CH-230-A

Programming in C and C++

C/C++

Lecture 6

Dr. Kinga Lipskoch

Fall 2019

Using Bits Operations: A Problem

- Think of a low-level communication program
- Characters are stored in some buffer
- Each character has a set of status flags
  - `ERROR` true if any error is set
  - `FRAMING_ERROR` framing error occurred
  - `PARITY_ERROR` wrong parity
  - `CARRIER_LOST` carrier signal went down
  - `CHANNEL_DOWN` power was lost on device

Size Considerations

- Suppose each status is stored in additional byte
  - 8k buffer (real data)
  - But 40k status flags (admin data)
- Need to pack data

A Communication System

- `0 - ERROR`
- `1 - FRAMING_ERROR`
- `2 - PARITY_ERROR`
- `3 - CARRIER_LOST`
- `4 - CHANNEL_DOWN`

How to Initialize Bits

- `const int ERROR = 0x01;`
- `const int FRAMING_ERROR = 0x02;`
- `const int PARITY_ERROR = 0x04;`
- `const int CARRIER_LOST = 0x08;`

- If more states needed: `0x10`, `0x20`, `0x40`, `0x80`
- It is not intuitive to know which hexadecimal-value has which bit set

How to "Nicely" Set Bits

- `const int ERROR = (1 << 0);`
- `const int FRAMING_ERROR = (1 << 1);`
- `const int PARITY_ERROR = (1 << 2);`
- `const int CARRIER_LOST = (1 << 3);`
- `const int CHANNEL_DOWN = (1 << 4);`

Everyone will directly understand encoding of the bits, additional documentation can be greatly reduced

Structures

- A structure (i.e., `struct`) is a collection of variables
  - Variables in a structure can be of different types
- The programmer can define its own structures
- Once defined, a structure is like a basic data type, you can define
  - Arrays of structures,
  - Pointers to structures,
  - ...

Example: Points in the Plane

- A point is an object with two coordinates (= two properties)
  - Each one is a `double` value
- Problem: Given two points, find the point lying in the middle of the connecting segment
  - It would be useful to have a point data type
  - C does not provide such a type, but it can be defined

Defining the point struct

- The keyword `struct` can be used to define a structure

```
struct point {
  double x;
  double y;
};
```

- A point is an object with two doubles, called `x` and `y` of type `double`

Defining `point` Variables

- To declare a point (i.e., a variable of data type `point`), the usual syntax is used: type followed by variable name
  ```
  struct point a, b;
  ```
- `a` and `b` are two variables of type `struct point`

Accessing the Components of a `struct`

To access (read / write) the components (i.e., fields) of a structure, the selection operator . is used

```
struct point a;
a.x = 34.5;
a.y = 0.45;
a.x = a.y;
```

`struct` Initialization

- Like in the case of arrays, a structure can be initialized by providing a list of initializers
  `struct point a = { 3.0, 4.0 };`
- Initializations can use explicit field names to improve readability and code robustness (e.g., if `struct` definitions are modified)
  `struct point a = { .x = 3.0, .y = 4.0 };`
- As for arrays, it would be an error to provide more initializers than members available
- Initializers' types must match the types of the fields

struct Assignment

- The assignment operator (=) can be used also with structures

```
struct point a, b;
a.x = a.y = 0.2345;
b = a;
```

- The copying is performed field by field (keep this in mind when your structures have pointers)
- Warning: the relational operators (including equality test) are not defined for structures

Structures and Functions

A function can have parameters of type structure and can return results of type structure

```
struct point middle(struct point a,
    struct point b) {
  struct point retp;
  retp.x = (a.x + b.x ) / 2;
  retp.y = (a.y + b.y ) / 2;
  return retp;
}
```

Arrays of Structures

- It is possible to define arrays of structures
- The selection operator must then be applied to the elements in the array (as every element is a structure)

```
struct point list[4];
list[0].x = 3.0;
list[0].y = 7.3;
```

Pointers to Structures

- Structures reside in memory, thus it is possible to get their address
- Everything valid for the basic data types still holds for pointers to structures

```
struct point p;
struct point *pointpointer;
pointpointer = &p;
```

The Arrow Operator

- A structure can be modified by using a pointer to it and the dereference operator
  `(*pointpointer).x = 45;`
  - Parenthesis are needed to adjust the precedence of the operators `*` and `.`
- The arrow operator achieves the same goal giving the same result
  `pointpointer->x = 45;`

Dynamic Structures

- Pointers to structures can be used to allocate dynamically sized arrays of structures

```
struct point *ptr;
int number;
scanf("%d\n", &number);
ptr = (struct point *)malloc(sizeof(
  struct point) * number);
```

- You can access the array as we have already seen

```
ptr[0] = { 0.9, 9.87 };
ptr[1].x = 7.45;
ptr[1].y = 57.3;
```

Pointers and Structures: Self-referential Structures

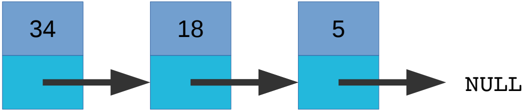
- Is it possible for a structure A to have a field of type A? No
- Is it possible for a structure A to have a field which is a pointer to A? Yes
  - This is called self reference
  - You will encounter many data structures organized by mean of self references
- Trees, Lists, ...

An Example: Lists

- A list is a data structure in which objects are arranged in a linear order
- The order in a list is determined by a pointer to the next element
  - While a vector has indices
- Advantages: lists can grow and shrink
- Disadvantages: access is not efficient

Linked Lists

- It is a standard way to represent lists
- A list of integers: every element holds an `int` plus a pointer to the next one
  - Recursive definition
- The last element's pointer points to `NULL`



Linked Lists in C

- Every element (node) holds two different information
  - The value (integer, float, double, char, array, ...)
  - Pointer to the next element
- This "calls" for a structure

```c
struct list {
  int info;
  struct list *next;  /* self reference */
};
```

Building the Linked List

```c
struct list a, b, c;
struct list *my_list;
my_list = &a;
a.info = 34;
a.next = &b;
b.info = 18;
b.next = &c;
c.info = 5;
c.next = NULL;    /* defined in stdlib.h */
```

- `NULL` is a constant indicating that the pointer is not holding a valid address
- In self-referential structures it is used to indicate the end of the data structure

Printing the Elements of a Linked List

```
void print_list(struct list* my_list) {
  struct list *p;
  for(p = my_list; p; p = p->next) {
    printf("%d\n", p->info);
  }
}
/* Using a while loop
void print_list(struct list* my_list) {
  while (my_list != NULL) {
    printf("%d\n", my_list->info);
    my_list = my_list->next;
  }
}*/
```

To print all the elements of a list, `print_list` should be called with the address of the first element in the list

Dynamic Growing and Shrinking

- Elements added and deleted to lists are usually allocated dynamically using the `malloc` and `free` functions
  - The example we have seen before is not the usual case (we assumed the list has content)
- Initially the list is set to empty (i.e., it is just a `NULL` pointer)

```
struct list *my_list = NULL;
```

Inserting an Element in a Linked List (1)

```
/* Inserts a new int at the beginning of the list
   my_list list where element should be inserted
   value integer to be inserted
   Returns the updated list
*/

struct list* push_front(struct list *my_list, int value) {
  struct list *newel;
  newel = (struct list *) malloc(sizeof(struct list));
  if (newel == NULL) {
    printf("Error allocating memory\n");
    return my_list;
  }
  newel->info = value;
  newel->next = my_list;
  return newel;
}
```

Inserting an Element in a Linked List (2)

```c
/* Like the previous one, but inserts at the end */

struct list* push_back(struct list* my_list, int value) {
  struct list *cursor, *newel;
  cursor = my_list;
  newel = (struct list *) malloc(sizeof(struct list));
  if (newel == NULL) {
    printf("Error allocating memory\n");
    return my_list;
  }
  newel->info = value;
  newel->next = NULL;
  if (my_list == NULL)
    return newel;
  while (cursor->next != NULL)
    cursor = cursor->next;
  cursor->next = newel;
  return my_list;
}
```

Freeing a Linked List

```
/*
  Disposes a previously allocated list
*/

void dispose_list(struct list* my_list) {
  struct list *nextelem;
  while (my_list != NULL) {
    nextelem = my_list->next;
    free(my_list);
    my_list = nextelem;
  }
}
```

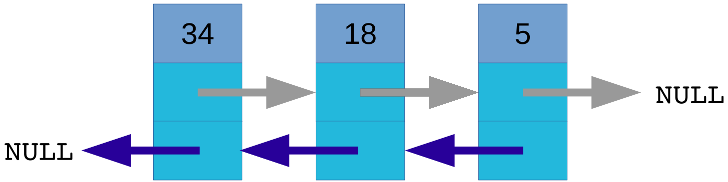
Using a Linked Lists

```
/*
  Here go the definitions we have seen before
*/

int main() {
  struct list* my_list = NULL;

  my_list = push_front(my_list, 34);
  my_list = push_front(my_list, 18);
  my_list = push_back(my_list, 56);
  print_list(my_list);
  dispose_list(my_list);
}
```

Doubly Linked Lists



Circular Doubly Linked Lists