

## Time Design

### Problem Statement:

The problem is to calculate and display a new time of day given as input a current time and a wait time. Specifically each input will be entered as two integer values representing hours and minutes. Time is expected to be in 24 hour notation.

To solve this problem an internal representation for time must be chosen. My design will keep the hours and minutes in integer variables with constraints on the possible integers allowed. For 24 hour notation this means that the hour portion must always be between 1 and 24 inclusive. and the minutes portion must always be between 0 and 59 inclusive. My design will also output to the user an indicator if the new time occurs on a different day than the current time (i.e. the new time is greater than 24:00). Another possible option would be to convert the input times into minute only representations. In this case 1 = 00:01 (1 minute past midnight), 720 = 12:00 (noon) and 1440 = 24:00 (midnight).

### Variable Names and Types for Input:

Along with the list of variable required you should state any constraints that you choose to imposed on the input values.

```
int curr_hours // hour portion of current time in 24 hour notation,
               // valid values range from 1 - 24, where 12 represents noon and 24 represents midnight
int curr_mins  // minute portion of current time, valid values range from 0 - 59
int wait_hours // number of hours to be added to current time, valid values limit is size of integer
int wait_mins  // number of minutes to be added to current time, valid values range from 0 - 59
```

### Procedural Abstraction:

#### INPUT DATA:

Since the values from the user are to be constrained to proper hour and minute values a function will be implemented to ensure that the user enters only valid information for the current time and the wait time. Since the edits checks for these to inputs are different there will be two different input functions.

```
//-----
// getCurrentTime
//
// pre-condition:
//   none
// post-condition:
//   hours must be entered as a value between 1 and 24 inclusive
//   minutes must be entered as a value between 0 and 59 inclusive
//   the current time will be returned to calling function in an
//   minute only representation
//-----
int getCurrentTime( );
```

```
//-----
// getWaitTime
//
// pre-condition:
//   formal parameter are passed by value
// post-condition:
//   hours will contain a positive value
//   minutes will contain a value between 0 and 59 inclusive
//   the wait time will be returned to calling function in an
//   minute only representation
//-----
int getWaitTime( );
```

#### CALCULATIONS:

There are two situations to consider given the representation identified above. The first is that adding minutes may result in the case that the hours portion needs to be incremented by 1. This happens only if the sum of the current minutes plus the wait minutes is greater than 59. The second issue is that adding hours may result in the

case that the hours portion is greater than 24.

These calculations can be isolated into two functions

Given the minute only representation calculating the new time is a simple addition operation followed by a mod operation.

```
new time = (curr_time + wait_time) % 1440
```

OUTPUT:

Since the internal representation is minute only the output function will need to convert the minute only representation back into the 24 hour clock notation the user is expecting.

```
//-----  
// convertMintimeTo24HourTime  
//  
// pre-condition:  
//   the first argument passed into this function must be between 1 and  
//   1440 inclusive (minute only time representation), followed by two  
//   pass by reference parameters for receiving the results  
// post-condition:  
//   the minute_time received will be converted into separate hour and  
//   minute representation where hours is a 24 hour clock value  
//-----  
output24HourTime(int minute_time, int& hour, int& minute)  
  
hour = minute_time/60  
minute = minute_time % 60
```

Weight Design

Problem Statement:

The problem is to convert a weight given in one representation (i.e. pounds and ounces) into a different representation (i.e. kilos and grams). The weight is to be entered in two parts constrained by the representation (i.e. english or metric). In the english case weight is to be input in terms of pounds and ounces. There are 16 oz in 1 pound and so the input from the user must be constrained to be no more than 15 for the ounces portion. In the metric case we choose to allow only integer values to be entered and we restrict the grams value to be less than 1000, since 1000 grams is equivalent to 1 kilo. Thus in either case the input will be received from the user as two integer values with the second value constrained as indicated above.

The conversion factors for converting between english and metric weight scales are know and will be coded as CONST variables. The internal representation for english weight will be calculated in an input data function from the two values input from the user and returned to the main program in the form of a double stored in a double variable type. A suitable function declaration would be:

```
//-----  
// getEnglishWeightInput  
//  
// pre-condition:  
//   none  
// post-condition:  
//   two integers are requested from the user which will contain  
//   the pounds and ounces portions of the weight to be converted.  
//   The ounces portion must be a positive value less than 16. The  
//   entered values will be combined into a single real valued pound  
//   representation and this value will be returned as a double.  
//   For example if the user enters 4 pounds and 5 ounces This  
//   function would return the value equivalent 4.3125,  
//   since  $5/16 = 0.3125$  and  $4 + 0.3125 = 4.3125$   
//-----  
double GetEnglishWeightInput()
```

Thus the input values are immediately converted into a single representation.

Once the weight is converted into a single number the conversion to metric is a simple arithmetic operation based upon the fact that 1 pounds = 0.45359237 kilograms

The result can then be passed to a print output function which will convert the weight into kilos and grams for output to the user.

```
//-----  
// outputMetricWeight  
//  
// pre-condition:  
//   an argument will be passed as a double that contains a weight  
//   represented in pounds. For example 4 pound 5 ounces is equivalent  
//   in value to 4.3125 pounds, since 5/16 = 0.3125  
//  
// post-condition:  
//   two integers are requested from the user which should contain  
//   the pounds and ounces portions of the weight to be converted.  
//   The ounces portion must be a positive value less than 16. The  
//   entered values will be combined into a single real valued pound  
//   representation and returned as a double  
//-----  
void outputMetricWeight(double EnglishWeightInPounds)
```

#### CALCULATIONS:

The grams portion of a metric weight value can be obtained by taking the mod 1000 of the Kilo value.

this gives us the following variable and calculations:

```
double kiloweight
```

```
int justKilosWeight
```

```
int justGramsWeight
```

`kiloweight % 1000` --> produces the grams portion as an integer and can be directly passed to a 'cout' or can be assigned to an integer variable like 'justGramsWeight' and then the variable can be passed to cout.

`kiloweight/1000` --> produces the kilo portion as an integer when assigned to the integer variable justKilosWeight