book.sty

0.00.5em

0.0.00.5em

0.0.0.00.5em

0em

cpp Documentation

Release 1.0.1

abhishekcs10

December 19, 2016

CONTENTS

Contents:

CHAPTER

ONE

INPUT

1.1 input number to arbitrary precision

```
#include <math.h>

float val = 37.777779;

float rounded_down = floorf(val * 100) / 100;   /* Result: 37.77 */
float nearest = roundf(val * 100) / 100;  /* Result: 37.78 */
float rounded_up = ceilf(val * 100) / 100;      /* Result: 37.78 */
```

Notice that there are three different rounding rules you might want to choose: round down (ie, truncate after two decimal places), rounded to nearest, and round up. Usually, you want round to nearest.

As several others have pointed out, due to the quirks of floating point representation, these rounded values may not be exactly the "obvious" decimal values, but they will be very very close.

Above code prints double number upto n precision

1.2 roundf->

round double number to nearest integer. Thus to get nearest integer precision do->

```
number=(number*(10^p))/(10^p); //number is rounder for p precision
```

CHAPTER

TWO

OUTPUT

2.1 print to arbitrary precision

```
printf("%.*lf",n,double);
```

Above code prints double number upto n precision

CHAPTER

THREE

DATE FORMAT

3.1 Determine day on given date

```
#include<stdio.h>

int dayofweek(int d, int m, int y)
{
        static int t[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
        y -= m < 3;
        return ( y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
}

/* Driver function to test above function*/
int main()
{
        int day = dayofweek(30, 8, 2010);
        printf ("%d", day);
        return 0;
}
```

3.2 Explanation of above

Let us start with the simple scenario in which leap years did not exist and every year had 365 days.

Knowing what day January 1 falls on a certain year, it is easy to find which day any other date falls. This is how you go about it : January has $31 = 7 \times 4 + 3$ days, so February 1 will fall on the day which follows three days after January 1. Similarly, March 1 will fall on the day three days after the day corresponding to January 1, April 1 will fall 6 days after, and so on. Thus, the first days of each month are offset with respect to January 1 by the array {0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5}. This array is essentially what t[] is. Notice that it is slightly different from the t[] given in the question, but that is due to leap years and will be explained later. Once the day corresponding to the first date of the month is known, finding the day for any other date is just a matter of addition.

Since $365 = 7 \times 52 + 1$, the day corresponding to a given date will become incremented by 1 every year. For example, July 14, 2014 is a Monday and July 14, 2015 will be a Tuesday. Hence adding the difference between year numbers allows us to switch from the day of a reference year to any other year.

At this stage, the leap-year free dow function can be written as such:

```
int dow(int y, int m, int d){
  static int t[] = {0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5};
  return (y + t[m-1] + d + c) % 7;
}
```

Here c is a constant which corresponds to setting the "origin" of the day system : y, t[m] and d only tell us how many days to shift by; we need a reference to start the shifting and which is provided by c.

Now let us look at the real case when there are leap years. Every 4 years, our calculation will gain one extra day. Except every 100 years when it doesn't. Except every 400 years when it does (Seriously, what kind of intelligent designer comes up with this stuff? Couldn't the duration of the year have been an integer multiple of the synodic day?). How do we put in these additional days? Well, just add y/4 - y/100 + y/400. Note that all division is integer division. This adds exactly the required number of leap days.

Except there is a tiny problem. The leap day is not January 0, it is February 29. This means that the current year should not be counted for the leap day calculation for the first two months.

How do we do this? Well, there are probably many intuitive ways to go about this. But this piece of code sacrifices intuition for brevity. Suppose that if the month were January or February, we subtracted 1 from the year. This means that during these months, the y/4 value would be that of the previous year and would not be counted.

Smart, right? Except for a tiny problem. We are subtracting 1 from the year for January and February for non-leap years too. This means that there would be a "blank" day between February 28 and March 1, that is, we have made every non-leap year a leap year, and leap years double-leap years. Hm, so what if we subtracted 1 from the t[] values of every month after February? That would fill the gap, and the leap year problem is solved. That is, we need to make the following changes:

```
t[] now becomes {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4}
```

if m corresponds to Jan/Feb (that is, m<3) we decrement y by 1 the annual increment inside the modulus is now y + y/4 - y/100 + y/400 in place of y

That's it, we get the full solution

```
int dow(int y, int m, int d){
  static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
  y -= m < 3;
  return (y + y/4 - y/100 + y/400 + t[m-1] + d + c) % 7;
}
```

Coincidentially, c just happens to be 0 - this is an empirical fact and cannot be "derived" from anything we have done till now - and we get back the function in the question.

CHAPTER

FOUR

ALLOCATION IN C

applies to pointer of data structure

4.1 malloc

```
(void *) malloc (num*sizeof(arr));
```

simply allocates memory

4.2 calloc

```
(void *) calloc(int num, sizeof(type));
```

clears memory and initializes to zero.

4.3 c++ allocation

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
int* foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.

```
int *k;
k=new (nothrow) int[i];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
1 int* foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4 // error assigning memory. Take measures.
5 }
```

CHAPTER

FIVE

ALGORITHM

5.1 sorting

5.1.1 constructors

```
sort(begin_iterator, end_iterator)
sort(begin,end,cmp)
```

where cmp is any function that defines what comparison makes the first arguement of the function smaller than the other.

eg:

::

cmp(a,b){

if(a<b) return true;

else return false;

CHAPTER

SIX

VECTOR

6.1 unique(begin,end)

returns iterator of last element inserted and does not alters the size of container

6.2 resize(count)

:: resize(end-begin)

where end denotes the last index of resized array and begin represents the starting index of resized array.

6.3 assign(int how_many, int what_value)

assigns new values to vector array

```
arr.assign(n,100)
```

assigns n elements initialized with 100

```
arr (c_arr,c_arr+n)
```

assigns value from c_array to vector

CHAPTER

SEVEN

STACK

7.1 Constructor

```
stack<type> var;
```

7.2 insert

```
var.push(i)
```

7.3 access-top

```
var.top()
```

return top element

7.4 pop top

```
var.pop()
```

deletes top element and return nothing

CHAPTER

EIGHT

QUEUE

8.1 Constructor

```
queue<type> var;
```

8.2 insert -rear

```
var.push(x)
```

8.3 delete -front

```
var.pop()
```

8.4 access front element //the first inserted element

```
var.front()
```

return the front element

8.5 check if queue is empty

```
var.empty()
```

returns true if empty

8.6 access rear element //newest inserted

```
var.back()
```

CHAPTER

NINE

PRIORITY QUEUE(MAX)

9.1 Constructor

```
priority_queue<type> var;
```

9.2 Insertion

```
var.push()
```

9.3 Deletion

```
var.pop()
```

returns nothing

9.4 Access top element

```
var.top()
```

returns the max element

CHAPTER

TEN

MAP

10.1 Create map

map<key_type, value_type> arr;

10.2 Insert

arr[key]=value;//inserts if key doesnot exist and replaces value if exists

10.3 iterator

map< , >::iterator it;

10.4 find key

arr.find(key);

returns iterator and if iterator ==arr.end() it is not present in map

10.5 access value using iterator

it->first gives key

it->second gives value

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search