Tutorial: Programming in C

Chapter 4, 7/6/93

NOTE: Previous chapters are available by anonymous ftp from isis.cshl.org in the directory pub/tutorial. Look there first before you ask me to send copies!

The Second Program: Blackjack

It's time to move on from the compound interest program to a new project. This time I've chosen an entertaining subject: a blackjack- playing program. This may seem trivial, but in fact it requires the introduction of several new and important features of the C language.

For the unfamiliar, and to clarify the particular set of rules that will be employed, blackjack is a card game, played with a standard 52-card deck. Each turn, each player is dealt a card from the deck, and given the option of "standing" (taking no card) or "hitting" (taking a new card). If the net value of the cards in a player's hand exceeds 21, they have lost the game. If the net value of a player's hand is exactly 21, they have won the game.

The 52 cards are divided into four suits (diamonds, clubs, hearts and spades). The 13 cards in each suit are named follows: two through ten, jack, queen, king, and ace. For simplicity's sake, the four "face cards" will be valued as worth 11, 12, 13, and 14 points (ace). The numbered cards will be worth their face value.

This is a substantial project, so we will break it into several chapters, in order to introduce new concepts gradually. The first version of the program will simply create a deck of cards, display them, shuffle the deck, display the deck again, and then deal several cards off the top.

The program follows. Note that here, for the first time, we write several other functions in addition to main(), in order to split up the program into manageable pieces. Several other new techniques are introduced as well.

(Name the file in which you type in the code "bj.c".)

```
--- #include  #include  #include
/* Blackjack, by Tom Boutell, 7/6/93 */
/* The function card_print takes a card as represented by an integer between 1 and 52 and
displays the name of the card. The actual function comes later; here at the beginning we give a
"prototype" to tell the compiler what to expect. */ void card_print(int card);
/* The function deck_shuffle accepts deck of cards, as represented by an array of integers, and
shuffles them into a random order. */
void deck_shuffle(int deck[]);
/* The function deck_draw draws a card from a deck. It returns an integer between 0 and 51
representing the card. deck_draw() reshuffles the deck if necessary. The function accepts an
array of cards and a pointer to the number of the top card in the array, which it will update after
drawing a card. */
int deck_draw(int deck[], int *card_top);
/* deck_init accepts a pointer to a deck of cards and fills it with an initial set of unshuffled cards.
*/
void deck_init(int deck[]);
/* deck_print prints out all the cards in the deck. (Naturally this isn't used in an actual game, but
the function will help us verify that the program is correct.) */
void deck_print(int deck[]);
/* Now the main program, which for now simply tests the functions above. */
```

```
int main() { /* The deck consists of an array of 52 integers. */ int deck[52]; /* The position in the
deck of the topmost card. In a freshly shuffled deck, this position will be 0; when only one card
is left, it will be 51. So deck[card_top] is the topmost card at any given time. */ int card_top = 0;
/* Counter variable. */ int i;
/* Card variable. */ int c;
/* Initialize random numbers. Take advantage of the clock to choose a different seed on each run.
If your compiler encounters difficulty compiling this line, you may wish to remove it, but if you
do so the same sequence of cards will appear on each run. */
srand(time(0));
/* Initialize the deck. */ deck_init(deck);
/* Print out the contents of the deck. */ printf("Unshuffled deck:\n"); deck_print(deck);
/* Now shuffle the deck. */ deck_shuffle(deck);
/* Print out the shuffled deck. */ printf("Shuffled deck:\n"); deck_print(deck);
/* Now draw the first five cards. */ printf("Top five cards:\n"); for (i=0; (i<5); i++) { /* Note the
use of the & operator to pass the address of card_top, rather than just its value, in order to permit
it to be changed. */ c = deck_draw(deck, &card_top);
printf("Drawing card %d: ", i); card_print(c); printf(" New position in deck: %d\n", card_top); }
printf("Test complete.\n"); /* All went well. */ return 0; } /* The actual code for card_print().
The "prototype" given earlier defined the arguments and return type of the function; now we
provide the actual implementation. */ void card_print(int card) { /* Suit of the card (0-3). */ int
suit; /* Value of the card (2-14). */ int value;
/* Calculate the value of the card. The % operator divides the number of the card by 13 and
yields the *remainder* of that division, yielding a number between 0 and 12. */
value = (card % 13) + 2;
/* Calculate the suit of the card. In C, division operations round down, so any card number
between 0 and 12, when divided by 13, will yield 0; any card between 13 and 25 will yield 1; and
so on. */
suit = card / 13;
/* Now print out the name of the card. */
printf("the ");
/* For the number cards (value between 2 and 10), just print the number */
if (value < 11) { printf("%d ", value); } else { /* Otherwise it's a "face" card; use a switch to print
the names */ switch (value) { case 11: printf("jack "); break; case 12: printf("queen "); break;
case 13: printf("king "); break; case 14: printf("ace "); break; default: /* This shouldn't happen,
so say so */ printf("Unknown card! Value: %d\n", value); break; } } printf("of "); switch (suit)
{ case 0: printf("hearts"); break; case 1: printf("diamonds"); break; case 2: printf("spades");
break; case 3: printf("clubs"); break; default: printf("Unknown suit! suit: %d\n", suit); break; } }
/* Implementation of deck_shuffle. */
void deck_shuffle(int deck[]) { /* Counter variable */ int i;
/* The card being swapped */ int c; /* Position of card being swapped with */ int cpos;
/* Our shuffling algorithm will be a simple one. It doesn't produce a perfect distribution of cards,
but neither does a real shuffling! We will exchange each card with another card, once. */
for (i=0; (i<52); i++) { /* Swap with a random position between 0 and 51. NOTE TO
EXPERTS: I know rand() is not one of the better random-number generators in the world, but at
least it's available in all standard C systems! */
```

cpos = rand() % 52; /* Get the card */ c = deck[i]; /* Now swap the two cards */ deck[i] = deck[cpos]; deck[cpos] = c; } }

/* Implementation of deck_draw(). The second argument is a "pointer" to a variable containing the number of the current top card. This approach allows deck_draw() to modify the value. */ int deck_draw(int deck[], int *card_top) { /* Position of top card */ int pos; /* Card to be returned */ int c;

/* Use the * operator, which in this context is the opposite of the & operator, to fetch the *value* stored at the *location* referred to by the pointer card_top */ pos = *card_top;

/* Fetch the card from the deck */ c = deck[pos]; /* Advance the card pointer */ pos++;

/* If we have just drawn the last card, reshuffle the deck */ if (pos == 52) { deck_shuffle(deck); /* Which brings us back to the first card, at position 0 */ pos = 0; }

/* Now store this pos back into the location pointed to by card_top */

*card_top = pos;

/* Finally we return the card drawn */ return c; }

/* Implementation of deck_init() */

void deck_init(int deck[]) { /* Counter variable */ int i;

/* Fill the deck with the 52 cards, in order */ for (i=0; (i<52); i++) { deck[i] = i; } }

/* Implementation of deck_print() */

void deck_print(int deck[]) { /* Counter variable */ int i; /* Current card */ int c;

/* Print out the 52 cards */ for (i=0; (i<52); i++) { c = deck[i]; card_print(c); /* Add a carriage return to separate cards */ printf("\n"); } } ---

Now I'll proceed through the program and explain the new features of the C language and its libraries used.

Note the new #include directives:

#include #include

The first is necessary to bring in the definitions of functions which deal with random numbers, which we need to shuffle the deck.

The second grants access to functions which manipulate the time of day. These are used to provide a different sequence of random numbers on each run. (Random number generators require a "seed" number, and will always produce the same sequence given the same seed.)

Now consider the following line:

void card_print(int card);

This line is a "function prototype." In this project, unlike the previous one, we break the program up into several distinct functions to perform various tasks. This allows a short, understandable main() program to be written, and places the details in separate, manageable packages.

In order to make the program easier to read, we place the main() function first, so the program can be immediately understood at a general level before tackling the details. The C language allows us to do this, but only if we give "prototypes" for each function that will be called. The prototype gives the return type of the function, the name of the function, and its list of arguments, which the compiler can use to check the accuracy of calls made to the function.

Function prototypes always end in a semicolon. This distinguishes them from actual functions, which are followed by code between braces ("{" and "}").

In this case, the function returns nothing, so its return type is "void". The function accepts one integer as an argument, so "int card" is specified. (The name of the argument is optional in a prototype, but providing a meaningful name helps make the purpose of the function clear.)

Now consider the prototype for deck_shuffle():
void deck_shuffle(int deck[]);
Once again, the return type is void.
The one argument, "deck", is an array of integers.
An array is like a numbered row of filing cabinets. To fetch something from one of them, you need to know the number of the cabinet. To represent a deck of cards, I use an array of 52 integers.
The "[]" characters after the name of the argument declare it to be an array.
Unlike simple variables, when an array is passed to a function, the array is the very same one that was passed from the calling function, not a copy of it. This is primarily because arrays can be quite large, and copying the entire array in order to pass it to a function is impractical and inefficient.
As a result, a function can change the elements of an array and expect that these changes will still hold true in the function which called it (main() or any other function). deck_shuffle() will take advantage of this to shuffle the deck.
Now consider the next prototype:
int deck_draw(int deck[], int *card_top);
This function, unlike those preceding it, does return a value, in this case an integer representing the card drawn from the deck. (I have chosen to represent cards with integers between 0 and 51.)
The first argument is the array of cards, as described above. The second is a "pointer" to a variable which holds the index of the first card in the deck.
A pointer is a special type of variable which, instead of containing a value, contains the *location* of a value.
"HUH?"
All right, listen closely. This is the most important concept in C programming, but fortunately it is not really that difficult to understand.
Imagine a friend wants to borrow your bicycle. There are two basic ways you can deal with this (besides saying no!):
You can buy him a brand-new bicycle identical to your own and give it to him. Needless to say this is not very practical.
OR, you can simply tell him where your bicycle is and give him the combination.
Alternatively, you may want your friend to *fix* your bicycle. In this case you have no choice: to fix the bicycle your friend needs to know where to find it!
A pointer is a piece of paper on which we have written down the *location* of something useful-- a bicycle, a house, or some other object that we either can't afford to copy or want to be modified by others.
Returning to our function prototype,
int *card_top
declares a *pointer* to an integer, giving the function the *location* of the integer instead of a copy of it. This way deck_draw() will be able to change which card is on top, and the change will affect the calling function as well.
Finally, we have:
void deck_init(int deck[]);
which declares a function that will be used to initialize the deck of cards with ascending values in ascending suits, and

void deck_print(int deck[]);
which will be used to print out the entire deck of cards, in order to help us decide if the program is correct.
Now we can move on to the main() function. Note the first variable declaration:
int deck[52];
This line declares an array of 52 integers.
Arrays, as discussed earlier, behave like a set of filing cabinets; you fetch something from the cabinets by knowing the number of the cabinet in which it is kept.
In the C language, "indexes" in an array begin at *ZERO* (NOT ONE). So in the deck[] array, above, the 52 integers are numbered from 0 through 51.
To refer to a particular item in an array, you follow the name of the array with a number in brackets ("[" and "]"). So the fifth element in the array above is:
deck[4]
(remember that indexes in an array begin at 0, so the fifth element is at index 4.)
Now note the following declaration:
int card_top = 0;
Here we have taken advantage of a convenience feature of C by specifying a value for card_top at the time we declare it. This is permissible, but the expression on the right-hand side of the "=" must be "static." A simplified explanation of this is that there should be no variables in the expression, so:
int x = 47+28/2;
is acceptable, but:
int x = y;
is *not* acceptable. (Of course x = y; is completely acceptable as a statement, *after* all variables have been declared!)
Now skip ahead to the following definition: srand(time(0));
This statement first calls the time() function, which returns the number of seconds that have elapsed since January 1st, 1970. It then passes the result to the srand() function, which accepts a seed and uses it to set up the random number generator. We use the time to do this in order to have a different series of random numbers, and thus a different shuffling of the deck, on each run of the program. As an experiment, try removing this line; you will find that the program produces identical results on every run.
Now consider the following:
deck_init(deck);
Here we call our deck_init() function to initialize the deck[] array.
Note that when passing an array to a function, no "[" and "]" characters are used. To pass the entire array we simply give the name of the array.
Notice how simple the main() function appears, due to the fact that the details have been "abstracted away" into separate functions.
In the same vein, we next invoke deck_print(), deck_shuffle(), and (again) deck_print() on our deck of cards, in order to test our functions, which together make up a family of tools for manipulating cards and decks of cards.
Finally we use a for loop to draw the first five cards. Inside the loop we use our deck_draw() function:
c = deck_draw(deck, &card_top);

Note the use of the & operator to pass the *location* of the variable card_top to the deck_draw() function, just as we passed the locations of variables in Chapter 3 to scanf() in order to store the user's input.
We then print out the card we have drawn:
```
printf("Drawing card %d: ", i); card_print(c); printf(" New position in deck: %d\n", card_top);
```
note the use of the %d specifier of printf() to output integers. The call to our card_print() function comes in between in order to place the name of the card on the same line; note the deliberate lack of a carriage return at the end of the first printf() call.
We print out the value of card_top in order to verify that it has been changed by deck_draw to reflect the new top card in the deck.
After the for loop, main() returns to the operating system with the usual success value.
Now consider the function card_print:
```
void card_print(int card) {
```
The first line looks much like the prototype; the difference is the opening "{" at the end, which signals that this is the beginning of the actual code of the function.
card_print() declares its own variables, just as main() does:
```
int suit; int value;
```
Note that these variables are COMPLETELY INDEPENDENT of the variables in any other function. We can declare variables by the same name as those in other functions and be confident that they will not interfere with those functions. Those who are used to older line-numbered versions of BASIC may initially find this confusing but it should come as a great relief to those tired of chasing down conflicts between identically- named variables.
Now consider the following:
```
value = (card % 13) + 2;
```
Here we compute the value of the card by using the % operator, which returns the remainder of an integer division (a number between 0 and 12 in this case), and then adding two to bring the value up to the range 2-14.
Next, we compute the suit:
```
suit = card / 13;
```
The suit of the card is represented by a number between 0 and 3. We get this number by dividing the card by 13, and taking advantage of the fact that division rounds down in C. Thus cards between 0 and 12, divided by 13, yield 0, while those between 13 and 25 yield 1, and so on.
The next several lines print the beginning of the card's name, then check to see if the card is a face card or a number card. If it is a number card, the value is simply printed out. If it is a face card (the "else" clause), then a "switch" statement is used to determine which face card it is.
The switch statement is useful for deciding between a large number of simple cases, such as the possible face cards. A sequence of if statements could be used instead to test each possible value, but switch is more convenient when there are a large number of possible values.
A switch statement consists of the "switch" keyword, followed by an expression in parentheses, followed by a series of "case" clauses and an optional "default" clause, all contained within a set of braces ("{" and "}"):
```
switch (value) { case 11: printf("jack "); break; case 12: printf("queen "); break; case 13: printf("king "); break; case 14: printf("ace "); break; default: /* This shouldn't happen, so say so */ printf("Unknown card! Value: %d\n", value); break; }
```

Here the expression in parentheses is simply the value of the card. A case clause begins with the keyword "case", followed by an integer expression, followed by a colon, followed by a sequence of statements, usually ending in a "break" statement.

When the switch statement executes, it evaluates the expression, and then executes the case clause whose expression matches that value. Thus, in this example, if the variable "value" contains 11, then "jack" will be printed.

IMPORTANT NOTE: the "break" statement is necessary to prevent the program from "falling through" and executing the *next* case clause as well! In sophisticated programs this technique is sometimes taken advantage of deliberately, but this is usually not intended and is a very common programming error.

If no case clause matches the value of the expression, then the "default:" clause will be executed. In this case, we have reason to believe no card's value should ever fall outside the range we have provided for, so we print an error message in the default clause.

To round out the function, we use a similar switch statement to print out the name of the suit.

Now consider the next function:

```
void deck_shuffle(int deck[]) {
```

Again, the first line of the function differs from the prototype only in the "{" that signals this is the beginning of the actual code for this function. deck_shuffle() accepts an array of cards to be shuffled and returns nothing.

Following the variable declarations, deck_shuffle() uses a simple for loop through the 52 cards (indexed from 0 to 51). At each card, it chooses another card at random from the deck, fetches it, and swaps it with the card at the "i"th position:

```
cpos = rand() % 52;
```

This statement fetches a random integer, then uses the "%" operator to take the remainder when dividing by 52. This will be a number between 0 and 51, and is the position of the card we are swapping the "i"th card with.

```
c = deck[i];
```

Here we fetch the "i"th card out of the array. Note the use of the "[]" operator to retrieve an individual integer from the array.

```
deck[i] = deck[cpos]; deck[cpos] = c;
```

Here we swap the cards, by assigning the card at position cpos to position i, and vice versa.

(If we had simply done the following:

```
deck[i] = deck[cpos]; deck[cpos] = deck[i];
```

then the original value of deck[i] would have been lost before the second statement could execute. This is why we store the value in the intermediate variable "c".)

Now consider deck_draw:

```
int deck_draw(int deck[], int *card_top) {
```

This function, as discussed earlier, accepts a deck of cards and the *location* of the variable which records the index of the top card (a "pointer" to that variable).

Consider the following line:

```
pos = *card_top;
```

Here we use the "*" operator, which in this context means "value at." Placing "*" in front of a pointer yields the actual value stored at the location pointed to. In this way we fetch the index of the top card and place it in the variable "pos" for our convenience.

Next we fetch the card at that position from the array:

```
c = deck[pos];
```
And then advance to the next card, using the "++" increment operator:
```
pos++;
```
The if statement that follows checks to see if we have reached index 52. Since the cards are indexed from 0 to 51 in the array, this means we have passed the last card; the bottom card has been dealt from the deck. Accordingly, we call deck_shuffle() to reshuffle the deck, and we set pos back to 0.

Finally, consider the following:
```
*card_top = pos;
```
Again, we use the "*" operator to access the "value at" card_top, and store our updated version of pos at that location. This updates card_top in the calling function, so it will remain changed when deck_draw() returns.

Finally, the deck_draw() function returns the value of c, the card it was originally asked to draw from the deck.

The function:
```
void deck_init(int deck[]) {
```
initializes the deck by setting the 52 elements of the array to ascending values. Recall that in C, variables are not automatically assigned any meaningful value. It is important to initialize the deck properly.

Finally, the function:
```
void deck_print(int deck[]) {
```
prints the name of each card in the deck. It does this by invoking card_print() on each card in a simple for loop.

Phew!

Now, go ahead and compile and run the program.

"The output scrolled off my screen!"

Well, yes. Since we print out the entire deck of cards twice, the screen is likely to scroll. If you are using a workstation or Macintosh, you'll have no problem using scrollbars to review the results. (The first printout of the deck should be unshuffled, the second shuffled.) If you are using a terminal on a Unix system, or an MSDOS system, or any other operating system that supports Unix-like pipes, and you have named your program "bj", try the following command at your operating system shell prompt:
```
bj | more
```
This will display the output a page at a time for your review.

FURTHER EXPLORATION

Try removing the srand() call in the main() function. Notice the effect this has on several consecutive runs of the program.

Experiment with other approaches to shuffling the deck in deck_shuffle(). See if you can find one that more realistically mimics a human dealer.

Add a printf() call to deck_shuffle() that alerts you that it has been called, and then change the for loop in main() to draw several hundred cards from the loop. Note that the deck reshuffles every 52 cards.

IN THE NEXT CHAPTER

We will continue the development of the program toward an actual game. Structures and character strings will be introduced.

IF YOU NEED HELP

Just contact me. I can help you directly or put you in touch with one of several Good Samaritans who have volunteered to assist the perplexed.

THANKS FOR THE EMAIL!

I've received gobs of email from those following the tutorial since my request at the end of Chapter 3. Thanks for sounding off, it's great to know there's an audience out there.