Data Structures in C

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

October 8, 2020

Outline

- Structures and unions
- Dynamic memory allocation

Structures

Structures

```c
struct date {
    int year;
    int month;
    int day;
};
void print_date(const struct date * d);
int main() {
    struct date moon_landing;
    moon_landing.year = 1969;
    moon_landing.month = 7;
    moon_landing.day = 20;
    print_date(&moon_landing);
}
void print_date(const struct date * d) {
    printf("%d/%d/%d\n", d->day, d->month, d->year);
}
```

Nested Structures

Nested Structures

```
struct person {
    const char * name;
    struct date birthdate;
    struct person * mother;
    struct person * father;
};
void print_person(const struct person * p) {
    printf("Name: %s\n", p->name);
    printf("Birthdate: ");
    print_date(&(p->birthdate));
    printf("Mother's Name: %s\n", p->mother->name);
    printf("Father's Name: %s\n", p->mother->name);
}
```

Initializers for Structs and Arrays

Initializers for Structs and Arrays

```c
struct Person {
    int age;
    char * name;
    struct Person * father;
    struct Person * mother;
};

struct Person p = { 18, "Antonio", NULL, NULL };

int moon_landing[3] = { 20, 7, 1969 };

const char * months[] = {
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" };
```

Enums and Unions

Enums and Unions

```
enum Types { INTEGER, FLOAT, STR };

union Value {
    int int_value;
    float float_value;
    char * str_value;
};
enum Type read_value(union Value * v);

int main(int argc, char *argv[]) {
    union Value v;
    switch(read_value(&v)) {
    case INTEGER: printf("v=%d\n", v.int_value); break;
    case FLOAT: printf("v=%f\n", v.float_value); break;
    case STR: printf("v=%s\n", v.str_value); break;
    }
}
```

More on Unions

You must always read the last element you write

More on Unions

You must always read the last element you write

```c
union Value {
    int int_value;
    float float_value;
    char * str_value;
};

int main(int argc, char *argv[]) {
    union Value v;
    v.str_value = "ciao";
    v.int_value = 100;
    printf("v = %s\n", v.str_value); /* undefined behavior! */
}
```

More on Unions

- The point is to share memory space to store mutually-exclusive values

More on Unions

- The point is to share memory space to store mutually-exclusive values

```
union Value {
    int int_value;
    float float_value;
    char * str_value;
};
struct MValue {
    int int_value;
    float float_value;
    char * str_value;
};
printf("union: %d bytes, struct: %d bytes\n",
       sizeof(union Value),sizeof(struct MValue));
```

Dynamic Memory Allocation

- The standard C library provides functions for the allocation and deallocation of memory
  - crucially important feature
  - basic concepts and functions are very simple
  - correct use requires complete comprehension and attention to details

Dynamic Memory Allocation

- The standard C library provides functions for the allocation and deallocation of memory
  - crucially important feature
  - basic concepts and functions are very simple
  - correct use requires complete comprehension and attention to details

- Use `malloc` to allocate memory

```c
char * copy_string(const char * s) {
    char * c, res;
    if (res = malloc(strlen(s) + 1))
        for (c = res; (*c = *s) != 0; ++s, ++c);
    return res;
}
```

The `sizeof` Operator

- Memory allocation functions take a *size* parameter
  - size in bytes
- The `sizeof` operators tells the size of a given type

```
struct Person {
    int age;
    char * name;
    struct Person * father;
    struct Person * mother;
};

struct Person * reproduction(struct Person * mom, struct Person * dad) {
    struct Person * child = malloc(sizeof(struct Person));
    if (child != 0) {
        child->age = 0; child->name = choose_name();
        child->mother = mom; child->father = dad;
    }
    return child;
}
```

Dynamic Memory Allocation

- Use `free` to deallocate memory

```
char * x = copy_string("ciao!");
if (x != NULL) {
    /* ... */;
    free(x);
 } else {
     printf("no more memory!\n");
  }
```

Dynamic Memory Allocation

- Use free to deallocate memory

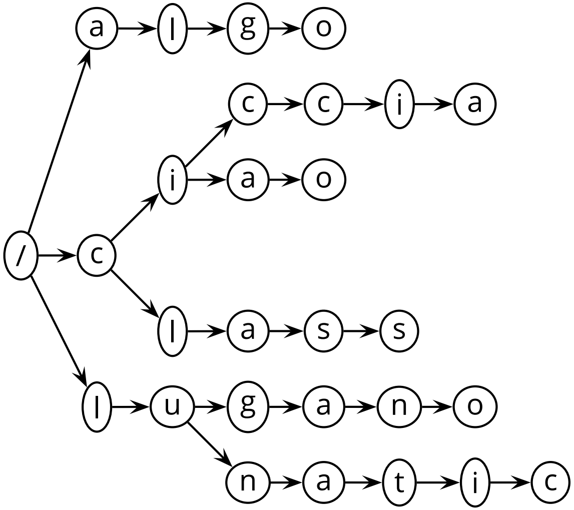
```
char * x = copy_string("ciao!");
if (x != NULL) {
    /* ... */;
    free(x);
 } else {
     printf("no more memory!\n");
  }
```

- A pointer value is no longer valid after the use of free

```
char * x = copy_string("ciao!");
free(x);
printf("%s", x); /* use of invalid pointer! */
```

Exercise

- Implement a “radix-256” tree to represent a set of strings



Exercise

- Every element of the data structure has 256 pointers to the next characters
- Implement the radix tree according to this API definition

```
#ifndef RADIX256_TREE_H_INCLUDED
#define RADIX256_TREE_H_INCLUDED

struct radix256_tree;

struct radix256_tree * radix256_tree_new ();       /* constructor */
void radix256_tree_delete (struct radix256_tree *); /* destructor */

/* insert the string defined by begin and end in tree t */
int radix256_tree_add (struct radix256_tree * t,
                       const char * begin, const char * end);

/* return true iff the string defined by begin and end is in tree t */
int radix256_tree_find (struct radix256_tree * t,
                        const char * begin, const char * end);
#endif
```

Exercise (2)

- Command-line parameter: name of a file containing a list of entries representing persons (you may assume less than 1000 entries)

Exercise (2)

- Command-line parameter: name of a file containing a list of entries representing persons (you may assume less than 1000 entries)
  - format: one entry per line:

    ```
    16/12/1969,Mario Rossi,Alberto Rossi,Diana Bianchi
    ```

Exercise (2)

- Command-line parameter: name of a file containing a list of entries representing persons (you may assume less than 1000 entries)
  - format: one entry per line:

    ```
    16/12/1969,Mario Rossi,Alberto Rossi,Diana Bianchi
    ```

- Input: a set of names, one per line

Exercise (2)

- Command-line parameter: name of a file containing a list of entries representing persons (you may assume less than 1000 entries)
  - format: one entry per line:

    ```
    16/12/1969,Mario Rossi,Alberto Rossi,Diana Bianchi
    ```

- Input: a set of names, one per line

- Output: if the name was found in the list, output the person's year of birth and all the names of his/her ancestors
  - first-level ancestors should be identified as “madre” and “padre”
  - second-level ancestors should be “nonna” and “nonno”
  - third-level ancestors should be “bisnonna” and “bisnonno”
  - fourth-level ancestors should be “bisbisnonna” and “bisbisnonno”
  - ...

Exercise (2)

- Example output

```
% ./genealogy data
Input name: Mario Rossi
born in 1969
madre: Diana Bianchi
padre: Alberto Rossi
nonna: Celeste Verdi
nonno: Piero Bianchi
nonna: Maria Villa
nonno: Piero Rossi
. . .
Input name:
```

Exercise (3)

- Implement a doubly-linked list of integers implemented as a list with sentinel.

```c
#ifndef LIST_INT_H_INCLUDED
#define LIST_INT_H_INCLUDED

struct list_int;

struct list_int * list_int_new ();    /* constructor */
void list_int_delete (struct list_int *); /* destructor */

typedef struct list_int * list_int_iterator;

list_int_iterator list_int_append (struct list_int * l, int v); /* add after */
list_int_iterator list_int_insert (struct list_int * l, int v); /* add before */
list_int_iterator list_int_erase (struct list_int * l);   /* return next */

list_int_iterator list_int_begin (struct list_int * l);
list_int_iterator list_int_end (struct list_int * l);

int list_int_get (list_int_iterator i);
void list_int_put (list_int_iterator i, int v);

list_int_iterator list_int_next (list_int_iterator i);
list_int_iterator list_int_prev (list_int_iterator i);

#endif
```