Fundamentals of Programming

Data structures: Lists

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

March 21, 2012

Outline

Using arrays

- The C language provides two simple primitive data structures: arrays and structures
- Structures are for grouping different data relevant to a single *object* (e.g. a student, a complex number, a bank account, etc.)
- Arrays are for storing multiple instances of the same data (e.g. an array of integers, of students, of bank accounts, etc.)
- Both are treated statically:
  - When declaring an array, the size of the array must be a constant known at compile time, because the compiler must compute how much memory to allocate for the array
  - If we do not know the size at compile time, we have to resort to an array created dynamically with `malloc`

Address book

- Suppose we want to implement an address book
- Each entry in the book will contain information about a person's name, address, telephone, etc.

```
typedef struct abook_entry {
    char name[50];
    char address[100];
    char telephone[20];
} ABOOK_ENTRY;
```

How to store addresses

- To store a set of addresses, we could prepare an array with a maximum number of entries

```
typedef struct address_book {
    ABOOK_ENTRY entries[100];
    int num;
} ABOOK;

void abook_init(ABOOK *book);
void abook_insert(ABOOK *book, ABOOK_ENTRY *e);
ABOOK_ENTRY abook_search(ABOOK *book, char *name);
void abook_print(ABOOK *book);
```

- See complete code at `./examples/07.lists-examples/addressmain.c`

Problems

- The problem of this approach is that we don't know how many addresses we will need to store
- If we need more than 100, then the program fails
- If we need much less, then we are wasting memory
- A better approach is to re-size the array depending on the number of elements
- Also, most code can be generalised

Vector of pointers

- Let us define a generic vector of pointers

```
typedef struct vector_ptr {
    void **array;
    int num_elem;
    int array_size;
} VECTOR_PTR;

void vptr_init(VECTOR_PTR *v);
void vptr_push_back(VECTOR_PTR *v, void *elem);
void * vptr_pop_back(VECTOR_PTR *v);
void * vptr_elem_at(VECTOR_PTR *v, int i);
int  vptr_mem_size(VECTOR_PTR *v);
int  vptr_num_elem(VECTOR_PTR *v);
```

Pointer to pointer to void!

- Array is a pointer to an array of pointers to void

Interface

- Notice how we define an interface, and we access the data structure only through functions
- we say that `vector_ptr` is an *opaque* data structure, because the user should access the structure only through functions and never directly access the data fields

Enlarging

vector.c

```c
static void vptr_enlarge(VECTOR_PTR *v)
{
    if (v->array_size == 0) vptr_init(v);
    else v->array_size *= 2;
    void **tmp = malloc(v->array_size * sizeof(void *));
    memcpy(tmp, v->array, v->num_elem * sizeof(void *));
    free(v->array);
    v->array = tmp;
    return;
}

void vptr_push_back(VECTOR_PTR *v, void *elem)
{
    if (v->num_elem == v->array_size) vptr_enlarge(v);
    v->array[v->num_elem++] = elem;
    return;
}

void * vptr_pop_back(VECTOR_PTR *v)
{
    if (v->num_elem == 0) return 0;
    else return v->array[--v->num_elem];
}
```

Exercises

- An example of usage of the data structure can be found in `./examples/07.lists-examples/vector-ptr-main.c`
- As an exercise, create a similar data structure by storing copies of objects instead of pointers
- Advantage:
  - we can copy an entire data structure;
  - we can operate on copies without modifying the original

Searching

- Searching the data structure takes linear time in the number of elements.
- We could improve is we keep the data structure sorted according to field on which we want to search
  - If we want to search by name, we should order alphabetically by name
- Then, we apply binary search
  - start looking in the middle
  - If we have found the entry, finish with success
  - If the entry is “greater” than the one we look for, continue looking in the first half
  - If the entry is “less” than the one we look for, continue looking in the second half

Sorting

- There are many algorithms for sorting
- Insertion, Selection, Bubble, Shell, Merge, Heap, Quicksort, etc.
  - A good summary is here `http://www.sorting-algorithms.com/`
- The most popular is **quicksort**, a very good compromise in efficiency in many different cases

Exercise

- Implement the quicksort and bubblesort algorithms for integers
- Compare their performance on randomly generated arrays

Generic sorting

- Now suppose we want to implement an address book using the vector of pointers we just provided
- We also want to sort the array, and apply binary search
  - We could write our own sorting algorithm (e.g. quicksort)
  - However, the sorting algorithm is always the same; we don't want to rewrite it from scratch every time we need to sort something
  - therefore, the standard c library provides a quicksort algorithm already implemented

```
void qsort (void *array, size_t count,
            size_t size, cmp_fn_t compare);
```

- where `cmp_fn_t` is the type of the `compare` function

```
int cmp_fn_t (const void *, const void *);
```

Pointers to functions

- In C it is possible to define a pointer to a function
- The syntax is a little strange, pay attention:

```
void (*pf1)(int);

int (*pf2)(double, double);

char* (*pf3)(char*);
```

Variable `pf1` is a pointer to a function that takes an integer and returns nothing (void)

Variable `pf2` is a pointer to a function that takes two doubles and returns an integer

Variable `pf3` is a pointer to a function that takes a pointer to char and returns a pointer to char

Other examples

- How to use pointers to functions

```
typedef int (cmp_fn_t)(void *, void *);

int myfun(void *a, void *b);
cmp_fn_t pf = myfun;


int x, y;
pf(&x, &y);
```

Defines `cmp_fn_t` as the type of pointer to function that takes two pointers to void and returns an integer

pf is a variable that points to `myfun`

calls `myfun` by passing the address of x and y

Using qsort

- Using qsort with integers:

```
int cmp_int(const void *x, const void *y)
{
    return *((int *)x) > *((int *)y);
}
int main() {
    int arrayint[] = {4, 5, 6, 1, 2, 3, 0, 9, 7, 8};
    ...
    qsort(arrayint, 10, sizeof(int), cmp_int);
    ...
}
```

Using qsort with strings

- Using qsort with array of strings

```
int cmp_str(const void *x, const void *y)
{
    const char **p = (const char **)x;
    const char **q = (const char **)y;

    return strcmp(*p, *q);
}

int main() {
    char *array[] = {"ABC", "ZGF", "HLK", "SDF", "PLM", "BSD",
                     "KKK", "JFL", "VMZ", "CDA"};
    ...
    qsort(array, 10, sizeof(char *), cmp_str);
    ...
}
```

Binary search

- Also binary search is a well-established algorithm, so it can be generalised

```
void * bsearch (const void *key, const void *array,
                size_t count, size_t size,
                comparison_fn_t compare);
```

- `key` is the pointer to the element to search
- The comparison function should return -1 , 0 or 1 if the key is less than, equal to or greater than the element in the array
- You have to be particularly careful with strings (as always)
- see `./examples/07.lists-examples/stringsort.c`

Exercise

- Continue the addressbook exercise
  - Implement sorting and searching by name
  - Also implement sorting and searching by address using a second vector of pointers
  - If you use a second vector, how it is possible to perform addition and removal of elements? (assume unique keys)

Lists

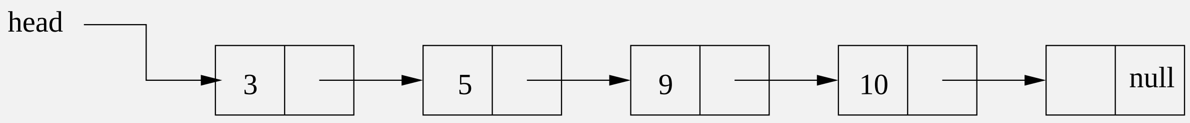
- With vector we can efficiently search and sort
- However, there are many cases where other data structures are more efficient
- for example, when we have frequent additions and deletions of elements in the middle
    - Adding an element and then sort has complexity $O(n^2)$ or $O(n \log n)$
    - Removing an element has always $O(n)$
    - Searching has $O(\log n)$ (if ordered)
- The list data structure can have some advantages over vector sometimes

Ordered Insertion

- **Problem:** suppose we have an ordered array of integers, from smalles to largest
- Suppose that we need to insert another number, and that after insertion the array must still be ordered
    - **Solution 1:** Insert at the end, then run a sorting algorithm (i.e. insert sort or bubble sort)
    - **Solution 2:** Identify where the number has to be inserted, and move all successive numbers one position forth
- Both solutions require additional effort to maintain the data structured ordered
- Another solution is to have completely different data structure

Lists

- A linked list is a collection of data structures, each one contains a pointer to the next one



- Every element of the list contains the data (in this case an integer), and a pointer to the following element in the list

List interface

list.h

```
#ifndef __LIST_H__
#define __LIST_H__

typedef struct l_node {
    int dato;
    struct l_node *next;
} LNODE;

typedef struct List {
    LNODE *head;
    int nelem;
} LIST;

void list_init(LIST *l);
void list_insert_h(LIST *l, int d);
void list_insert_t(LIST *l, int d);
int list_extract_h(LIST *l);
int list_extract_t(LIST *l);
void list_print(LIST *l);

#endif
```

Implementation

list.c

```c
void list_insert_h(LIST *l, int d)
{
    LNODE *p = (LNODE *)malloc(sizeof(LNODE));
    p->dato = d;
    p->next = l->head;
    l->head = p;
}

void list_insert_t(LIST *l, int d)
{
    LNODE *q = (LNODE *)malloc(sizeof(LNODE));
    q->dato = d;
    q->next = 0;

    LNODE *p = l->head;
    // caso particolare: lista vuota
    if (p == 0) l->head = q;
    else {
        // scorri fino all'ultimo elemento
        while (p->next != 0) p = p->next;
        // collega il nuovo elemento
        p->next = q;
    }
}
```

Problems with lists

- One of the problems with the list is that searching is a O(n) operation
  - while the previous algorithm on the array was O(log(n))
- The list is useful if we frequently insert and extract from the head
  - For example, inside an operating system, the list of processes (executing programs) may be implemented as a list ordered by process priority
  - In general, when most of the operations are inserting/estracting from the headm the list is the simplest and most effective solution

Data structures so far

- **Stack**
  - Insertion/extraction only at/from the top (LIFO)
  - All operations are *O(1)*
- **Queue** (Circular Array)
  - Insertion at tail, extraction from head (FIFO)
  - All operations are *O(1)*
- **Array** (random access)
  - Insertion at any point requires *O(n)*
  - Extraction from any point requires *O(n)*
  - Sorting requires *O(n log(n))*
  - Searching (in sorted array) requires *O(log(n))*
- **List** (ordered)
  - Insertion at any point requires *O(n)*
  - Extraction from any point requires *O(1)*
  - Searching requires *O(n)*

Exercise

- Implement a stack, using a list as a reference implementation
- Implement AddressBook as a list
- Implement a *double-linked list* (with pointers to go back and forth)
- Implement a method to visit a list in order (use the visitor pattern)