6.x — Chapter 6 comprehensive quiz

BY ALEX ON OCTOBER 5TH, 2015 | LAST MODIFIED BY ALEX ON MAY 10TH, 2018

Words of encouragement

Congratulations on reaching the end of the longest chapter in the tutorials! Unless you have previous programming experience, this chapter was probably the most challenging one so far. If you made it this far, you're doing great!

The good news is that the next chapter is easy in comparison. And in the chapter beyond that, we reach the heart of the tutorials: Object-oriented programming!

Chapter summary

Arrays allow us to store and access many variables of the same type through a single identifier. Array elements can be accessed using the subscript operator ([]). Be careful not to index an array out of the array's range. Arrays can be initialized using an initializer list or uniform initialization (in C++11).

Fixed arrays must have a length that is set at compile time. Fixed arrays will usually decay into a pointer when evaluated or passed to a function.

Loops can be used to iterate through an array. Beware of off-by-one errors, so you don't iterate off the end of your array. For-each loops are useful when the array hasn't decayed into a pointer.

Arrays can be made multidimensional by using multiple indices.

Arrays can be used to do C-style strings. You should generally avoid these and use std::string instead.

Pointers are variables that store the memory address of (point at) another variable. The address-of operator (&) can be used to get the address of a variable. The dereference operator (*) can be used to get the value that a pointer points at.

A null pointer is a pointer that is not pointing at anything. Pointers can be made null by initializing or assigning the value 0 (or in C++11, nullptr) to them. Avoid the NULL macro. Dereferencing a null pointer can cause bad things to happen. Deleting a null pointer is okay (it doesn't do anything).

A pointer to an array doesn't know how large the array they are pointing to is. This means sizeof() and for-each loops won't work.

The new and delete operators can be used to dynamically allocate memory for a pointer variable or array. Although it's unlikely to happen, operator new can fail if the operating system runs out of memory, so make sure to check whether new returned a null pointer.

Make sure to use the array delete (delete[]) when deleting an array. Pointers pointing to deallocated memory are called dangling pointers. Dereferencing a dangling pointer can cause bad things to happen.

Failing to delete dynamically allocated memory can result in memory leaks when the last pointer to that memory goes out of scope.

Normal variables are allocated from limited memory called the stack. Dynamically allocated variables are allocated from a general pool of memory called the heap.

A pointer to a const value treats the value it is pointing to as const.

```
1 int value = 5;
2 const int *ptr = &value; // this is okay, ptr is pointing to a "const int"
```

A const pointer is a pointer whose value can not be changed after initialization.

```
1 int value = 5;
2 int *const ptr = &value;
```

A reference is an alias to another variable. References are declared using an ampersand, but this does not mean address-of in this context. References are implicitly const -- they must be initialized with a value, and a new value can not be assigned to them. References can be used to prevent copies from being made when passing data to or from a function.

The member selection operator (->) can be used to select a member from a pointer to a struct. It combines both a dereference and normal member access (.).

Void pointers are pointers that can point to any type of data. They can not be dereferenced directly. You can use static_cast to convert them back to their original pointer type. It's up to you to remember what type they originally were.

Pointers to pointers allow us to create a pointer that points to another pointer.

std::array provides all of the functionality of C++ built-in arrays (and more) in a form that won't decay into a pointer. These should generally be preferred over built-in fixed arrays.

std::vector provides dynamic array functionality that handles its own memory management and remember their size. These should generally be favored over built-in dynamic arrays.

Quiz time

1) Pretend you're writing a game where the player can hold 3 types of items: health potions, torches, and arrows. Create an enum to identify the different types of items, and a fixed array to store the number of each item the player is carrying (use built-in fixed arrays, not std::array). The player should start with 2 health potions, 5 torches, and 10 arrows. Write a function called countTotalItems() that returns how many items the player has in total. Have your main() function print the output of countTotalItems().

Hide Solution

```
#include "stdafx.h"
#include <iostream>

enum ItemTypes
{
    ITEM_HEALTH_POTION,
    ITEM_TORCH,
    ITEM_ARROW,
    MAX_ITEMS
};

int countTotalItems(int *items) // we don't need to pass the size because all item arrays should be of
 length MAX_ITEMS
{
    int totalItems = 0;
    for (int index = 0; index < MAX_ITEMS; ++index)
        totalItems += items[index];

    return totalItems;
}

int main()
{
    int items[MAX_ITEMS]{ 2, 5, 10 }; // Use uniform initialization to set initial number of items play
er has (C++11)
//  int items[MAX_ITEMS] = { 2, 5, 10 }; // if not using C++11, use an initializer list instead

    std::cout << "The player has " << countTotalItems(items) << " items in total.\n";

    return 0;
}
```

2) Write the following program: Create a struct that holds a student's first name and grade (on a scale of 0-100). Ask the user how many students they want to enter. Dynamically allocate an array to hold all of the students. Then prompt the user for each name and grade. Once the user has entered all the names and grade pairs, sort the list by grade (highest first). Then print all the names and grades in sorted order.

For the following input:

```
Joe
82
Terry
73
```

```
Ralph
4
Alex
94
Mark
88
```

The output should look like this:

```
Alex got a grade of 94
Mark got a grade of 88
Joe got a grade of 82
Terry got a grade of 73
Ralph got a grade of 4
```

Hint: You can modify the selection sort algorithm from lesson **6.4 -- Sorting an array using selection sort** to sort your dynamic array. If you put this inside its own function, the array should be passed by address (as a pointer).

Hide Solution

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <utility> // include <algorithm> instead if not C++11

struct Student
{
    std::string name;
    int grade;
};

// Function to sort our students.
// Since students is a pointer to an array and doesn't know its length, we need to pass in the length e
xplicitly
void sortNames(Student *students, int length)
{
    // Step through each element of the array
    for (int startIndex = 0; startIndex < length; ++startIndex)
    {
        // largestIndex is the index of the largest element we've encountered so far.
        int largestIndex = startIndex;

        // Look for largest element in the remaining array (starting at startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
        {
            // If the current element is larger than our previously found smallest
            if (students[currentIndex].grade > students[largestIndex].grade)
                // This is the new largest number for this iteration
                largestIndex = currentIndex;
        }

        // Swap our start element with our largest element
        std::swap(students[startIndex], students[largestIndex]);
    }
}

int main()
{
    int numStudents = 0;
    do
    {
        std::cout << "How many students do you want to enter? ";
        std::cin >> numStudents;
```

```
    } while (numStudents <= 1);

    // Allocate an array to hold the names
    Student *students = new Student[numStudents];

    // Read in all the students
    for (int index = 0; index < numStudents; ++index)
    {
        std::cout << "Enter name #" << index + 1 << ": ";
        std::cin >> students[index].name;
        std::cout << "Enter grade #" << index + 1 << ": ";
        std::cin >> students[index].grade;
    }

    // Sort the names
    sortNames(students, numStudents);

    // Print out all the names
    for (int index = 0; index < numStudents; ++index)
        std::cout << students[index].name << " got a grade of " << students[index].grade << "\n";

    // Don't forget to deallocate the memory
    delete[] students;

    return 0;
}
```

3) Write your own function to swap the value of two integer variables. Write a main() function to test it.

Hint: Use reference parameters

Hide Solution

```
#include "stdafx.h"
#include <iostream>

// Use reference parameters so we can modify the values of the arguments passed in
void swap(int &a, int &b)
{
    // Temporarily save value of a
    int temp = a;
    // Put value of b in a
    a = b;
    // Put previous value of a in b
    b = temp;
}

int main()
{
    int a = 6;
    int b = 8;
    swap(a, b);

    if (a == 8 && b == 6)
        std::cout << "It works!";
    else
        std::cout << "It's broken!";

    return 0;
}
```

4) Write a function to print a C-style string character by character. Use a pointer to step through each character of the string and print that character. Stop when you hit a null terminator. Write a main function that tests the function with the string literal “Hello, world!”.

Hint: Use the ++ operator to advance the pointer to the next character.

Hide Solution

```
#include "stdafx.h"
#include <iostream>

// str will point to the first letter of the C-style string.
// Note that str points to a const char, so we can not change the values it points to.
// However, we can point str at something else.  This does _not_ change the value of the argument.
void printCString(const char *str)
{
    // While we haven't encountered a null terminator
    while (*str != '\0')
    {
        // print the current character
        std::cout << *str;

        // and point str at the next character
        ++str;
    }
}

int main()
{
    printCString("Hello world!");

    return 0;
}
```

5) What's wrong with each of these snippets, and how would you fix it?

a)

```
int main()
{
    int array[5] { 0, 1, 2, 3 };
    for (int count = 0; count <= 5; ++count)
        std::cout << array[count] << " ";

    return 0;
}
```

Hide Solution

The loop has an off-by-one error, and tries to access the array element with index 5, which does not exist. The conditional in the for loop should use < instead of <=.

b)

```
int main()
{
    int x = 5;
    int y = 7;

    const int *ptr = &x;
    std::cout << *ptr;
    *ptr = 6;
    std::cout << *ptr;
    ptr = &y;
    std::cout << *ptr;

    return 0;
}
```

Hide Solution

ptr is a pointer to a const int. You can't assign the value 6 to it. You can fix this by making ptr non-const.

c)

```
void printArray(int array[])
```

```
{
    for (const int &element : array)
        std::cout << element << ' ';
}


int main()
{
    int array[] { 9, 7, 5, 3, 1 };
    printArray(array);

    return 0;
}
```

Hide Solution

array decays to a pointer when it is passed to function printArray(). For-each loops can't work with a pointer to an array because the size of the array isn't known. One solution is to add a length parameter to function printArray(), and use a normal for loop. Another solution is to use std::array instead of built-in fixed arrays.

d)

```
int* allocateArray(const int length)
{
    int temp[length];
    return temp;
}
```

Hide Solution

temp is a fixed array, but length is not a compile-time constant. Variable temp will also go out of scope at the end of the function, so the return value will be pointing to something invalid. temp should use dynamic memory allocation.

e)

```
int main()
{
    double d(5.5);
    int *ptr = &d;
    std::cout << ptr;

    return 0;
}
```

Hide Solution

You can't assign an int pointer to point at a non-int variable. ptr should be of type double*.

6) Let's pretend we're writing a card game.

6a) A deck of cards has 52 unique cards (13 card ranks of 4 suits). Create enumerations for the card ranks (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace) and suits (clubs, diamonds, hearts, spades).

Hide Solution

```
enum CardSuit
{
    SUIT_CLUB,
    SUIT_DIAMOND,
    SUIT_HEART,
    SUIT_SPADE,
    MAX_SUITS
};

enum CardRank
{
    RANK_2,
```

```
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_9,
    RANK_10,
    RANK_JACK,
    RANK_QUEEN,
    RANK_KING,
    RANK_ACE,
    MAX_RANKS
};
```

6b) Each card will be represented by a struct named Card that contains a rank and a suit. Create the struct.

Hide Solution

```
struct Card
{
    CardRank rank;
    CardSuit suit;
};
```

6c) Create a printCard() function that takes a const Card reference as a parameter and prints the card rank and value as a 2-letter code (e.g. the jack of spades would print as JS).

Hide Solution

```
void printCard(const Card &card)
{
    switch (card.rank)
    {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
    }

    switch (card.suit)
    {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
    }
}
```

6d) A deck of cards has 52 cards. Create an array (using std::array) to represent the deck of cards, and initialize it with one of each card.

Hint: Use static_cast if you need to convert an integer into an enumerated type.

Hide Solution

```
int main()
{

```

```
    std::array<Card, 52> deck;

    int card = 0;
    for (int suit = 0; suit < MAX_SUITS; ++suit)
    for (int rank = 0; rank < MAX_RANKS; ++rank)
    {
        deck[card].suit = static_cast<CardSuit>(suit);
        deck[card].rank = static_cast<CardRank>(rank);
        ++card;
    }

    return 0;
}
```

6e) Write a function named printDeck() that takes the deck as a const reference parameter and prints the values in the deck. Use a for-each loop.

Hide Solution

```
void printDeck(const std::array<Card, 52> &deck)
{
    for (const auto &card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

    std::cout << '\n';
}
```

6f) Write a swapCard function that takes two Cards and swaps their values.

Hide Solution

```
void swapCard(Card &a, Card &b)
{
    Card temp = a;
    a = b;
    b = temp;
}
```

6g) Write a function to shuffle the deck of cards called shuffleDeck(). To do this, use a for loop to step through each element of your array. Pick a random number between 1 and 52, and call swapCard with the current card and the card picked at random. Update your main function to shuffle the deck and print out the shuffled deck.

Hint: Review lesson **5.9 -- Random number generation** for help with random numbers.
Hint: Don't forget to call srand() at the top of your main function.
Hint: If you're using Visual Studio, don't forget to call rand() once before using rand.

Hide Solution

```
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()

// Generate a random number between min and max (inclusive)
// Assumes srand() has already been called
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static used for eff
iciency, so we only calculate this value once
    // evenly distribute the random number across our range
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

void shuffleDeck(std::array<Card, 52> &deck)
{
    // Step through each card in the deck
```

```
    for (int index = 0; index < 52; ++index)
    {
        // Pick a random card, any card
        int swapIndex = getRandomNumber(0, 51);
        // Swap it with the current card
        swapCard(deck[index], deck[swapIndex]);
    }
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    rand(); // If using Visual Studio, discard first random value

    std::array<Card, 52> deck;

    // We could initialize each card individually, but that would be a pain.  Let's use a loop.
    int card = 0;
    for (int suit = 0; suit < MAX_SUITS; ++suit)
    for (int rank = 0; rank < MAX_RANKS; ++rank)
    {
        deck[card].suit = static_cast<CardSuit>(suit);
        deck[card].rank = static_cast<CardRank>(rank);
        ++card;
    }

    printDeck(deck);

    shuffleDeck(deck);

    printDeck(deck);

    return 0;
}
```

6h) Write a function named getCardValue() that returns the value of a Card (e.g. a 2 is worth 2, a ten, jack, queen, or king is worth 10. Assume an Ace is worth 11).

Hide Solution

```
int getCardValue(const Card &card)
{
    switch (card.rank)
    {
    case RANK_2:        return 2;
    case RANK_3:        return 3;
    case RANK_4:        return 4;
    case RANK_5:        return 5;
    case RANK_6:        return 6;
    case RANK_7:        return 7;
    case RANK_8:        return 8;
    case RANK_9:        return 9;
    case RANK_10:       return 10;
    case RANK_JACK:     return 10;
    case RANK_QUEEN:    return 10;
    case RANK_KING:     return 10;
    case RANK_ACE:      return 11;
    }

    return 0;
}
```

7) Alright, challenge time! Let's write a simplified version of Blackjack. If you're not already familiar with Blackjack, the Wikipedia article for **Blackjack** has a summary.

Here are the rules for our version of Blackjack:
* The dealer gets one card to start (in real life, the dealer gets two, but one is face down so it doesn't matter at this point).

* The player gets two cards to start.
* The player goes first.
* A player can repeatedly “hit” or “stand”.
* If the player “stands”, their turn is over, and their score is calculated based on the cards they have been dealt.
* If the player “hits”, they get another card and the value of that card is added to their total score.
* An ace normally counts as a 1 or an 11 (whichever is better for the total score). For simplicity, we’ll count it as an 11 here.
* If the player goes over a score of 21, they bust and lose immediately.
* The dealer goes after the player.
* The dealer repeatedly draws until they reach a score of 17 or more, at which point they stand.
* If the dealer goes over a score of 21, they bust and the player wins immediately.
* Otherwise, if the player has a higher score than the dealer, the player wins. Otherwise, the player loses (we’ll consider ties as dealer wins for simplicity).

In our simplified version of Blackjack, we’re not going to keep track of which specific cards the player and the dealer have been dealt. We’ll only track the sum of the values of the cards they have been dealt for the player and dealer. This keeps things simpler.

Start with the code you wrote in quiz #6. Create a function named playBlackjack() that returns true if the player wins, and false if they lose. This function should:
* Accept a shuffled deck of cards as a parameter.
* Initialize a pointer to the first Card named cardPtr. This will be used to deal out cards from the deck (see the hint below).
* Create two integers to hold the player’s and dealer’s total score so far.
* Implement Blackjack as defined above.

Hint: The easiest way to deal cards from the deck is to keep a pointer to the next card in the deck that will be dealt out. Whenever we need to deal a card, we get the value of the current card, and then advance the pointer to point at the next card. This can be done in one operation:

```
getCardValue(*cardPtr++);
```

This returns the current card’s value (which can then be added to the player or dealer’s total), and advances cardPtr to the next card.

Also write a main() function that plays a single game of Blackjack.

Hide Solution

```
#include <iostream>
#include <array>
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()

enum CardSuit
{
    SUIT_CLUB,
    SUIT_DIAMOND,
    SUIT_HEART,
    SUIT_SPADE,
    MAX_SUITS
};

enum CardRank
{
    RANK_2,
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_9,
    RANK_10,
    RANK_JACK,
    RANK_QUEEN,
    RANK_KING,
    RANK_ACE,
    MAX_RANKS
```

```
};

struct Card
{
    CardRank rank;
    CardSuit suit;
};

void printCard(const Card &card)
{
    switch (card.rank)
    {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
    }

    switch (card.suit)
    {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
    }
}

void printDeck(const std::array<Card, 52> &deck)
{
    for (const auto &card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

    std::cout << '\n';
}

void swapCard(Card &a, Card &b)
{
    Card temp = a;
    a = b;
    b = temp;
}

// Generate a random number between min and max (inclusive)
// Assumes srand() has already been called
int getRandomNumber(int min, int max)
{
    static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static used for ef
ficiency, so we only calculate this value once
    // evenly distribute the random number across our range
    return static_cast<int>(rand() * fraction * (max - min + 1) + min);
}

void shuffleDeck(std::array<Card, 52> &deck)
{
    // Step through each card in the deck
```

```
    for (int index = 0; index < 52; ++index)
    {
        // Pick a random card, any card
        int swapIndex = getRandomNumber(0, 51);
        // Swap it with the current card
        swapCard(deck[index], deck[swapIndex]);
    }
}

int getCardValue(const Card &card)
{
    switch (card.rank)
    {
    case RANK_2:         return 2;
    case RANK_3:         return 3;
    case RANK_4:         return 4;
    case RANK_5:         return 5;
    case RANK_6:         return 6;
    case RANK_7:         return 7;
    case RANK_8:         return 8;
    case RANK_9:         return 9;
    case RANK_10:        return 10;
    case RANK_JACK:      return 10;
    case RANK_QUEEN:     return 10;
    case RANK_KING:      return 10;
    case RANK_ACE:       return 11;
    }

    return 0;
}

char getPlayerChoice()
{
    std::cout << "(h) to hit, or (s) to stand: ";
    char choice;
    do
    {
        std::cin >> choice;
    } while (choice != 'h' && choice != 's');

    return choice;
}

bool playBlackjack(const std::array<Card, 52> &deck)
{
    // Set up the initial game state
    const Card *cardPtr = &deck[0];

    int playerTotal = 0;
    int dealerTotal = 0;

    // Deal the dealer one card
    dealerTotal += getCardValue(*cardPtr++);
    std::cout << "The dealer is showing: " << dealerTotal << '\n';

    // Deal the player two cards
    playerTotal += getCardValue(*cardPtr++);
    playerTotal += getCardValue(*cardPtr++);

    // Player goes first
    while (true) // infinite loop (until we break or return)
    {
        std::cout << "You have: " << playerTotal << '\n';

        // See if the player has busted
        if (playerTotal > 21)
            return false;
```

```
165
166         char choice = getPlayerChoice();
167         if (choice == 's')
168             break;
169
170         playerTotal += getCardValue(*cardPtr++);
171     }
172
173     // If player hasn't busted, dealer goes until he has at least 17 points
174     while (dealerTotal < 17)
175     {
176         dealerTotal += getCardValue(*cardPtr++);
177         std::cout << "The dealer now has: " << dealerTotal << '\n';
178     }
179
180     // If dealer busted, player wins
181     if (dealerTotal > 21)
182         return true;
183
184     return (playerTotal > dealerTotal);
185 }
186
187 int main()
188 {
189     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
190     rand(); // If using Visual Studio, discard first random value
191
192     std::array<Card, 52> deck;
193
194     // We could initialize each card individually, but that would be a pain.  Let's use a loop.
195     int card = 0;
196     for (int suit = 0; suit < MAX_SUITS; ++suit)
197     for (int rank = 0; rank < MAX_RANKS; ++rank)
198     {
199         deck[card].suit = static_cast<CardSuit>(suit);
200         deck[card].rank = static_cast<CardRank>(rank);
201         ++card;
202     }
203
204     shuffleDeck(deck);
205
206     if (playBlackjack(deck))
207         std::cout << "You win!\n";
208     else
209         std::cout << "You lose!\n";
210
211     return 0;
}
```

7a) Extra credit: Critical thinking time: Describe how you could modify the above program to handle the case where aces can be equal to 1 or 11.

Hint: It's important to note that we're only keeping track of the sum of the cards, not which specific cards the user has.

Hide Solution

One way would be to keep track of how many aces the player and the dealer got dealt (separately, as an integer). If either the player or dealer go over 21 and their ace counter is greater than zero, you can reduce their score by 10 (convert an ace from 11 points to 1 point) and "remove" one from the ace counter. This can be done as many times as needed until the ace counter reaches zero.

7b) In actual blackjack, if the player and dealer have the same score (and the player has not gone bust), the result is a tie and neither wins. Describe how you'd modify the above program to account for this.

Hide Solution

playBlackjack() currently returns true if the player wins and false otherwise. We'll need to update this function to return three possibilities: Dealer win, Player win, tie. The best way to do this would be to define an enumeration for these three options, and have

the function return the appropriate enum value:

```
enum BlackjackResult
{
    BLACKJACK_PLAYER_WIN,
    BLACKJACK_DEALER_WIN,
    BLACKJACK_TIE
};

BlackjackResult playBlackjack(const std::array<Card, 52> &deck);
```



7.1 -- Function parameters and arguments


Index


6.16 -- An introduction to std::vector

Share this:

C++ TUTORIAL | PRINT THIS POST

420 comments to 6.x — Chapter 6 comprehensive quiz

« Older Comments 1 … 4 5 6

Clint
July 15, 2018 at 11:21 pm · Reply

Hi!

New student here, haven't done any coding since Apple Basic on the Apple IIe back when I was a kid. Thanks very much for the work put into this site! I am finding it most helpful. I fell down a bit of a hole on the 2nd question of this quiz, figuring that I wanted to use a vector, but quickly realizing I didn't really know everything I needed. Nevertheless, after some searching, I have come up with the following, which at least on my computer does what it's supposed to. Any suggestions for improvements? Thanks.

```
#include <iostream>
#include <string>
#include <vector>

/*
Write the following program: Create a struct that holds a student's first name and grade (on a scale
 of 0-100).
Ask the user how many students they want to enter.
Dynamically allocate an array to hold all of the students.
Then prompt the user for each name and grade.
Once the user has entered all the names and grade pairs, sort the list by grade (highest first).
Then print all the names and grades in sorted order.
*/

struct Student
{
    std::string name;
```