Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe and Sarah Winkler

22. April 2008

Proseminar Algorithmen und Datenstrukturen

Exercise sheet 5

Exercise 1 (Records)

This exercise deals with representation of records in C.

a) Consider the following C program:

```
#include <stdio.h>

struct example_struct{ char c1; char c2; } s;

int main() {

  printf("size␣of␣s:␣%d\n",sizeof(s));
  printf("size␣of␣s␣elements:␣%d\n", sizeof(s.c1)+sizeof(s.c2));

 return 0;
}
```

Compile and run it. Now consider the following program:

```
#include <stdio.h>

struct example_struct{ char c1; char c2; int i;} s;

int main() {

  printf("size␣of␣s:␣%d\n",sizeof(s));
  printf("size␣of␣s␣elements:␣%d\n", sizeof(s.i)+sizeof(s.c1)+sizeof(s.c2));
```

```
 return 0;
}
```

Consider now the following program:

```
#include <stdio.h>

struct example_struct{ char c; int i;} s;

int main() {

 printf("size of s: %d\n",sizeof(s));
 printf("size of s elements: %d\n", sizeof(s.i)+sizeof(s.c));

 return 0;
}
```

And finally this one:

```
#include <stdio.h>

struct example_struct{ char c1; int i; char c2;} s;

int main() {

 printf("size of s: %d\n",sizeof(s));
 printf("size of s elements: %d\n", sizeof(s.i)+sizeof(s.c1)+sizeof(s.c2));

 return 0;
}
```

What happens ? Explain the results and give a schema representing what happens in the memory stack for each of these examples.

In order to understand what is going on we need the notion of *memory alignment*. According to the C99 standard *memory alignment* is the requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address. For example, on a processor where an **int** occupies 4 bytes, a value of type **int** is *properly aligned* if its memory address is a multiple of 4. Certain processors require that data is properly aligned while others (e.g. INTEL x86 CPUs) do not have this restriction. However, accessing unaligned data is in general slower than accessing properly aligned data.

As far as the memory layout of structures is concerned, the C99 standard imposes the following constraints:

- Each non-bit-field member of a structure object is aligned in an implementation-defined manner appropriate to its type.
- Within a structure object, the non-bit-field members have addresses that increase in the order in which they are declared.
- There may be unnamed padding within a structure object, but not at its beginning.
- There may be unnamed padding at the end of a structure or union.

Regarding the current exercise, we see that when an **int** is declared after a **char**, the compiler obviously adds padding bytes such that the **int** is properly aligned.

The lesson to be learned from this exercise is that the size of a structure is not necessarily the same as the sum of the sizes of its members. Hence, do not write code that relies on this false assumption! The actual memory layout of a structure object depends on the compiler and the underlying hardware. In addition, most compilers possess compiler switches that influence the memory layout of structure objects.

b) Write a program that stores the names, two line addresses and ages of a group of people. Each person should be stored in a structure. Use an array of such structures to hold the data for a group of people. Write functions for reading a person's details from the standard input stream and for printing a person.

```
#include <stdio.h>
#define MAX 10

/*Read details from stdin, write output to screen */

struct person
{
  char surname[15], initial, address[2][25];
  int age;
} people[MAX];

void get_record (void);
void print_record (void);

int count = 0;

int
main ()
{
  get_record ();
  print_record ();
  return 0;
}

void
get_record (void)
{
  int i;
  char repeat = 'y';
  printf ("Type in the details for each person:\n");
  for (i = 0; repeat != 'n' && repeat != 'N' && i < MAX; count = ++i)
    {
      printf ("\nDetails for record no.%d\n", i + 1);
      printf ("Name (Surname) :");
      scanf ("%s", people[i].surname);
      getchar ();
      printf ("Initial%8c:", ' ');
      people[i].initial = getchar ();
      printf ("Address%8c:", ' ');
      scanf ("%s", people[i].address[0]);
      printf ("%16c", ' ');
      scanf ("%s", people[i].address[1]);
      printf ("Age%12c:", ' ');
      scanf ("%d", &people[i].age);
      getchar ();
      printf ("\nInput details for another person? ");
      repeat = getchar ();
      fflush (stdin);
    }
  printf ("\nTotal number of records read: %d\n\n", count);
}

void
print_record (void)
{
  int i;
  for (i = 0; i < count; i++)
    {
      printf ("%c. %s", people[i].initial, people[i].surname);
      printf ("\t%d\n", people[i].age);
      printf ("%s\n%s\n", people[i].address[0], people[i].address[1]);
    }
}
```

c) Consider the following record (in Pseudo-code):

Listing 1 RECORDS

```
record $T$ =
begin
  $n_1$ : character;
  $n_2$ : integer;
end
$A$ is declared as: array [1..10] of array [1..10] of T;
$B$ is declared as: array [1..5] of array [1..5] of T;
```

Given the size of a character is 1 and the size of an integer is 4.

- What's the size of A?
- What's the position of $A[3][5].n1$ in memory relative to the base address of A?
- What's the size of B?
- What's the position of $B[2][4].n2$ in memory relative to the base address of B?

- $sizeof(A) = 10 * 10 * sizeof(T) = 10 * 10 * (1 + 4) = 500$
- position of $A[3][5].n1 = ((3 - 1) * 10 + (5 - 1)) * sizeof(T) = 24 * 5 = 120$
- $sizeof(B) = 5 * 5 * sizeof(T) = 5 * 5 * (1 + 4) = 125$
- position of $B[2][4].n2 = ((2-1)*5+(4-1))*sizeof(T)+sizeof(character) = 8 * 5 + 1 = 41$

Exercise 2 (Data structures)

What are the data structures needed to construct:

- a doubly-linked list ?
- a tree with branching factor n ?
- a tree with arbitrary branching factor ?

Give these structures both in pseudo-code and in C code.

Exercise 3 (Linked List)

In this exercise you will implement a linked list: Provide insert, delete, search and print methods to manipulate the list according to the following specifications:

a) **data** the data in each cell is an integer.

add this method adds a given integer value at the list head.

delete this method removes a given integer value from the list.

search this method searches the list for a given integer value. It returns TRUE in case of success, FALSE otherwise.

print this method prints the list on the standard output.

Provide a main containing an example of creating a list, printing it and deleting some of its values.

b) **append** this method appends two lists given as arguments such that the first cell of the second list is linked to the last cell of the first list.

insert this method inserts a given integer value into the list such that the list is always ordered with increasing values.

Update your main to demonstrate these functions.