EPITA - Practical Programming

Laboratory of Epita

03 - Structures

Data Structures

Data Structures

A good program is first **good data structures** and then, *eventually good code*.

Struct

struct

- C struct are basic block of data
- Fields accesses are resolved at compile-time
- Layout and size are as important as field names

LSE Security System
Laboratory of Epita

Anonymous Struct

```c
#include <stdio.h>
int main() {
  // No need for type definitions
  struct { int a,b; float c,d; } s =
    { 42, 0, 3.14, 1.414 };
  printf("a = %d\n", s.a);
  printf("b = %d\n", s.b);
  printf("c = %g\n", s.c);
  printf("d = %g\n", s.d);
  return 0;
}
```

LSE Security System Laboratory of Epita

Named Struct

```c
// Fake struct definition
struct my_struct {
  int              a,b;
  float            c,d;
};

// struct def is now known
float f(struct my_struct s) {
  return s.a + s.b + s.c + s.d;
}
```

Struct As Argument Or Result

- A struct is a block of data
- When passed without pointer, struct are copied
- When returned without pointer, they are copied

```
struct my_struct build() {
  struct my_struct s =
    { 42, 0, 3.14, 1.414 };
  return s;
}
```

```
float g() {
  struct my_struct s1, s2;
  s1 = build();
  s2 = s1;
  return f(s2);
}
```

Struct Through a Pointer

```c
void display(struct my_struct *s) {
  printf("{\n");
  printf("  a = %d;\n", s->a);
  printf("  b = %d;\n", s->b);
  printf("  c = %g;\n", s->c);
  printf("  d = %g;\n", s->d);
  printf("}\n");
}
```

Struct Layout

- Fields may be aligned, leaving gaps in the struct
- Alignment depends on data size and word size
- **Basic rule: a field start on an address multiple of its size.**

LSE Security System
Laboratory of Epita

Struct Layout

```c
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

struct demo
{
  uint8_t  f1;
  uint64_t f8;
  uint32_t f4;
};

int main()
{
  printf("sizeof (struct demo) = %zu\n", sizeof (struct demo));
  printf("Layout:\n");
  printf("  f1: %zu\n", offsetof(struct demo, f1));
  printf("  f8: %zu\n", offsetof(struct demo, f8));
  printf("  f4: %zu\n", offsetof(struct demo, f4));
  return 0;
}
```

Struct Layout

In 64bit:

```
sizeof (struct demo) = 24
Layout:
                f1:         0
                f8:         8
  f4: 16
```

In 32bit:

```
sizeof (struct demo) = 16
Layout:
                f1:         0
                f8:         4
  f4: 12
```

Arrays In Struct

```c
struct s_user {
  unsigned          uid, gid;
  char              login[16];
};

int main() {
  struct s_user     u;
  printf("u.login:\t%p\n",u.login);
  printf("&(u.login):\t%p\n",&(u.login));
  return 0;
}
```

Arrays In Struct

```
u.login:            0x7fffed0045b0
&(u.login):         0x7fffed0045b0
```

Recursive Structure

```
struct list {
  struct list          *next;
  int                   value;
};
```

Lists

Linked Lists

- Classical data structure
- Easy to implement
- Heavily use pointers
- Base structure for queues and stacks

LSE Security System Laboratory of Epita

Lists

```
struct list {
  struct list          *next;
  int                   value;
};

struct list *empty_list() { return NULL; }

int list_is_empty(struct list *l) {
  return l == NULL;
}
```

LSE Security System Laboratory of Epita

Adding Elements

```c
struct list *add(struct list *l, int x) {
  struct list           *tmp;
   // Size DOES matter !
  tmp = malloc(sizeof (struct list));
  tmp->value = x;
  tmp->next = l;
  return tmp;
}
```

Altering The Head

```c
// Note the double * on l
void addin(struct list **l, int x) {
  struct list          *tmp;
  tmp = malloc(sizeof (struct list));
  tmp->value = x;
  tmp->next = *l;
  *l = tmp;
}
```

Classical Loop On List

```c
for (; l; l = l->next) {
    // work on element
}
```

List Length

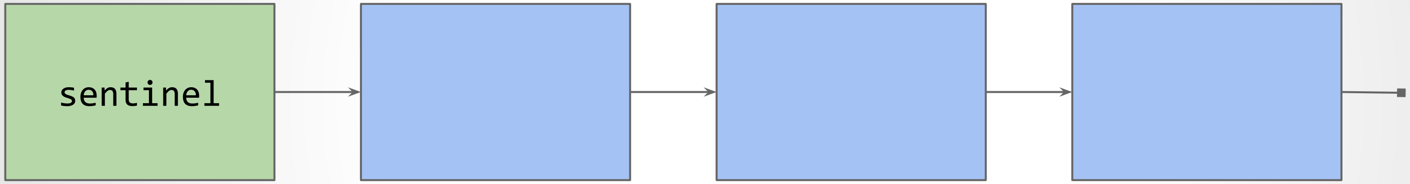
```c
size_t list_len(struct list *l) {
  size_t                    len;
  for (len=0; l; l = l->next)
    len += 1;
  return len;
}
```

Sentinel

Goal: remove corner cases and double pointers

➢ add a fake node at the beginning of the list

Sentinel



Example

```
void list_remove(struct list **list, struct list *elm)
{
  struct list *cur = *list;
  struct list *prev = NULL;
  for (; cur != elm; cur = cur->next)
    prev = cur;
  if (prev == NULL)
    *list = cur->next;
  else
    prev->next = cur->next;
}
```

```
void list_remove(struct list **list, struct list *elm)
{
  if (*list == elm) {
    *list = elm->next;
    return;
  }
  struct list *cur = *list;
  for (; cur->next != elm; cur = cur->next)
    continue;
  cur->next = elm->next;
}
```

Sentinel - example

```c
void list_remove(struct list **list, struct list *elm)
{
  struct list *cur = *list;
  struct list *prev = NULL;
  for (; cur != elm; cur = cur->next)
    prev = cur;
  if (prev == NULL)
    *list = cur->next;
  else
    prev->next = cur->next;
}
```

```c
void list_remove(struct list *list, struct list *elm)
{
  for (; list->next != elm; list = list->next)
    continue;
  list->next = elm->next;
}
```

LSE Security System Laboratory of Epita

Another solution

```c
void list_remove(struct list **list, struct list *elm)
{
  struct list **ref = list;
  for (; *ref != elm; ref = &((*ref)->next))
    continue;
  *ref = elm->next;
}
```

Security System
Laboratory of Epita

Queue

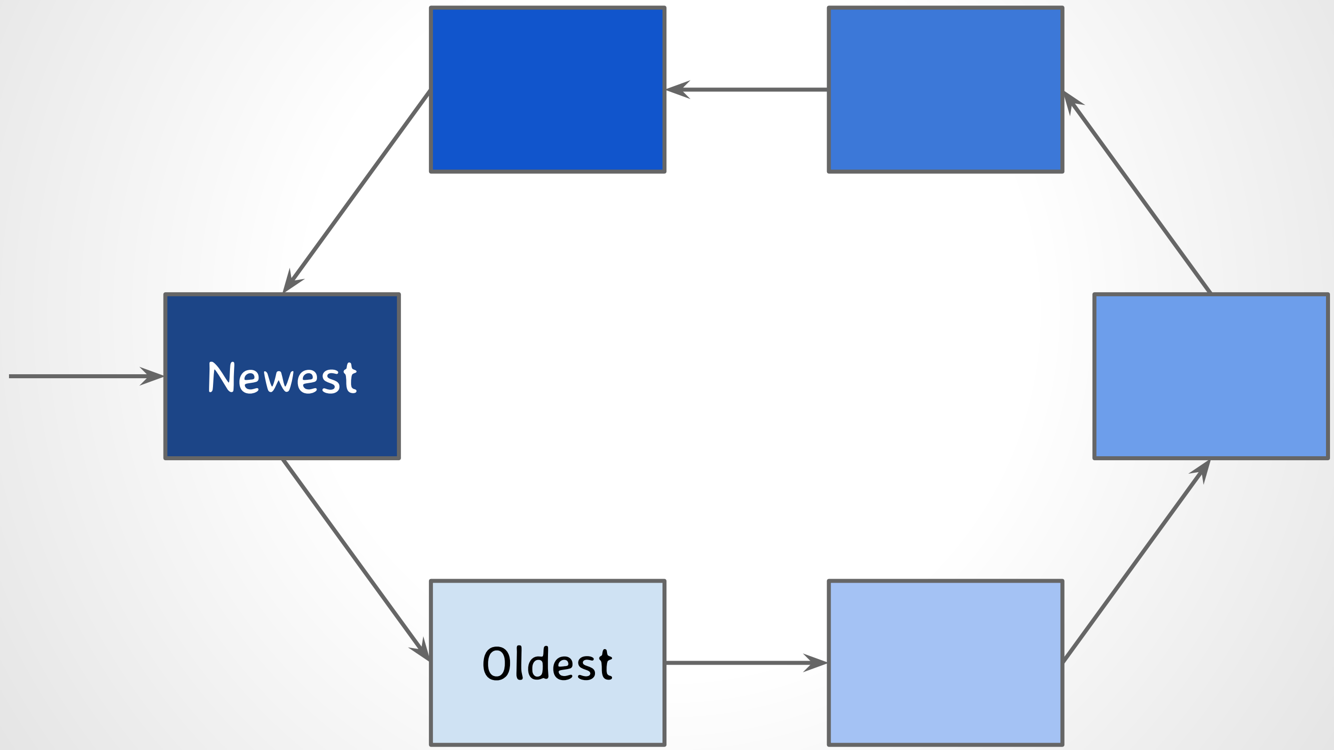
Classic Queue

```c
struct queue {
  struct queue *next;
  void         *value;
};

struct queue* queue_empty(void) { return NULL; }

int queue_is_empty(struct queue *q) { return q == NULL; }
```

Circular Lists as Queue



Queue Push

```
struct queue* queue_push(struct queue *q, void *x) {
  struct queue *tmp;
  tmp = malloc(sizeof (struct queue));
  tmp->value = x;
  if (q) {
    tmp->next = q->next;
    q->next = tmp;
  } else {
    tmp->next = tmp;
  }
  return tmp;
}
```

Queue Pop

```
void* queue_pop(struct queue **q) {
  struct queue *tmp = (*q)->next;
  void         *x   = tmp->value;
  if (tmp == tmp->next)
    *q = NULL;
  else
    (*q)->next = tmp->next;
  free(tmp);
  return x;
}
```

LSE Security System Laboratory of Epita

Trees

LSE Security System

Laboratory of Epita

Binary Trees

```c
struct tree {
  // Children
  struct tree           *left, *right;
  // Content
  int                    key;
};
```

LSE Security System
Laboratory of Epita

Binary Tree

```c
// Compute size
size_t tree_size(struct tree *t) {
  if (t == NULL)
    return 0;
  return 1 + tree_size(t->left) + tree_size(t->right);
}

// Compute height
static inline int max(int a, int b) { return a > b ? a : b; }

int tree_height(struct tree *t) {
  if (t == NULL)
    return -1;
  return 1 + max(tree_height(t->left), tree_height(t->right));
}
```

Binary Tree

```
// Prefix print
void prefix_print(struct tree *t) {
  if (t) {
    printf("%d; ", t->key);
    prefix_print(t->left);
    prefix_print(t->right);
  }
}
```

LSE Security System
Laboratory of Epita

Binary Tree

```c
// Breadth first print
void breadth_print(struct tree *t) {
  if (t) {
    struct queue       *q = queue_empty();
    q = queue_push(q, t);
    q = queue_push(q, NULL);
    do {
      t = queue_pop(&q);
      if (t == NULL) {
        printf("\n");
        if (!queue_is_empty(q))
          q = queue_push(q, NULL);
      } else {
        printf("%d ", t->key);
        if (t->left)
          q = queue_push(q, t->left);
        if (t->right)
          q = queue_push(q, t->right);
      }
    } while (!queue_is_empty(q));
  }
}
```