1D Arrays
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
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 \( T* \)

Reference

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>( x + 4 \ i )</td>
</tr>
</tbody>
</table>
#define LEN 5

```c
int zip1[LEN] = { 1, 5, 2, 1, 3 };
int zip2[LEN] = { 0, 2, 1, 3, 9 };;
int zip3[LEN] = { 9, 4, 7, 2, 0 };
```

- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
int get_zip_digit ( int zip [LEN], int digit ) {
    return zip[digit];
}

# %rdi = z  <- it's an int pointer
# %esi = digit
movslq %esi, %rsi
movl (%rdi,%rsi,4), %eax
ret

- 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 incr( int zip[] ) {
    int i;
    for (i = 0; i < LEN; i++)
        zip[i]++;
}
```

```assembly
# %rdi is zip
movl $0, %eax
jmp .L3
.L4:
movslq %eax, %rdx
addl $1, (%rdi,%rdx,4) # z[i] ++
addl $1, %eax
.L3:
cmpl $4, %eax
jle .L4
rep ret
```

See p. 208 (Aside) for explanation of the rep instruction.
2D Arrays
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

\[
\begin{array}{c}
\text{int A[R][C];}
\end{array}
\]
Nested Array Example

Variable `zips`: 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 $A + i \times (C \times K)$

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

![Diagram showing nested array row access](image-url)
Nested Array Row Access Code

```
#define ROWS 4
#define COLS 5
int* get_zip ( int zips [], [COLS], int ind ) {
    return zips[ind];
}
```

```
#define ROWS 4
#define COLS 5

int* get_zip ( int zips [], [COLS], int ind ) {
    return zips[ind];
}
```

Row Vector
- `zips[ind]` is array of 5 int’s
- Starting address `zips+20*ind`

Machine Code
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`
Nested Array Element Access

Array Elements

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

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

```
+ (i*C*4)
```

```
+ ((R-1)*C*4)
```

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

```c
#define ROWS 4
#define COLS 5
int get_zips_digit ( int zips [],[COLS], int ind, int dig ) {
    return zips[ind][dig];
}
```

```assembly
movslq %esi, %rsi
leaq (%rsi,%rsi,4), %rax
salq $2, %rax
addq %rdi, %rax
movslq %edx, %rdx
movl (%rax,%rdx,4), %eax
```

**Array Elements**

- `zips[ind][dig]` is int
- Address: `zips + 20*ind + 4*dig`
  
  = `zips + 4*(5*index + dig)`
Structures
Structure Representation

- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - Even if another 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

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
Access to Structure Members

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

int * get_a (struct rec *r) {
    return r->a;
}
```

```
# r in %rdi
movq %rdi, %rax
ret
```
Access to Structure Members

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

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

```assembly
# r in %rdi
movslq %esi, %rsi
movl (%rdi,%rsi,4), %eax
ret
```
Access to Structure Members

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

int get_i (struct rec *r) {
    return r->i;
}

# r in %rdi
movl 16(%rdi), %eax
ret
Access to Structure Members

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

```
struct rec * get_next (struct rec *r ) {  
    return r->next;  
}
```

```
# r in %rdi
movq 24(%rdi), %rax
ret
```
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

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 $0_2$

- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be $00_2$

- **8 bytes: double, long, char *, ...**
  - lowest 3 bits of address must be $000_2$

- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be $0000_2$
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;
```
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;
```
Saving Space

- Put large data types first

```c
struct S1 {
    char c1;
    int n1;
    char c2;
    int n2;
};
```

```c
struct S2 {
    int n1;
    int n2;
    char c1;
    char c2;
};
```

- Effect (K=4)

S1 uses 16 bytes

```
c1  3 bytes  n1  c2  3 bytes  n2
```

S2 uses 12 bytes

```
n1  n2  c1  c2  2 bytes
```
Memory Layout
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()

- **Data**
  - Statically allocated data
  - E.g., global vars, static vars, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only
Memory Allocation Example

```c
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */
int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8);  /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
x86-64 Example Addresses

address range $\sim 2^{47}$

local
p1
p3
p4
p2
big_array
huge_array
main()
useless()

```plaintext
local
p1 0x0000000000007ffe4d3be87c
p3 0x000000007f7262a1e010
p4 0x000000007f7162a1d010
p2 0x00000000008359d120
big_array 0x00000000008359d010
huge_array 0x00000000000601060
main() 0x000000000040060c
useless() 0x00000000000400590
```