

CH-230-A

Programming in C and C++

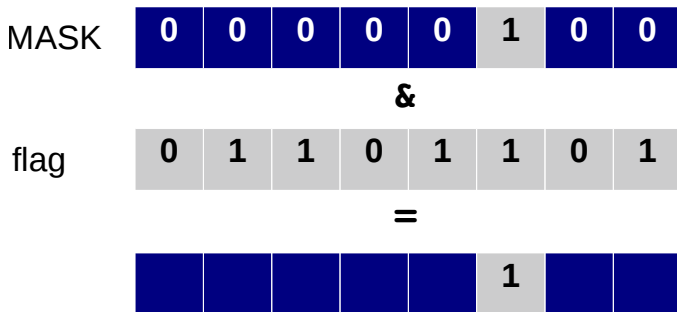
C/C++

Tutorial 6

Dr. Kinga Lipskoch

Fall 2021

Using Masks to Identify Bits



Using Masks

- ▶ Bitwise AND often used with a mask
- ▶ A mask is a bit pattern with one (or possibly more) bit(s) set
- ▶ Think of 0's as opaque and the 1's being transparent, only the mask 1's are visible
- ▶ If `result > 0` then at least one bit of mask is set
- ▶ If `result == MASK` then the bits of the mask are set

binary.c

```
1  #include <stdio.h>
2  char str[sizeof(int) * 8 + 1];
3  const int maxbit = sizeof(int) * 8 - 1;
4  char* itobin(int n, char* binstr) {
5      int i;
6      for (i = 0; i <= maxbit; i++) {
7          if (n & 1 << i) {
8              binstr[maxbit - i] = '1';
9          }
10         else {
11             binstr[maxbit - i] = '0';
12         }
13     }
14     binstr[maxbit + 1] = '\\0';
15     return binstr;
16 }
17 int main()
18 {
19     int n;
20     while (1) {
21         scanf("%i", &n);
22         if (n < 0) break;
23         printf("%6d: %s\\n", n, itobin(n, str));
24     }
25     return 0;
26 }
```

How to Turn on a Particular Bit

- ▶ To turn on bit 1 (second bit from the right), why does `flags += 2` not work?
 - ▶ If `flags = 2 = 000000010(2)`
 - ▶ Then `flags += 2` will result in
 - ▶ `flags = 4 = 00000100(2)` which "unsets" bit 1
- ▶ Correct usage:
 - ▶ `flags = flags | 2` is equivalent to
 - ▶ `flags |= 2` and turns on bit 1

How to Toggle a Particular Bit

- ▶ To toggle bit 1
 - ▶ `flags = flags ^ 2;`
 - ▶ `flags ^= 2;` toggles on bit 1
- ▶ General form
 - ▶ `flags ^= MASK;`

How to Test a Particular Bit

- ▶ To test bit 1, why does `flags == 2` not work?
- ▶ Testing whether any bit of MASK are set:
 - ▶ `if (flags & MASK) ...`
- ▶ Testing whether all bits of MASK are set:
 - ▶ `if ((flags & MASK) == MASK) ...`

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

```
1 struct point {  
2     double x;  
3     double y;  
4 };
```

- ▶ 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

```
1 struct point a;  
2 a.x = 34.5;  
3 a.y = 0.45;  
4 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

```
1 struct point a, b;  
2 a.x = a.y = 0.2345;  
3 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

```
1 struct point middle(struct point a,  
    struct point b) {  
2     struct point ret;   
3     ret.x = (a.x + b.x ) / 2;  
4     ret.y = (a.y + b.y ) / 2;  
5     return ret;  
6 }
```

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)

```
1 struct point list[4];  
2 list[0].x = 3.0;  
3 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

```
1 struct point p;  
2 struct point *pointpointer;  
3 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

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

- ▶ You can access the array as we have already seen

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


Pointers and Structures: Self-referential Structures

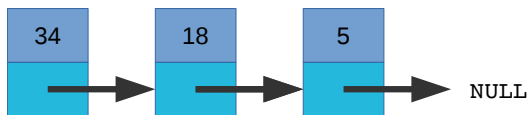
- ▶ Is it possible for a structure **A** to have a field of type **A**? No
- ▶ Is it possible for a structure **A** to have a field which is a pointer to **A**? Yes
 - ▶ This is called self reference
 - ▶ You will encounter many data structures organized by mean of self references
- ▶ Trees, Lists, ...

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

```
1 struct list {  
2     int info;  
3     struct list *next;    /* self reference */  
4 };
```

Building the Linked List

```
1 struct list a, b, c;  
2 struct list *my_list;  
3 my_list = &a;  
4 a.info = 34;  
5 a.next = &b;  
6 b.info = 18;  
7 b.next = &c;  
8 c.info = 5;  
9 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

```
1 void print_list(struct list* my_list) {
2     struct list *p;
3     for(p = my_list; p; p = p->next) {
4         printf("%d\n", p->info);
5     }
6 }
7 /* Using a while loop
8 void print_list(struct list* my_list) {
9     while (my_list != NULL) {
10         printf("%d\n", my_list->info);
11         my_list = my_list->next;
12     }
13 }*/
```

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)

```
1  /* Inserts a new int at the beginning of the list
2     my_list list where element should be inserted
3     value integer to be inserted
4     Returns the updated list
5  */
6
7  struct list* push_front(struct list *my_list, int value) {
8      struct list *newel;
9      newel = (struct list *) malloc(sizeof(struct list));
10     if (newel == NULL) {
11         printf("Error allocating memory\n");
12         return my_list;
13     }
14     newel->info = value;
15     newel->next = my_list;
16     return newel;
17 }
```


Inserting an Element in a Linked List (2)

```
1  /* Like the previous one, but inserts at the end */
2
3  struct list* push_back(struct list* my_list, int value) {
4      struct list *cursor, *newel;
5      cursor = my_list;
6      newel = (struct list *) malloc(sizeof(struct list));
7      if (newel == NULL) {
8          printf("Error allocating memory\n");
9          return my_list;
10     }
11     newel->info = value;
12     newel->next = NULL;
13     if (my_list == NULL)
14         return newel;
15     while (cursor->next != NULL)
16         cursor = cursor->next;
17     cursor->next = newel;
18     return my_list;
19 }
```

Freeing a Linked List

```
1  /*
2     Disposes a previously allocated list
3  */
4
5  void dispose_list(struct list* my_list) {
6      struct list *nextelem;
7      while (my_list != NULL) {
8          nextelem = my_list->next;
9          free(my_list);
10         my_list = nextelem;
11     }
12 }
```

Using a Linked Lists

```
1  /*
2   Here go the definitions we have seen before
3  */
4
5  int main() {
6      struct list* my_list = NULL;
7
8      my_list = push_front(my_list, 34);
9      my_list = push_front(my_list, 18);
10     my_list = push_back(my_list, 56);
11     print_list(my_list);
12     dispose_list(my_list);
13 }
```