Array Allocation

- **Basic Principle**
  - \( T \ A[i] \):
    - Array of data type \( T \) and length \( l \)
    - Contiguously allocated region of \( l \times \text{sizeof}(T) \) bytes in memory

Array Access

- **Basic Principle**
  - \( T \ A[i] \):
    - Array of data type \( T \) and length \( l \)
    - Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

Array Example

```c
#define LEN 5
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

Array Accessing Example

```c
int get_zip_digit ( int zip [LEN], int digit ) {
    return zip[digit];
}
```

```
// Register %rdi contains starting address of array
// Register %rsi contains array index
// Desired digit at %rdi + 4*%rsi
movl %esi, %rsi
movl (%rdi, %rsi, 4), %eax
ret
```
Array Loop Example

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

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

![Array Layout](image)

Nested Array Example

```c
#define COUNT 4
int *zips[LEN][COUNT] = {
  {1, 5, 2, 0, 6},
  {1, 5, 2, 1, 3},
  {1, 5, 2, 1, 7},
  {1, 5, 2, 2, 1}
};
```

![Array Example](image)

Nested Array Row Access

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

![Row Access](image)

Nested Array Row Access Code

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

![Code Example](image)

2D Arrays

- 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

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

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

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

- **Array Elements**
  - zips[ind][dig] is int
  - Address: $zips + 20*ind + 4*dig$
    $= zips + 4*(5*index + dig)$

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

Access to Structure Members

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

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

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

```c
#define get_a_element ( struct rec *r, int idx ) {
    return r->a[idx];
}
```
Access to Structure Members

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

int get_1 (struct rec *r) {
    return r->l;
}
```

# r in %rdi
movl 16(%rdi), %eax
ret

Access to Structure Members

```c
struct rec {
    int a[4];
    size_t l;
    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

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

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

Specific Cases of Alignment (x86-64)

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

- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0

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

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

- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be 0000

Alignment Principles

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

- **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

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:**

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

```c
p=0  p+4  p+8  p+16  p+24
Multiple of 4  Multiple of 8  Multiple of 8
```
Meeting Overall Alignment Requirement

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>i[0]</th>
<th>i[1]</th>
<th>c</th>
<th>7 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiple of K=8
```

Saving Space

- Put large data types first

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

```
<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>n1</th>
<th>c2</th>
<th>n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Effect (K=4)

S1 uses 16 bytes

S2 uses 12 bytes

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 Layout

```
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 << 24); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```

x86-64 Example Addresses

```
local
p1
p3
p4
big_array
main()
useless()
```

Where does everything go?