CHAPTER XIII

Times and Dates

Times and dates might be difficult for the beginning programmer to understand because they are not simple variables. They consist of several, perhaps many, components. To further confuse the issue, a C compiler typically comes with many different functions that all handle time differently. When should each of these be used? This chapter attempts to answer some of the frequently asked questions relating to times and dates.

XIII.1: How can I store a date in a single number? Are there any standards for this?

Answer:

You might want to convert a date to a single number for several reasons, including for efficient storage or for simple comparison. Additionally, you might want to use the resultant number as part of a coding scheme. In any case, if you want to represent a date as a single number, you need to ask yourself why you need to do this and what you intend to do with the number after you have converted it. Answering these questions will help you determine which method of conversion is superior. First, look at a simple-minded example:

```
#include <stdio.h>
#include <stdlib.h>
```

```
main()
{
    int month , day , year;
    unsigned long result;

    printf( "Enter Month, Day, Year: \n" );
    fflush( stdout );
    scanf( " %d %d %d" , &month , &day , &year );

    result = year;
    result |= month << 12;
    result |= day << 14;

    printf( "The result is: %ul.\n" , result );
}
```

This program converts three variables into a single number by bit manipulation. Here is a sample run of the program:

```
Enter Month, Day, Year:
11 22 1972
The result is: 47028l.
```

Although this program does indeed work (you can test it by entering it into your computer), it contains several deficiencies. It might be a good idea to try to figure out what some of the deficiencies are before proceeding to the next paragraph.

Did you think of any defects? Here are several:

- The month, day, and year are not constrained. This means that the fields must be larger than is perhaps necessary, and thus efficiency is being sacrificed. Furthermore, the user could enter arbitrarily large values that would overwrite the bounds of the bit fields, resulting in a corrupted date.
- The numbers that are produced for the date are not in order; you cannot compare dates based on their numbers. This feature might be very convenient to have!
- The placement of the elements into the final number is simple, if arbitrary. Extraction is, however, not so simple. (Can you see a simple way to do it?) You might want a simpler format for storing dates that will allow for simple extraction.

These issues will be addressed one by one.

The months need to range only from 1 to 12, and the days need to range only from 1 to 31. Years, however, are another matter. You might, depending on your purpose, choose a limit on the range of years that you need to represent in the program. This range will vary, depending on the purpose of the program. Some programs might need to represent dates in the distant past, and others might need to store dates in the distant future. However, if your program needs to store only years from 1975 to 2020, you can save quite a bit of storage. You should, of course, test all the elements of the date to ensure that they are in the proper range before inserting them into the number.

NOTE

An archaeological database might be a good example of a system that would need dates in the far past.

In the C language, generally, counting (for arrays and such) begins at zero. It will help, in this case, to force all the number ranges to begin at zero. Therefore, if the earliest year you need to store is 1975, you should subtract 1975 from all the years, causing the year series to start at zero. Take a look at the program modified to work in this way:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    int month , day , year;
    unsigned long result;

    /* prompt the user for input */
    printf( “Enter Month, Day, Year: \n” );
    fflush( stdout );
    scanf( “ %d %d %d” , &month , &day , &year );

    /* Make all of the ranges begin at zero */
    --month;
    --day;
    year -= 1975;

    /* Make sure all of the date elements are in proper range */
    if (
        ( year < 0 || year > 127 ) ||  /* Keep the year in range */
        ( month < 0 || month > 11 ) || /* Keep the month in range */
        ( day < 0 || day > 31 )        /* Keep the day in range */
      )
    {
        printf( “You entered an improper date!\n” );
        exit( 1 );
    }

    result = year;
    result |= month << 7;
    result |= day << 11;

    printf( “The result is: %ul.\n” , result );
}
```

This program doesn’t account for the fact that some months have fewer than 31 days, but this change is a minor addition. Note that by constraining the range of dates, you need to shift the month and date values by a lesser amount.

The numbers produced are still unsortable, however. To fix this problem, you need to make the observation that the bits farthest to the left in the integer are *more significant* than the ones to the right. What you should do, therefore, is place the most significant part of the date in the leftmost bits. To do this, you should change the part of the program that places the three variables into resultant numbers to work this way:

```
    result = day;
    result |= month << 5;
    result |= year << 9;
```

Here’s a test of this modification on some sample dates:

```
Enter Month, Day, Year:
```

```
11 22 1980
The result is: 11077l.

Enter Month, Day, Year:
12 23 1980
The result is: 11621l.

Enter Month, Day, Year:
8 15 1998
The result is: 7415l.
```

You can now store records with their date in this format, and sort them based on this number, with full confidence that the dates will be in the proper order.

The only issues that still need to be addressed are the somewhat arbitrary nature of storage within the value, and the question of extraction. Both problems can be solved by the use of bit fields. Bit fields are explained in Chapter X. Take a look at the code without further ado:

```
/* These are the definitions to aid in the conversion of
 * dates to integers.
 */
typedef struct
{
    unsigned int year : 7;
    unsigned int month : 4;
    unsigned int day : 5;
} YearBitF;

typedef union
{
    YearBitF date;
    unsigned long number;
} YearConverter;


/* Convert a date into an unsigned long integer. Return zero if
 * the date is invalid. This uses bit fields and a union.
 */

unsigned long DateToNumber( int month , int day , int year )
{
    YearConverter yc;
    /* Make all of the ranges begin at zero */
    --month;
    --day;
    year -= 1975;

    /* Make sure all of the date elements are in proper range */
    if (
        ( year < 0 || year > 127 ) ||  /* Keep the year in range */
        ( month < 0 || month > 11 ) || /* Keep the month in range */
        ( day < 0 || day > 31 )        /* Keep the day in range */
      )
      return 0;

    yc.date.day = day;
    yc.date.month = month;
    yc.date.year = year;
    return yc.number + 1;
}



/*  Take a number and return the values for day, month, and year
 *  stored therein. Very elegant due to use of bit fields.
 */

void NumberToDate( unsigned long number , int *month ,
                   int *day , int *year )
{
    YearConverter yc;

    yc.number = number - 1;
    *month = yc.date.month + 1;
    *day = yc.date.day + 1;
    *year = yc.date.year + 1975;
}

/*
 *  This tests the routine and makes sure it is OK.
 */


main()
{
    unsigned long n;
    int m , d , y;
    n = DateToNumber( 11 , 22 , 1980 );
    if ( n == 0 )
    {
        printf( "The date was invalid.\n" );
        exit( 1 );
    }
    NumberToDate( n , &m , &d , &y );
    printf( "The date after transformation is : %d/%d/%d.\n" ,
             m , d , y );
}
```

There is still a certain amount of inefficiency due to the fact that some months have fewer than 31 days. Furthermore, this variance in the number of days will make it hard to increment dates and to find the difference between dates in days. The built-in C functions are best for these more complex tasks; they will save the programmer from having to rewrite a great deal of code.

Cross Reference:

XIII.2: How can I store time as a single integer? Are there any standards for this?

XIII.2: How can I store time as a single integer? Are there any standards for this?

Answer:

The question of storing a time in a single byte is similar to that of storing a date in a single byte; therefore, it will be helpful for you to have read FAQ XIII.1. Nonetheless, there are differences.

The first thing you should note is that a time of day is more "deterministic" than a date of a year. You know exactly how many seconds there will be in a minute, how many minutes there will be in an hour, and how many hours there will be in a day. This uniformity makes handling times of day somewhat easier and less prone to error.

Following is a list of features that are desirable when you are choosing a method to convert a time to a number:

- It should be as efficient on space as possible.
- It should be able to store different kinds of time (standard, military).
- It should enable you to advance through time quickly and efficiently.

If you've read the preceding FAQ about dates, you might decide that a good way to handle the problem is to represent time as a bit field. That is not a bad idea, and it has several salient advantages. Look at the code for representing time as an integer:

```
/*
 *  The bit field structures for representing time
 */

typedef struct
{
    unsigned int Hour : 5;
    unsigned int Minute : 6;
} TimeType;

typedef union
{
    TimeType time;
    int Number;
} TimeConverter;


/*
 *  Convert time to a number, returning zero when the values are
 *  out of range.
 */

int TimeToNumber( int Hour , int Minute )
{
    TimeConverter convert;

    if ( Hour < 1 || Hour > 24 || Minute < 1 || Minute > 60 )
        return 0;
    convert.time.Hour = Hour;
    convert.time.Minute = Minute;
    return convert.Number +1 ;
}


/*
 *  Convert a number back into the two time
 *  elements that compose it.
 */

void NumberToTime( int Number , int *Hour , int *Minute )
{
    TimeConverter convert;

    convert.Number = Number - 1;
    *Hour = convert.time.Hour;
    *Minute = convert.time.Minute;
}


/*
 *   A main routine that tests everything to
 *   ensure its proper functioning.
 */

main()
{
    int Number , Hour , Minute;
    Hour = 13;
    Minute = 13;
    Number = TimeToNumber( Hour , Minute );
    NumberToTime( Number , &Hour , &Minute );
    printf( "The time after conversion is %d:%d.\n" , Hour , Minute );
}
```

Adding seconds to the time class is relatively easy. You would need to add a seconds field to the time structure and add one extra parameter to each of the conversion functions.

Suppose, however, that you want to use this resulting number as a clock, to “tick” through the day. To carry out this task using a bit field, you would have to convert the number into a bit field, increment the seconds, and test whether the seconds value had passed 60; if it had, you would have to increment the minutes, again testing to see whether the value had passed 60, and so on. This process could be tedious!

The problem here is that the elements of the time structure do not fit evenly into bits—they are not divisible by two. It is therefore more desirable to represent time mathematically. This can be done quite simply by representing a time of day by how many seconds (or minutes) have elapsed since the start of the day. If you represent the time in this fashion, incrementing the number will move the time to the next second (or minute). Take a look at some code that represents time in this way:

```
#include <stdio.h>
#include <stdlib.h>


/*
 *  A subroutine to convert hours and minutes into an
 *  integer number. This does no checking for the sake
 *  of brevity (you’ve seen it done before!).
 */

int TimeToNumber( int Hours , int Minutes )
{
    return Minutes + Hours * 60;
}


/*
 *  Convert an integer to hours and minutes.
 */

void NumberToTime( int Number , int *Hours , int *Minutes)
{
    *Minutes = Number % 60;
    *Hours = Number / 60;
}

/*
 *  A quickie way to show time.
 */

void ShowTime( int Number )
{
    int Hours , Minutes;
    NumberToTime( Number , &Hours , &Minutes );
    printf( “ %02d:%02d\n” , Hours , Minutes );
}



/*
 *  A main loop to test the salient features of the time class.
 */

main()
{
    int Number , a;

    Number = TimeToNumber( 9 , 32 );

    printf( “Time starts at : %d “ , Number );
    ShowTime( Number );

    /*
     *  Assure yourself that minutes are added correctly.
     */

    for( a = 0 ; a < 10 ; ++ a )
    {
        printf( “After 32 minutes : “ );
        Number += 32; /* Add 32 minutes to the time. */
        ShowTime( Number );
    }
}
```

This code provides a better representation of time. It is easy to manipulate and more compact, and it even allows for shorter code. Adding seconds is an exercise left to the reader.

This format is much like that used by the C functions `timelocal()` and `timegm()`. These functions count seconds from some arbitrary time/date. A slight modification of the routines presented here for both time and date should enable the programmer to utilize these functions or even his own definition of time.

Cross Reference:

XIII.1: How can I store a date in a single number? Are there any standards for this?

XIII.5: What is the best way to store the time?

XIII.3: Why are so many different time standards defined?

Answer:

Depending on the computer and compiler you are using, you might find many different time standards defined. Although having so many time standards might be convenient, it obviously took a lot of time to write all of them. And storing them all is taking up extra space on your computer's hard disk. Why bother? There are several reasons.

First, C is intended to be a portable language. Thus, a C program written on one make of computer should run on another. Often, functions that were particular to one system have had to be added to the C language when it was created on a new system. Later, when C programs need to be moved from that system to another, it is often easiest to add the specific commands to the target system. In this way, several versions of the same function could eventually be integrated into the C language. This has happened several times with the time function.

Second, there are several different possible uses for times (and dates). You might want to count time in seconds, you might want to count time as starting from a specified time and date, or you might want to count time in the smallest interval possible to ensure that your measure of time will be as accurate as possible. There is no best way to measure time. When you begin a program that involves time, you must examine the functions available to you and determine which are best suited to your purpose. If you are handling time in various ways, you might want to use several different time formats and functions. In this case, you might be glad that there are so many formats for time and that there was one to fulfill your needs.

Cross Reference:

XIII.1: How can I store a date in a single number? Are there any standards for this?

XIII.2: How can I store time as a single integer? Are there any standards for this?

XIII.4: What is the best way to store the date?

XIII.5: What is the best way to store the time?

XIII.4: What is the best way to store the date?

Answer:

To put it simply, there is no best way to store the date. The way you choose to store the date will depend on what exactly you plan to do with it. You might want to store it as an integer (perhaps counting days from a fixed time in history); as a structure containing month, day, year, and other information; or as a textual string. A textual string might seem to be impractical and difficult to handle, but as you shall see, it has its uses.

If you are merely keeping track of numerical dates, the problem is relatively simple. You should use one of the built-in formats, or represent time as an integer, and so on. You should determine whether you need to store the current date, update the date, check how far apart two dates are, and so on. There are ways to carry out many of these tasks and more using the formats and functions contained in the standard C library. However, you might be restricted if you become "locked into" one format too early in the development of your program. By keeping an open mind and by keeping your code flexible, you can use the most suited functions when the time comes.

However, you might want to represent dates in a more complex fashion. You remember dates in different ways. You don't always remember the exact date for everything; you might remember an important date in your life as "three days after my 16th birthday party" or remember a historical date as "10 years after the fall of the Ottoman Empire." Such kinds of dates cannot be stored as a simple numeric or structure. They require a more complex schema. You still might want to store such relative dates along with a reference to a known date that the computer can handle, or with a fixed date. This technique would aid in sorting and manipulating dates.

Cross Reference:

XIII.1: How can I store a date in a single number? Are there any standards for this?

XIII.5: What is the best way to store the time?

Answer:

The best way to store time depends entirely on what you need to store time for, and in what way you intend to manipulate the time values. Take a look at the different uses you might have for time, and how that might influence your choice of storage method.

Suppose that you only need to track events and that you need to track them in "real time." In other words, you want to ascertain the real-world time of when a given event occurred. The events you want to track might include the creation of a file, the start and completion of a long, complex program, or the time that a book chapter was turned in. In this case, you need to be able to retrieve and store the current time from the computer's system clock. It is better and simpler to use one of the built-in time functions to retrieve the time and to store it directly in that format. This method requires comparatively little effort on your part.

For various reasons, you might not want to store the time as formatted by the standard C functions. You might want a simpler format, for easier manipulation, or you might want to represent time differently.

In this case, it might be a good idea to represent time as an integer value, as demonstrated in the answer to FAQ XIII.2. This technique would enable you to advance through periods of time very simply and quickly and to compare different times to see which is earlier.

As with dates, you might have completely relative measures of time that will be difficult to quantify exactly. Although "half past noon" is not too hard to quantify, "after I eat lunch" is. This, however, might be not only the simplest way to track time but, in some cases, the only way! In these cases, you would simply have to store the textual string that describes the time. In this case, this is the best way to store time.

Cross Reference:

XIII.2: How can I store time as a single integer? Are there any standards for this?