 16$, $K = 4$ \hspace{1cm} (fixed dimension)

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

```
# a in %rdi, i in %rsi, j in %rdx
salq $6, %rsi  # 64*i
addq %rsi, %rdi # a + 64*i
movl (%rdi,%rdx,4), %eax # M[a + 64*i + 4*j]
ret
```
Array elements

- Address $A + i \times (C \times K) + j \times K$
- $C = n, K = 4$ (variable dimension)
- Must perform integer multiplication

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

```assembly
# n in %rdi, a in %rsi, i in %rdx, j in %rcx
imulq %rdx, %rdi # n*i
leaq (%rsi,%rdi,4), %rax # a + 4*n*i
movl (%rax,%rcx,4), %eax # a + 4*n*i + 4*j
ret
```
Structure Representation

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if different ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
Generating Pointer to Structure Member

```c
struct rec {  
  int a[4];  
  size_t i;  
  struct rec *next;  
};
```

❖ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as `r + 4*idx`

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

```assembly
# r in %rdi, idx in %rsi  
leaq (%rdi,%rsi,4), %rax  
ret
```
**C Code**

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

---

**Register Value**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
<tr>
<td>%rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

---

**Trace**

```
.L11:
    movslq 16(%rdi), %rax   # i = M[r+16]
    movl %esi, (%rdi,%rax,4) # M[r+4*i] = val
    movq 24(%rdi), %rdi     # r = M[r+24]
    testq %rdi, %rdi        # Test r
    jne .L11                # if !=0 goto loop
```
**Structures & Alignment**

- **Unaligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Alignment Principles

❖ Aligned Data
  ▪ Primitive data type requires $K$ bytes
  ▪ Address must be multiple of $K$
  ▪ Required on some machines; advised on x86-64

❖ Motivation for Aligning Data
  ▪ Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    ▪ Inefficient to load or store datum that spans quad word boundaries
    ▪ Virtual memory trickier when datum spans 2 pages

❖ Compiler
  ▪ Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (x86-64)

- **1 byte**: `char, ...`
  - no restrictions on address

- **2 bytes**: `short, ...`
  - lowest 1 bit of address must be 0₂

- **4 bytes**: `int, float, ...`
  - lowest 2 bits of address must be 00₂

- **8 bytes**: `double, long, char *, ...`
  - lowest 3 bits of address must be 000₂

- **16 bytes**: `long double` (GCC on Linux)
  - lowest 4 bits of address must be 0000₂
**Satisfying Alignment with Structures**

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K =$ Largest alignment of any element
  - Initial address & structure length must be multiples of $K$

- **Example:**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

<table>
<thead>
<tr>
<th>Byte Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td><code>c</code> - 3 bytes</td>
</tr>
<tr>
<td>p+4</td>
<td><code>i[0]</code></td>
</tr>
<tr>
<td>p+8</td>
<td><code>i[1]</code></td>
</tr>
<tr>
<td>p+16</td>
<td><code>v</code> - 8 bytes</td>
</tr>
<tr>
<td>p+24</td>
<td></td>
</tr>
</tbody>
</table>

- Multiples: 8 and 4

![Alignment Diagram](alignment_diagram.png)
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures

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

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

- **Compute array offset 12*idx**
  - `sizeof(S3)`, including alignment spacers

- **Element j is at offset 8 within structure**

- **Assembler gives offset a+8**
  - Resolved during linking (actual address of a[] is determined)

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

```asm
# %rdi = idx
leaq (%rdi,%rdi,2),%rax  # 3*idx
movzwl a+8(%rax,4),%eax
```
Saving Space

❖ **Put large data types first**

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

❖ **Effect: 4 bytes are saved**

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
</table>
Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u)
{
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u?  
Same as (unsigned) f?
Arrays in C
- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler optimizations
- Compiler often turns array code into pointer code
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures
- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Union
- Overlay all elements