Tutorial: Programming in C

Chapter 5, 7/13/93

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!

Completing the Second Program: Blackjack, Structures and Strings

In the previous chapter we developed the functions needed to assemble a blackjack- playing program. Now we will complete the set and put together a complete game. Along the way two new important C features will be introduced.

The code for the complete second program follows. Many functions are the same; several new functions have been added. And of course the main() function is completely new.

```
--- #include  #include  #include
/* A new include file: fetches handy text string functions. */ #include
/* Blackjack, by Tom Boutell, 7/6/93 */
/* The maximum number of players (including the computerized dealer). */
#define PLAYERS_MAX 10
/* The maximum number of characters in a player's name. */
#define PLAYER_NAME_MAX_LEN 31
/* The maximum number of cards in a player's hand. */
#define HAND_CARDS_MAX 10
/* A player_struct structure holds all the information associated with a particular player. */
struct player_struct { char name[PLAYER_NAME_MAX_LEN+1]; int cards_total; int
hand[HAND_CARDS_MAX]; int games_won; int busted; };
/* Now we declare it to be a type, so we can conveniently declare variables containing player
information. */
typedef struct player_struct player_t; /* 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[]);
/* Return 1 if player has hit blackjack (21), false (0) if not. Pass address of player structure. */
int player_hand_blackjack(player_t *player);
```

```
/* Return 1 if player has busted, 0 if not. */
int player_hand_busted(player_t *player);
/* Return the minimum value of the player's hand (aces counted as 1). */
int player_hand_min_value(player_t *player);
/* Return the maximum value of the player's hand (aces counted as 11). */
int player_hand_max_value(player_t *player);
/* Read a line of input from standard input (stdin) and store it into the buffer s. Discard any
characters beyond max_len-1 (leaving the last space for a null terminating character). */
void line_read(char *s, int max_len);
/* Fetch an integer (followed by a carriage return) from standard input. Takes advantage of
line_read(). Takes no arguments. We replace scanf() with line_read() and int_read() because
scanf() does not respect carriage returns well, which can lead to serious confusion for the user of
the program. */
int int_read();
/* Add a card to the player's hand. */
void player_hand_add(player_t *player, int c);
/* Print out a player's status. */
void player_print(player_t *player);
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;
/* Flag: have we completed the loop yet? */ int done;
/* Flag: do the players want a new game? */ int new_game;
/* The number of players. */ int players_total;
/* The array of players. */
player_t players[PLAYERS_MAX];
/* Convenience pointer to the dealer. */ player_t *dealer = &players[0];
/* 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.
If you are using a non- ANSI compiler you may need to declare a variable of type time_t
("time_t t") and replace the line below with these two lines: "time(&t);" and "srand(t);". */
srand(time(0));
/* Find out how many players we will have. */ printf("The computer will be the dealer.\n"); done
= 0; /* Loop until we have an acceptable number */ while (!done) { printf("How many human
players (1-%d)? ", PLAYERS_MAX-1); players_total = int_read();
/* Make sure there is at least one human and at least one free slot for the dealer */ if
((players_total >= 1) && (players_total <= (PLAYERS_MAX-1))) { /* An acceptable number */
done = 1; } else { printf("That is not an acceptable number of human players.\n"); } }
/* Add one player to account for computerized dealer */ players_total++;
/* Set up dealer (player 0) */ strcpy(dealer->name, "HAL (Dealer)"); dealer->games_won = 0;
/* Now loop through players, initializing information and prompting for names. Human players
start at index 1. */
```

```
for (i=1; i /* Initialize the deck. */ deck_init(deck);
/* Now shuffle the deck. It need not be reshuffled before each hand (game) of blackjack; the
deck_draw routine automatically shuffles it when it is empty. Enjoy yourselves, card- counters.
*/
deck_shuffle(deck);
/* Now let the players take turns, the dealer always going last. */ /* Outer loop: games. */ do { /*
Flag */ int blackjack = 0;
/* Count of busted players */ int busted_total = 0;
/* # of player who hit blackjack */ int blackjack_player;
/* For each game we must clear the busted and cards_total fields. */ for (i=0; i /* Loop through
turns. */ /* A game is over when either: all but one player has busted, or any player has hit
blackjack. */ while ((busted_total < (players_total-1)) && (!blackjack)) { /* Human players who
stood this turn */ /* Loop through players and take turns. Save dealer for last (handle human
players for now). */ for (i=1; ibusted) { /* Go on to next player if player is busted. */ continue; }
player_print(p); printf("1. Hit\n"); printf("2. Stand\n"); printf("Your choice? "); choice =
int_read(); if (choice == 1) { int c; c = deck_draw(deck, &card_top); printf("You drew: ");
card_print(c); printf("\n"); player_hand_add(p, c); if (player_hand_blackjack(p)) { blackjack = 1;
blackjack_player = i; printf("BLACKJACK!\n"); /* Break out of loop of players */ break; } if
(player_hand_busted(p)) { p->busted = 1; busted_total++; printf("You BUSTED!\n"); } } else
{ /* Stand */ printf("You stand this turn.\n"); } } /* Now dealer's turn, but only if not busted and
only if blackjack hasn't been hit */ if ((!dealer->busted) && (!blackjack))
{ player_print(dealer); /* STRATEGY: If the maximum value of the dealer's hand is less than
18, OR the minimum value (with aces treated as 1) is less than 12, take a card. If all other players
have busted, do nothing (and win). */ if ((busted_total < (players_total - 1)) &&
((player_hand_max_value(dealer) < 18) || (player_hand_min_value(dealer) < 12))) { int c; c =
deck_draw(deck, &card_top); printf("Dealer drew: "); card_print(c); printf("\n");
player_hand_add(dealer, c); if (player_hand_busted(dealer)) { dealer->busted = 1; busted_total+
+; printf("Dealer BUSTED!\n"); } if (player_hand_blackjack(dealer)) { blackjack = 1;
blackjack_player = 0; printf("Dealer hits BLACKJACK!\n"); } } else { printf("Dealer stands.
\n"); } } } printf("The game is over.\n");
/* Who won? */ if (blackjack) { printf("%s won by hitting blackjack.\n",
players[blackjack_player].name); players[blackjack_player].games_won++; } else { for (i=0; i /*
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. */
printf("Shuffling\n"); 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"); } }
void line_read(char *s, int max_len) { int len; /* Now use fgets to read a line of input from the
user. scanf("%s") would be easier, but would not allow spaces. */ fgets(s, max_len, stdin); len =
strlen(s); if (s[len-1] == '\n') { s[len-1] = '\0'; } }
int int_read() { char s[10]; int i; line_read(s, 10); /* atoi(): a standard function which accepts a
string such as "123" and returns an integer such as 123. */ i = atoi(s); return i; }
int player_hand_blackjack(player_t *p) { int min_value; int aces; min_value =
player_hand_min_value(p); aces = player_hand_count_aces(p); if (min_value == 21) { /*
Regular cards total 21- all set */ return 1; } if (!aces) { /* Cards don't total 21 and no aces- can't
be blackjack */ return 0; } else { /* Try counting one ace as 11 instead of 1 by adding 10. No
```

more than one ace can be usefully counted as 11, since 2 would be 22, so even if we hold several aces we only need to check the result of holding one. */ if ((min_value + 10) == 21) { /* Blackjack with one ace counted as 11 */ return 1; } } /* No blackjack */ return 0; }
int player_hand_min_value(player_t *p) { int i; int value; int c; int score = 0; for (i=0; i<(p->cards_total); i++) { c = p->hand[i]; value = (c % 13) + 2; /* Take care of aces and face cards */ if (value == 14) { score += 1; } else if (value > 10) { score += 10; } else { score += value; } } return score; }
int player_hand_count_aces(player_t *p) { int i; int c; int aces = 0; for (i=0; i<(p->cards_total); i++) { int value; c = p->hand[i]; value = (c % 13) + 2; if (value == 14) { /* Increment count of aces */ aces++; } } return aces; }
int player_hand_max_value(player_t *p) { int score; /* Instead of repeating code, we'll take advantage of the functions already written to calculate the maximum value of the hand (counting aces as 11). */ score = player_hand_min_value(p);
/* Add 10 for each ace (we already added 1 for min_value). */ score += player_hand_count_aces(p) * 10; return score; }
int player_hand_busted(player_t *p) { /* Just return whether minimum value of hand is greater than 21 or not. */ return (player_hand_min_value(p) > 21); }
void player_hand_add(player_t *p, int c) { int i; i = p->cards_total; if (p->cards_total == HAND_CARDS_MAX) { fprintf(stderr, "Too many cards in hand-- maximum is %d\n", HAND_CARDS_MAX); exit(1); } p->hand[i] = c; p->cards_total++; }
void player_print(player_t *p) { int i; /* Flag indicating whether we are at the start of a line or not. We use this to print two cards on each line consistently. */ int nline = 1; printf("\nPlayer: %s Cards: %d Games won: %d\n", p->name, p->cards_total, p->games_won); for (i=0; icards_total; i++) { card_print(p->hand[i]); if (!nline) { printf("\n"); } else { printf(" "); } nline = !nline; } /* Final check, so we get a carriage return after an odd number of cards, and no extra one after an even number. */ if (!nline) { printf("\n"); } }

Once again we'll proceed line by line in examining the new features of the program.
First, note at the beginning the use of a new #include file:
#include
string.h contains routines for manipulating strings. A string, generally speaking, is a series of text characters. We have already used very simple strings in the form of quoted text, as in the first argument of the printf() function.
Strings, in C, differ greatly from their treatment in languages such as BASIC, where there is a distinct string type in the language. Their treatment in C is closer to standard dialects of PASCAL. (Some versions of PASCAL include BASIC- like strings as well.)
To understand C strings, first take note that there is a distinct variable type for characters in C. This type is known as char. A variable of type "char" is usually one byte in size (a value between 0 and 255). This varies in very rare cases (the Mix C/ Power C compiler for MSDOS uses two bytes to represent char), but the important thing is that char is big enough to store one character of text-- a space, letter, number, or other symbol.
char variables are actually numeric variables, just like int and float; you can store a number between -128 and 127 in a char variable (assuming you have byte- sized char, which, again, is almost universal). Each common text character has a standard value between 32 and 255, and certain special sequences (such as carriage return and line feed) lie between 0 and 31.

(Actually, this describes only one character set, the ASCII character set. ASCII is by far the most common, and is the set of character values found on all personal computers and practically all Unix systems. Another system, EBCDIC, appears on some mainframes.)

So you can store a single character in a char variable. Of course this is usually insufficient. What is needed is a way to store a series of characters, such as a line of text.

In C, this is done by creating what is known as a null- terminated string. A null- terminated string is a series of characters followed by a null, or nil, character, which is always equivalent to the value zero (0). From now on when I refer to a string, I am referring to a null- terminated string, unless I explicitly say otherwise.

Since a C string is a series of char values, it is generally referred to by a pointer to type char, or copied into an an array of type char.

Since strings themselves are not a type in C per se, a collection of functions to to conveniently manipulate them has been provided; these functions are brought in by string.h.

Now consider the following line:

```
#define PLAYERS_MAX 10
```

This is not a C statement! That may seem odd, since this is a C program, but your program is not read immediately by the C compiler. It is first read by a program (or a portion of your compiler program) referred to as the "C preprocessor."

What the preprocessor does is take care of special "directives" that begin with a "#" sign.

You have seen one already in the form of the "#include" directive, which causes the compiler to treat the included file as part of your program. (Do not infer from this that the code of your functions should be broken off into separate include files; include files are generally used for prototypes and other declarations only. Separating actual code into separate files is called multiple- module programming, and we will explore it soon.)

Another preprocessor directive is "#define". "#define" allows you to create a "macro"- a single word which represents a number or even a short block of code.

So "#define PLAYERS_MAX 10" makes PLAYERS_MAX a macro name that represents the number 10. When the preprocessor runs, it will replace PLAYERS_MAX with 10 wherever it encounters PLAYERS_MAX.

In this case, we define the macro in order to do two things:

-- Provide a meaningful name for the maximum number of players in the game, instead of an arbitrary number.

-- Make the number EASY TO CHANGE. Since we use the macro consistently throughout the code instead of many occurrences of 10, we can change the maximum number of players simply by changing the macro definition and recompiling. No muss, no fuss.

I highly recommend this use of macros for most "constants" and especially for those that are likely to change.

The macros:

```
#define PLAYER_NAME_MAX_LEN 31
```

and

```
#define HAND_CARDS_MAX 10
```

provide the same benefits for the length of player names and the number of cards that can be in a player's hand, respectively. If either turns out to be insufficient, it can be easily changed.

Now consider the following "structure definition":

```
struct player_struct { char name[PLAYER_NAME_MAX_LEN+1]; int cards_total; int hand[HAND_CARDS_MAX]; int games_won; int busted; };
```

It would be possible to write the complete game using only simple variables and arrays. But consider how many different arguments would have to be passed to functions in order to do this! A function which prints out information about a player would need to be passed the player's name, the number of cards the player has, the array of cards in that player's hand, and so on.

Instead, we take advantage of C's "struct" keyword to declare a "record type" which can be passed and copied as a whole, and its individual "fields" accessed when necessary.

A structure declaration consists of the word "struct", an (optional) name for the structure, and a list of variable declarations inside braces ("{" and "}"). The declaration is followed by a final semicolon. (Omitting this is a common error among new and old programmers.)

Now take a look at the first field's declaration:

```
char name[PLAYER_NAME_MAX_LEN+1];
```

Note the use of the "char" type described above. Here we are declaring an array of char large enough to store a player's name.

IMPORTANT: We must add one extra char of space in order to leave room for the NULL (0) character that terminates the string!

Now the structure has been described; but there is one more step necessary in order to turn it into a full- fledged C type. For this we use the "typedef" keyword:

```
typedef struct player_struct player_t;
```

A "typedef" statement begins with the keyword "typedef" and is followed by type declaration (such as a struct definition or an existing type like int, char or float), followed by the name we wish the newly defined type to have.

If we did not do this, we could still use player_struct, but we would have to declare each variable of that type as follows:

```
struct player_struct player;
```

By defining player_struct to be a type in its own right called player_t, we become able to use a much simpler declaration:

```
player_t player;
```

and so our player structure becomes a type in the same sense that char, int and float are types. We can even declare a pointer to a player_t, and in fact we will often do so, since structures are large and it is often "cheaper" to pass them by reference (address) rather than by copying them.

Again, typedef can be used to create simpler types as well. Consider the following:

```
typedef int card;
```

Had we used this tactic in the blackjack program, we would be able to declare variables of type "card". They would behave exactly as integers behave, but have the benefit of a more specific, readable name.

Now, skip past the functions we declared in the last chapter and take note of the following new prototype:

```
int player_hand_blackjack(player_t *player);
```

The purpose of this function is to determine whether a player has hit blackjack. Note that we pass a pointer to a player_t structure. This is the sort of highly convenient coding that makes structures so useful.

Why do we pass a pointer to the structure (note the "*") and not just the player_t itself? Because the structure is fairly large, and so copying it (remember, when you pass a value to a function, the function receives a copy) may not be a good idea, especially on systems where the "stack size" is limited, such as MSDOS systems. In general sizable structures should be passed by address (by way of a pointer) rather than directly.

The function will return 1 for true, or 0 for false, making its return value useful in an if statement.

The next prototype:

int player_hand_busted(player_t *player);

serves a similar purpose, but determines whether a player has busted (gone over 21 points even with aces counted as 1) rather than whether a player has hit blackjack.

Now consider the following pair of function prototypes:

int player_hand_min_value(player_t *player);

int player_hand_max_value(player_t *player);

Both of these functions return the value of a player's hand in points, but there is a subtle difference. Since, in Blackjack, numbered cards are worth face value, face cards (jack, queen, king) are worth 10 points, and aces are worth *either 1 or 11* points, the value of a hand differs depending on how you count the aces! So we provide two functions, one which returns the value of the hand with aces counted as 1, and another which returns the value of the hand with aces counted as 11. These functions together are useful in helping the computerized dealer arrive at a strategy.

The next two prototypes are of functions that perform input in a "safer", less sensitive fashion than the scanf() function. First we have:

void line_read(char *s, int max_len);

This function's purpose is to read a line of input (ending with the user striking carriage return), and copy the characters to the location pointed to by "s".

The argument, max_len, indicates the size of the buffer which s points to. line_read promises to copy up to max_len-1 characters into the location pointed to by s (even if the user entered more than that) and add a null character at the end (remember, C strings MUST end in a null character to behave properly!).

For convenience, we also provide:

int int_read();

This function reads a line of input also, converts the text the user entered to an integer, and returns the integer. It is similar to scanf("%d", &x), where x is the integer variable being read into, but not identical, because scanf() is sensitive to spaces; if the user enters "4 5" instead of "45", scanf() treats this as two separate numbers by scanf() and the *next* call to scanf() will get the 5, which is confusing to the user (particularly in a situation where we are prompting for one number and the user should not be entering two!). Even worse, scanf() does not get rid of the carriage return following the user's input, which confuses attempts to read a line of text elsewhere after scanf() has read in a value. For these reasons we have replaced scanf() for our purposes with this pair of functions. (We will return to scanf() and its relatives for other purposes, however.)

To complete our set of tools, consider the following two prototypes:

void player_hand_add(player_t *player, int c);

void player_print(player_t *player);

The first accepts a pointer to a player_t structure and an integer representing a card, and adds that card to the player's hand. (Note that the array of cards in the player's hand and the total number

of cards in the player's hand, which are needed to do this, are both within the player structure; so by passing one pointer we get access to everything we need to manipulate.)
The second simply prints out a player's status.
Now, turn your attention to the main() function, which has been completely replaced with a new version which actually plays Blackjack.
The function begins by declaring several simple variables. Move past these to examine the following declaration: player_t players[PLAYERS_MAX];
Here we are declaring an array of player_t structures. We can declare arrays of types we create using typedef in the same manner that we declare arrays of built-in types like int, float and char. Note the use of the PLAYERS_MAX macro, discussed earlier; the preprocessor replaces this with 10 (or whatever we have changed the value to).
Now consider the following: player_t *dealer = &players[0];
Here we declare a *pointer* to a player_t structure- a variable that will contain the location of a player_t structure, not the structure itself.
We initialize "dealer" to point at the first player in the array. Recall that "[" and "]" are used to index into an array, so players[0] yields the first player structure in the array. The "&" operator then yields the *location* of players[0]. Thus we set dealer to point (contain the location of) the first player, which will be the dealer.
(Note that we could have said the following:
player_t *dealer = players;
due to the fact that the name of an array, *by itself,* can be used as a pointer to its first element (element 0). But for clarity we have explicitly referred to element 0 and taken its address.)
We do this for convenience, to avoid referring to "players[0]" whenever we wish to refer to the dealer. (We could have used a macro instead, but the macro would be in effect throughout the file, so dealer could not have a different meaning in another function if we so desired.)
Next, we encouter a while loop which prompts the player for a number of human players between 1 and the maximum number minus one (to leave room for the dealer). The loop exits only when a number in that range has been entered, setting "done" to escape the loop.
Consider the "if" statement used to test whether the number of players is acceptable:
if ((players_total >= 1) && (players_total <= (PLAYERS_MAX-1))) { ... Actual code ... }
"&&" is a new operator which means "and". As long as the left and right arguments to "&&" are true (not zero), "&&" yields 1; otherwise it yields 0.
Next, we increment the number of players:
players_total++;
to allow for the dealer.
Finally, we begin initializing the information in each player_t structure. We begin with the dealer:
strcpy(dealer->name, "HAL (Dealer)");
the function strcpy() accepts two pointers to type char, and copies the null- terminated string pointed to by the SECOND into the space pointed to by the FIRST. (The order of such things can be confusing; keep your reference books handy, and take advantage of your online help, whenever you are in doubt. On Unix systems, use the command "man strcpy" to get information about strcpy and related functions.)
But what's going on in the first argument, "dealer->name"?
Recall that dealer is a pointer to a player_t structure. Structures contain one or more fields.

"->" is a new operator which accesses a field through a structure pointer.
That is, if you have a pointer p to a structure which contains two fields, a and b, you can refer to field b with the following expression:
p->b
So in the case of the strcpy() call, "->" is used to access the name field, which is an array of char. strcpy() expects to be passed a pointer to char, but fortunately, as mentioned earlier, the name of an array can be used as a pointer to its first element.
So, as a result of the call, "HAL (Dealer)" is copied into dealer->name.
Now consider the next line:
dealer->games_won = 0;
Here we have a simpler case of the use of the "->" operator, again accessing a field through the dealer pointer.
There is a corresponding operator "." which does exactly the same thing for cases in which you have the structure itself, rather than a pointer to it. For instance, the expression:
players[3].games_won++;
Would increment the number of games won by player 3 if placed in our main() function. (Note that we have used "[3]" to access structure number 3. players itself is an array of player_t structures, so we need to specify which structure we wish to access.)
Now consider the following for loop:
for (i=1; iHere we have the same basic goal as above, but this time we are initializing the human players, so we need to interact with the user. Note the call to our line_read() function; we pass players[i].name, which is an array of characters and is passed as an array to the first character in the arary. We also pass the maximum length of a player's name, to make sure that the user doesn't "run over" the space we have alloted for the name.
(Note that the for loop begins at element 1. This is because we have reserved element 0 for the dealer.)
After initializing and shuffling the deck using functions developed in the last chapter, we begin the actual game, or rather series of games. We enclose the game in a new kind of loop, called a "do" loop:
do { /* Flag */ int blackjack = 0;
... additional code ...
} while (new_game);
A "do" loop is just like a while loop, except that the condition comes at the end. This is convenient when we know we always want the loop to execute at least once.
A "do" statement is constructed as follows: the "do" keyword, followed by a statement (usually a compound statement between "{" and "}"), followed by the "while" keyword, followed by a conditional expression between parentheses, and completed by a final semicolon.
The statement (or series of statements between "{" and "}") following the "do" keyword is executed first. Then the condition following "while" is tested. As long as that condition remains true, the loop repeats.
(In this case, new_game is set by asking the user whether or not he or she wants to continue.)
Now, moving inside the loop, consider the following declaration:
/* Flag */ int blackjack = 0;
Earlier I stated that variables should be declared at the beginning of a function. It is also acceptable to declare variables at the beginning of any compound statement. Variable

declarations must follow the initial "{", and be completed before any other statements appear. It is often convenient to declare variables inside such blocks for clarity; since they are only used within the block, it is convenient to be able to see their declarations when editing the block.
Variables declared inside a block supersede those outside the block. In other words, if there is an "int i" declared at the start of main() and another "int i" declared inside a "for" loop, code within the "for" loop sees the latter variable's value.
Now consider the following loop:
for (i=0; iHere we initialize the cards_total and busted fields each player (including the dealer). These fields need to be reset each game (hand) of Blackjack. (Some other fields, such as name and games_won, are *not* reset for each game.)
Next, we encounter a "while" loop through the game:
while ((busted_total < (players_total-1)) && (!blackjack)) { ... Additional code ... };
The conditional expression here is a bit complex, but it states the conditions under which a game should continue for another turn. In English:
-- At least two players have not "busted" (gone over 21 points and lost the game), and
-- No player has hit Blackjack (21 points).
As long as these conditions are true (which they are at the beginning of a game), the game continues for another round.
Now, consider the following loop:
for (i=1; ibusted) { continue; } ... Additional code ... }
Here we loop through the human players (again, human players begin at index 1). Just as we did for the dealer, we declare a convenience pointer and assign it the location of the current player. This saves a great deal of typing later on.
Note the if statement, which checks to see if a player is busted and, if so, skips to the next player by way of the "continue" statement.
The "continue" statement, when encountered inside any loop, skips over the remainder of the code inside the innermost loop and goes on to the next pass through the loop (if any). If several "for", "while", or "do" loops have been "nested", the innermost loop executing is the one affected.
Now consider the following:
player_print(p); printf("1. Hit\n"); printf("2. Stand\n"); printf("Your choice? "); choice = int_read();
Here we invoke the player_print function, passing it the location of the current player.
Next, we prompt the user for a choice: whether to "hit" (take a new card) or "stand" (take no card this turn). We take advantage of the new int_read() function here.
After this, we use an if statement to see whether the user has elected to "hit." If so, we execute the following code:
int c; c = deck_draw(deck, &card_top); printf("You drew: "); card_print(c); printf("\n"); player_hand_add(p, c);
(Again, we declare a new variable, which is acceptable at the beginning of the block.)
Here we draw a card from the deck, as in the previous chapter, and display it to the user. We then invoke player_hand_add to add the card to the player's hand.
Now consider the following:
if (player_hand_blackjack(p)) { blackjack = 1; blackjack_player = i; printf("BLACKJACK!\n"); break; }

Here we invoke the player_hand_blackjack() function to find out if the player has won the game by hitting 21 exactly; if so, we set the "blackjack" flag to 1 (true), and set the blackjack_player variable to indicate the index of the winning player. We then use the "break" statement to escape the for loop, since the game is over at this point.

The break statement is similar to the continue statement, except that it escapes the innermost loop entirely, instead of skipping to the next iteration (pass through the loop).

We then do nearly the same thing for the case in which the player has "busted" (gone over 21 points), except that here we do not break out of the loop, since the other players continue when one player goes bust.

Following this we encounter the "else" clause of the if statement which determined whether the player was taking a "hit" or a "stand". In the case of a stand we simply print a message to that effect.

Next we encounter the code for the computerized dealer's turn. This code is very similar to the code for the human players.

First, we encounter a test to see whether the computer should move at all:

if ((!dealer->busted) && (!blackjack)) { ... dealer code ... }

If the dealer has *not* busted, and no one has hit blackjack, the dealer moves.

The interesting part of the dealer code lies in the following "if" statement, which encodes the dealer's strategy for deciding whether to "hit" or "stand":

if ((busted_total < (players_total - 1)) && ((player_hand_max_value(dealer) < 18) || (player_hand_min_value(dealer) < 12))) { ... Dealer takes a hit ... }

We see another new operator introduced here, the "||" operator (two vertical lines), which means "or". If either of its left and right arguments is not zero, it yields 1; otherwise it yields 0 (false).

Here is an English translation:

Take a new card (hit) if:

-- At least one other player has not busted (we haven't already won!), and

EITHER

-- the maximum value of our hand (aces counted as 11) is less than 18, or

-- the minimum value of our hand (aces counted as 1) is less than 12.

Thus, assuming the dealer hasn't already won, the dealer will hit if he either has a poor hand (worth less than 18) or can comfortably risk a hit (worth less than 12, so even a face card won't bust).

(Not being a professional dealer, I can't say whether this is the best strategy in the world, but it beats the pants off me most of the time!)

The remainder of the code, which handles the dealer's drawing of a card (hit), is similar to that for human players.

After the outer while() loop controlling the game has exited, the next block of code determines who won the game. If a player hit blackjack, this is simple.

if (blackjack) { printf("%s won by hitting blackjack.\n", players[blackjack_player].name); players[blackjack_player].games_won++; }

Note the use of the "%s" sequence in our call to the printf() function. This sequence outputs a string argument, just as "%d" and "%f" output numeric arguments. Thus the name of the player is substituted into the sentence.

Now consider the else clause (in which no player hit blackjack). Here, we know that whoever did *not* bust (go over 21) must be the winner!

else { for (i=0; iAfter determining the victor, we print out the current standings in a simple for loop, and then ask the player(s) whether they wish to continue. If not, the program returns to the operating system.
Now we will consider the additional functions added to break up the code:
void line_read(char *s, int max_len) { ... }
This function reads a line of input from the user, ending in a carriage return, and copies up to max_len-1 characters into the space pointed to by s. It then adds a terminating null.
Consider the following line:
fgets(s, max_len, stdin);
Here we call the function fgets(), which is part of the standard I/O package brought in by stdio.h. fgets accepts three arguments: a pointer to char (s in this case), a maximum number of characters (of which it reserves the last for a terminating null), and a "stream."
Streams are variables of type "FILE *". They are used to access files on disk, and also to communicate with other devices. In this case, we are using the stream "stdin".
"stdin" is a global variable, accessible to any program which includes stdio.h. It refers to the stream of input from the user.
(There is a gets() function which always uses stdin without the need to state it explicitly, but unfortunately it is unsafe, because it does not allow you to specify a maximum number of characters! Thus the user can enter more characters than you provided space for and crash your program in unpredictable, usually awful ways.)
We will explore streams (and files) further in the next chapter.
Now consider the following:
len = strlen(s);
Here we invoke the function strlen(), provided by string.h, in order to fetch the length of the string pointed to by s. (The length of the string is the number of characters *before* the terminating null, and does not include the null itself.)
We have one last task to take care of: fgets() stores the carriage return itself in the string, so we need to remove it. Consider the following code:
if (s[len-1] == '\n') { s[len-1] = '\0'; }
Here we access the last element in the string (again, not including the null), test to see if it is a carriage return, and, if so, set it to a null, ending the string at that point to get rid of the carriage return.
"But s is a pointer, not an array!"
Yes, but we can use "[n]" after a pointer name to refer to the nth item following the location pointed to. Thus:
s[0]
is equivalent to
*s
in that it accesses the value of the first (0th) item at the location s points to.
"So pointers and arrays are the same?"
Not quite. There are significant differences, which will be discussed in later chapters. But whenever an array is passed to a function, it "decays" into a pointer to the same type. In fact, when we declare functions like this:
int total(int x[]);
x[] is only a synonym for this:

int total(int *x);
"OK, so what's this '\n' business?"
As previously mentioned, double- quotes ("") are used to code entire strings in C, and the compiler will null- terminate them.
Single- quotes, on the other hand, encode "character constants", or single characters. So the following:
char x = 'z';
sets the char variable x equal to the numeric representation of the character 'z'.
"Right. So why the \n?"
Recall that \n means "carriage return" in a printf statement's first argument (or in any other string for that matter). It can also be used as a single character. Here we need to test the *single character* at the end of the string, so we compare it to the character constant '\n' (carriage return):
s[len-1] == '\n'
tests whether the last character before the null is a carriage return, and
s[len-1] = '\0';
sets it to a null if so, shortening the string by one character as far as functions that examine strings (like printf()) are concerned.
After taking care of this nuance, we return to the calling function.
The function int_read() contains just one new item of note:
i = atoi(s);
The function atoi() accepts a pointer to char as an argument, reads the null- terminated string at that location, and returns an equivalent integer. So atoi("5") returns the integer 5.
The remaining functions break no new ground, and so I will not discuss them in detail here. The sole exception is tucked inside player_hand_add():
fprintf(stderr, "Too many cards in hand-- maximum is %d\n", HAND_CARDS_MAX); exit(1);
This code is executed when too many cards have been added to the player's hand. In this case, an error message is printed, using the function fprintf().
fprintf is a close relative of printf; the only difference is that it takes a stream (like stdin, discussed earlier, but an output stream in this case) as its first argument. In this case we pass the stream "stderr", or standard error, which is used to report internal errors in the program. (The regular printf() function is equivalent to calling fprintf with a first argument of "stdout", the standard output stream.)
The exit() call that follows is interesting because it EXITS THE PROGRAM IMMEDIATELY, returning the result it is passed (1, not 0, since something has gone wrong).
This sort of error- detecting code is important because it draws the user's attention, or better yet your own attention, before another user sees the program. If the error were to be quietly ignored the problem might not be detected until it became serious. (Of course, this is only a blackjack game, but more serious programs need such precautions.)
PHEW!
Compile and run the game. Grab a few friends if you like, it supports plenty of players.
FURTHER EXPLORATION
Try altering the dealer's strategy.
Introduce betting into the game. You can even implement loans from the house by integrating parts of the previous program if you're feeling particularly cruel.

Try to break up the main() function further into even more sub- functions.

Right now it is possible for the dealer and the human players to wind up standing turn after turn without an end to the game. I do not recall how the official rules of Blackjack resolve this. If you know the answer, implement it!

IN THE NEXT CHAPTER

We will return to a practical application, introduce global variables, and access files from within our programs. Numerous practical applications will be within our reach after the next chapter.

IF YOU NEED HELP

Contact me and I'll either assist you directly or put you in touch with one of several Good Samaritans who have signed up to assist new programmers. If your email address is unusual, try to give me the best possible address so I'll have a good chance of contacting you successfully; I had difficulty reaching one student with a question about pointer declarations.