Storage Classes and Memory Layout

**Stack**: objects created/destroyed in last-in, first-out order

**Heap**: objects created/destroyed in any order; automatic garbage collection optional

**Static**: objects allocated at compile time; persist throughout run
Stack-Allocated Objects

```c
int foo(int a, int b) {
    int c, d;
    bar(1, 2, 3);
}
```
Implementing Nested Functions with Access Links

```ocaml
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
(access link) (last frame ptr)

x = 5
s = 42
```

Implementing Nested Functions with Access Links

\[
\text{let } a \ x \ s = \\
\text{let } b \ y = \\
\quad \text{let } d \ w = w + s \text{ in} \\
\quad d (y+1) \text{ in } (* b *) \\
\text{let } e \ q = b (q+1) \text{ in} \\
\quad e (x+1) (* a *) 
\]

What does “a 5 42” give?

\begin{align*}
a: \quad & (\text{access link}) \\
& (\text{last frame ptr}) \\
& x = 5 \\
& s = 42 \\
e: \quad & (\text{access link}) \\
& (\text{last frame ptr}) \\
& q = 6
\end{align*}
Implementing Nested Functions with Access Links

```
let a =
  let b =
    let d = w + s in
    d (y+1) (* b *)
  in e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
(a) last frame ptr  x = 5
(b) last frame ptr  s = 42
```

```
(e) last frame ptr  q = 6
```

```
(a) last frame ptr  y = 7
```
let a x s =
  let b y =
    let d w = w + s in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)

What does “a 5 42” give?
In-Memory Layout Issues
Layout of Records and Unions

Modern processors have byte-addressable memory.

The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

16-bit integer: 1 0
32-bit integer: 3 2 1 0
Modern memory systems read data in 32-, 64-, or 128-bit chunks:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Reading an aligned 32-bit value is fast: a single operation.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

How about reading an unaligned value?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Padding

To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

• Each $n$-byte object must start on a multiple of $n$ bytes (no unaligned accesses).
• Any object containing an $n$-byte object must be of size $mn$ for some integer $m$ (aligned even when arrayed).

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    short y; /* 2 bytes */
    char w; /* 1 byte */
};
```

```c
struct padded {
    char a; /* 1 byte */
    short b; /* 2 bytes */
    short c; /* 2 bytes */
};
```
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each \( n \)-byte object must start on a multiple of \( n \) bytes (no unaligned accesses).
- Any object containing an \( n \)-byte object must be of size \( mn \) for some integer \( m \) (aligned even when arrayed).

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    char w; /* 1 byte */
    short y; /* 2 bytes */
};
```

```c
struct padded {
    char a; /* 1 byte */
    short b; /* 2 bytes */
    short c; /* 2 bytes */
};
```
Padding: (1) or (2)?

```c
struct padded {
    int a; /* 4 bytes */
    char b; /* 1 byte */
    char c; /* 1 byte */
};
```

(1)

(2)
A C union shares one space among all fields

```c
union int char {
    int i;    /* 4 bytes */
    char c;   /* 1 byte */
};

union twostructs {
    struct {
        char c;   /* 1 byte */
        int i;    /* 4 bytes */
    } a;
    struct {
        short s1; /* 2 bytes */
        short s2; /* 2 bytes */
    } b;
};
```

or

```
s1 s1 s2 s2 c
i i i i i
```
Arrays

Basic policy in C: an array is just one object after another in memory.

```c
int a[10];
```

What if we remove rule 2 of padding?

```c
struct {
    int a;
    char c;
} b[2];
```
The largest primitive type dictates the alignment

```c
struct {
    short a;
    short b;
    char c;
} d[4];
```
The largest primitive type dictates the alignment

```c
struct {
    short a;
    short b;
    char c;
} d[4];
```
Arrays of Arrays

```c
char a[4];
```

```c
char a[3][4];
```

```
```
The Heap
A heap is a region of memory where blocks can be dynamically allocated and deallocated in any order.
```c
struct point {
    int x, y;
};

int play_with_points(int n)
{
    int i;
    struct point *points;

    points = malloc(n * sizeof(struct point));

    for (i = 0; i < n; i++) {
        points[i].x = random();
        points[i].y = random();
    }

    /* do something with the array */

    free(points);
}
```
Dynamic Storage Allocation

free() \[\rightarrow\]
\%one.osf/seven.osf\]
Dynamic Storage Allocation

↓ free([ ])

↓ malloc([ ]/one.osf/seven.osf)
Dynamic Storage Allocation

\[
\text{free(\quad)}
\]

\[
\text{malloc(\quad)}
\]
Dynamic Storage Allocation

\[
\downarrow \text{free(□)}
\]

\[
\downarrow \text{malloc(□)}
\]
Dynamic Storage Allocation

Rules:

Each allocated block contiguous
Blocks stay fixed once allocated

malloc()
free()
Simple Dynamic Storage Allocation

Maintaining information about free memory
   Simplest: Linked list
The algorithm for locating a suitable block
   Simplest: First-fit
The algorithm for freeing an allocated block
   Simplest: Coalesce adjacent free blocks
Simple Dynamic Storage Allocation

malloc()

free()
Simple Dynamic Storage Allocation

malloc( )
Simple Dynamic Storage Allocation

malloc()

free()
Simple Dynamic Storage Allocation

malloc()

free()
Simple Dynamic Storage Allocation

malloc( )

free( )
Fragmentation

malloc( ) seven times give

free() four times gives

malloc( ) ?

Need more memory; can’t use fragmented memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection.
Always reference memory through “handles.”

The original Macintosh did this to save memory.
Fragmentation and Handles

Standard CS solution: Add another layer of indirection.
Always reference memory through “handles.”

The original Macintosh did this to save memory.
Automatic Garbage Collection
Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

Advantages?  Disadvantages?
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```ocaml
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

• Maintain count of references to each object
• Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Circular structures defy reference counting?

![Diagram of a and b with arrows connecting them](/two.osf/five.osf)
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```ocaml
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

• Stop-the-world algorithm invoked when memory full
• Breadth-first-search marks all reachable memory
• All unmarked items freed

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```plaintext
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```
Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky