C Programming for Techies

This month we'll look at the rational and a bit of the thought processes in translating a procedure into a C program... as well as a rather interesting bit of code.

Steve Rimmer

Programming is an art form, and like all art forms it must be approached with a certain amount of discipline, lest it become chaotic. Art only looks undisciplined to people who don't understand it.

The process of translating an idea into a working program is to a large extent intuition wrapped around a core of programming expertise and some rules which can be applied to programming projects in general. While you may never have to consider the creation of a program with quite this degree of abstract philosophy, the philosophy remains none the less.

It's all very zen-like.

This month we're going to take a rule and a desired function and walk through the steps involved in creating a program to make it all happen. Some are purely programming steps, but many are decisions based on an intuitive understanding of how programs should work and how users will apply them.

Visa Madness

This month's program is a small utility which will ascertain whether a credit card number is, in fact, valid. It's based on a somewhat unknown property of credit card numbers, that is, that they are self checking. The first digit of a credit card number tells one what kind of credit card the number belongs to. The last digit is a checksum. The intervening digits, which are unique to each individual card, can be run through a calculation which will spit out the checksum if all is well.

You might want to scrape the pocket lint off your plastic and see if all this makes sense to your cards.

The easiest thing to check by eye is the card type digit, the first one. This will by five for a MasterCard, four for a Visa card and three for an American Express card.

Credit card numbers are based on the concept of a checksum, something you might not have encountered before. In its simplest incarnation, a checksum is a value which is derived by adding together all the bytes in a block of data. If you store the data and store the checksum separately, you can subsequently make sure that the data hasn't been mangled by adding up the bytes again and seeing if the result still equals the checksum.

Obviously a checksum is not infallible. If one byte were to be decremented by one and another byte incremented by one, the checksum value would remain unchanged even though the data did not. However, for practical purposes a checksum provides a moderate level of data integrity with a tiny penalty in computer time. Checksums are quick and easy to calculate.

A checksum rarely consists of a true sum of the data it checks. Most checksums have an inherent modulus value imposed by the size of the object used to hold the checksum. For example, if you add all the bytes in a text file together and use a *char* to hold the running checksum, you will get a unique eight bit value with 256 possible states. All the bits beyond eight will be thrown away. This is a pretty decent checksum for most purposes.

The checksum used by credit card numbers is handled with a modulus of ten, as credit card digits can only have values from zero through nine.

The formula for working out a credit card checksum is a bit complex, presumably to make it difficult to fake in one's head. Each digit in the string has a position number, zero being the first, one the next one and so on. Digit zero is the card type digit, but it's included in the checksum calculation. The last digit in the string is the checksum, and is not.

The calculation is different depending on whether the length of the string is odd or even.

If the string length is odd, start with the last digit to be checked in the string and count backwards through the string. For each digit, if the position of the digit in the string is odd multiply the value of the digit by two and add it to the running checksum. Otherwise, just add it to the checksum as it is. The checksum will always be worked to modulo ten.

If the string length is even, the even numbered string position values are multiplied by two.

Practical Plastic

The intent, in writing a credit card checking program, should be to make it practical to use in applications where credit cards turn up a lot, such as in a store. We'll allow that the computers in most stores are pretty simple... and pret-

ty slow. We'll also allow that fancy graphics, a graphical user interface, complex instructions and so on will probably not go over well in a situation wherein someone must check a lot of numbers in a hurry, probably with someone waiting for their bill.

This is one of those situations in which a command line utility is probably ideal. Actually, a resident, pop-up program might be better still, but it's beyond the scope of this article to discuss how to write one.

I called the program which does this function PLASTIC.C, the resulting executable file from which got renamed to PL.EXE. The idea was to be able to type something like

```
PL 3999-999999-99000
```

and have it come back saying that this was a valid American Express number. In fact, it wouldn't do so in this case because this is *not* a valid number. Make sure you keep those valid plastic numbers you are aware of to yourself.

The credit card number is actually just a string of text, and it's desirable to treat it as such, rather than as a number. However, because the calculation of the checksum is dependant upon the position of the digits in the string, non- digit characters such as the dashes which separate the segments of the number on a credit card can't be allowed to interfere with the works.

You could just insist that the argument to the program be typed with nothing but numbers, but this would make the result hard to check by eye, especially because it would look different from the number as it's displayed on a credit card. It's better to create an internal version of the argument which has been cleaned up for the calculation.

Cleaning up the string is pretty easy, actually. Simply scan through the source string and copy to a destination string only those characters which are digits. The *isdigit* macro is helpful in this. You can see such a function in the source code accompanying this article.

There is another function which we'll find useful in performing the checksum calculations. Bearing in mind that it's frequently necessary to know whether a number is odd or even, we might do well to create a function that works this out. A number is odd if it can't be divided evenly by two, and if such a number is expressed in binary form, it will be found to have its first bit set no matter what the value of the complete number is. Thus, if the number is *n*, *(n & 1)* will be true if the number is odd.

You can write this into a function,

```
odd(n)
 int n;
{
 return(n & 1);
}
```

or you can make the compiler handle the calculation for you each time it occurs by using it as a macro.

```
#define odd(n)  (n & 1)
```

In the first case, your code will be minutely slower because your program will actually have to go and call the *odd* function each time it's needed. In the second case it will be faster but minutely larger because the actual code to do the calculation will appear multiple times throughout your program.

In the case of the credit card program the difference doesn't matter, but you should think about the differences between using functions and macros in more involved programming projects when you want to trade off size and speed.

Having done all this preliminary juggling, you can write the actual function to do the calculations pretty simply, as shown in the complete source for PLASTIC.C.

Don't Leave Home Without It

The dreadful temptation in doing little programs like this one is to embellish them into unusability. Add a couple of sound effects, flashing error messages, some graphics and pretty soon no one will use your code.

It's a good rule of thumb to assume that most people don't have colour monitors, don't listen to the sounds their computers make if, in fact, they can actually hear them over the ambient noise and wouldn't know what to do with a mouse short of tying an anchovy to it and taking it home to serve as an ersatz cat toy. ❐

```
 /*
 credit card number check program
 copyright (c) 1990 Alchemy Mindworks Inc.
 derived from a Pascal program by Daniel J.
Karnes
*/

#include "stdio.h"
#include "ctype.h"

#define MASTERCARD  '5'
#define VISA        '4'
#define AMEX        '3'

main(argc,argv)
 int argc;
 char *argv[];
{
 char b[128];
 int i;

 cprintf("\r\nAlchemy Mindworks "
     "Inc. credit card verification "
     "program version 1.0\r\n");

 if(argc>1) {
  for(i=1;i<argc;++i) {
   cleanString(b,argv[i]);
   cprintf("Card %-30.30s - ",argv[i]);
   verify(b);
   cprintf("\r\n");

  }
 } else cprintf("\r\nI need one or more "
         "credit card numbers to verify");
}
```

Cont'd. next page

Cont'd. from previous page

```
verify(s)
 char *s;
{
 int i,len,x=0,y=0,v=0;
 if(strlen(s) < 12) v=0;
 else {
  len = strlen(s);
  if(odd(len)) {
   for(i=(len-2);i>=0;-i) {
    if(odd(i)) y=((s[i]-'0')*2);
    else y=(s[i]-'0');
    if(y>=10) y=((y-10)+1);
    x+=y;
   }
  }
  else {
   for(i=(len-2);i>=0;-i) {
    if(odd(i)) y=(s[i]-'0');
    else y=((s[i]-'0')*2);
    if(y >= 10) y=((y-10)+1);
    x+=y;
   }
  }
  x=(10-(x % 10));
  if(x==10) x=0;
  if(x==(s[strlen(s)-1]-'0')) v=s[0];
  else v=0;
 }
 switch(v) {
  case VISA:
   cprintf("verifies as a Visa card.");
   break;
  case MASTERCARD:
   cprintf("verifies as a MasterCard.");
   break;
  case AMEX:
   cprintf("verifies as an American Express
card.");
   break;
  default:
   cprintf("is not a good credit card
number.");
   break;
 }
}

odd(n) /* return true if n is an odd number
*/
 int n;
{
 if(n & 0x0001) return(1);
 else return(0);
}

cleanString(dest,source)
 char *dest,*source;
{
 while(*source) {
  if(isdigit(*source)) *dest++=*source;
  ++source;
 }
 *dest=0;
}
```