Поддержка модульности. Написание игры Blackjack

Единицы компиляции. Разделение на файлы заголовков и реализации. Директивы препроцессора. Макросы и условная компиляция. Написание игры Blackjack.

[Термины](#_gjdgxs)

[От исходного кода к исполняемому модулю](#_30j0zll)

[Разделение текста программы на модули](#_1fob9te)

[Интерфейс и реализация](#_3znysh7)

[Практический пример](#_yyn9fbbavw6v)

[Типичные ошибки](#_tyjcwt)

[Разработка игры Blackjack](#_3dy6vkm)

[Практическое задание](#_1t3h5sf)

[Дополнительные материалы](#_y937sk8fclye)

[Используемая литература](#_uvp6qax5r1ok)

# Термины

Прежде чем приступить к изучению данной главы, вспомним ряд определений, связанных с программированием.

**Исходный код** — программа, написанная на языке программирования, но в текстовом формате. А также текстовый файл, содержащий исходный код.

**Компилятор** — программа, выполняющая компиляцию. На данный момент среди начинающих наиболее популярными компиляторами C/C++ являются GNU g++ (и его порты под различные ОС) и версии MS Visual Studio C++.

**Компиляция** — преобразование исходного кода в объектный модуль.

**Объектный модуль** — двоичный файл, который содержит особым образом подготовленный исполняемый код, который может быть объединен с другими объектными файлами при помощи редактора связей (компоновщика), чтобы получить готовый исполняемый модуль или библиотеку.

**Компоновщик** (**редактор связей**, **линкер**, **сборщик**) — это программа, которая производит компоновку (линковку, сборку): принимает на вход один или несколько объектных модулей и собирает по ним исполняемый модуль.

**Исполняемый модуль** (**исполняемый файл**) — файл, который может быть запущен на исполнение процессором под управлением операционной системы.

**Препроцессор** — программа для обработки текста. Может существовать отдельно или интегрироваться в компилятор. В любом случае входные и выходные данные для препроцессора имеют **текстовый формат**. Препроцессор преобразует текст в соответствии с *директивами препроцессора*. Если текст не содержит так директив, то остается без изменений.

**IDE** (англ. Integrated Development Environment) — интегрированная среда разработки. Программа (или их комплекс), упрощающая написание исходного кода, отладку, управление проектом, установку параметров компилятора, линкера, отладчика. Важно не путать IDE и компилятор. Как правило, компилятор самодостаточен и может не входить в состав IDE. С другой стороны, с некоторыми IDE можно использовать различные компиляторы.

**Объявление** — описание сущности: сигнатура функции, определение типа, описание внешней переменной, шаблон. Объявление уведомляет компилятор о ее существовании и свойствах.

**Определение** — реализация сущности: переменная, функция, метод класса и подобное. При обработке определения компилятор генерирует информацию для объектного модуля: исполняемый код, резервирование памяти под переменную и так далее.

# От исходного кода к исполняемому модулю

Классическая схема создания исполняемого файла подразумевает три этапа:

1. Обработка исходного кода препроцессором.
2. Компиляция в объектный код.
3. Компоновка объектных модулей, включая модули из объектных библиотек, в исполняемый файл.

Это классическая схема для компилируемых языков, хотя сейчас уже используются и другие подходы.

**Компиляция** — это сборка программы, включающая трансляцию всех ее модулей, написанных на языке программирования, в эквивалентные программные модули на машинном языке и последующую сборку исполняемой машиной программы.

IDE обычно скрывают три отдельных этапа создания исполняемого модуля. Они проявляются только в тех случаях, когда на этапе препроцессинга или компоновки обнаруживаются ошибки.

Допустим, у нас есть программа на C++ *«Hello, World!»*:

| #include <iostream>  using namespace std;  int main() {  cout << "Hello, World!\n";  } |
| --- |

Сначала исходный код обрабатывается препроцессором. Он находит директиву**#include <iostream>**, ищет файл *iostream* и заменяет директиву текстом из этого файла, попутно обрабатывая все директивы препроцессора во включаемом тексте.

Файл, указанный в директиве **#include**, в данном случае является *заголовочным файлом*. Это обычный текстовый файл, содержащий **объявления** (объявления типов, прототипы функций, шаблоны, директивы препроцессора и подобное). После **текстуального** включения заголовочного файла в текст программы (или модуля) можно использовать в тексте программы все то, что описано в этом заголовочном файле.

Затем результат работы препроцессора передается компилятору. Он производит весь положенный комплекс работ: от синтаксического разбора и поиска ошибок до создания объектного файла (понятно, что если есть синтаксические ошибки, то объектный файл не создается). В объектном файле обычно есть *таблица внешних ссылок* — в которой, в частности, перечислены имена подпрограмм, которые используются в объектном модуле, но код которых отсутствует в данном объектном модуле. Эти подпрограммы *внешние* по отношению к модулю.

Исходный код, который может быть откомпилирован, называется **единицей компиляции**. Наша программа содержит одну единицу компиляции.

Чтобы получить нормальный исполняемый модуль, необходимо «разрешить» внешние ссылки. То есть добавить в исполняемый модуль код отсутствующих подпрограмм и настроить все ссылки на этот код. Этим занимается компоновщик. Он анализирует таблицу внешних ссылок объектного модуля, ищет в объектных библиотеках недостающие модули, копирует их в исполняемый модуль и настраивает ссылки. После этого исполняемый модуль готов.

**Библиотека** (объектная библиотека) — это набор откомпилированных подпрограмм, собранных в единый структурированный файл. Подключение библиотеки происходит на этапе компоновки исполняемого файла из объектных файлов (то есть из тех файлов, которые получаются в результате компиляции исходного текста программы).

Необходимые объектные библиотеки входят в комплект поставки компилятора. В комплект поставки библиотек (любых) входит набор заголовочных файлов, которые содержат объявления, необходимые компилятору.

Если исходный код программы разделен на несколько файлов, то процесс компиляции и сборки происходит аналогично. Сначала все единицы компиляции по отдельности компилируются, а затем компоновщик собирает полученные объектные модули (с подключением библиотек) в исполняемый файл. Такой процесс называется **раздельной компиляцией**.

# Разделение текста программы на модули

Помимо того, что с большим текстом просто неудобно работать, для разделения исходного текста программы на несколько файлов есть такие основания:

1. Разделение на отдельные модули по функциональному признаку.
2. Разделение согласно паттерну DRY (Don’t repeat yourself), который подразумевает повторное использование модулей в программах.
3. Реализация паттерна MVC, предусматривающего разделение интерфейсной и функциональной частей приложения.

Как только мы решаем разделить исходный текст программы на несколько файлов, возникают две проблемы:

1. Необходимо от простой компиляции программы перейти к раздельной. Для этого надо внести соответствующие изменения:
   1. либо в последовательность действий при построении приложения вручную;
   2. либо в командные или make-файлы, автоматизирующие процесс построения;
   3. либо в проект IDE.
2. Надо решить, каким образом разбить текст программы на отдельные файлы.

Первая проблема — чисто техническая. Она решается чтением руководств по компилятору и/или линкеру, утилите *make* или IDE. В самом худшем случае просто придется проштудировать все эти руководства.

Вторая проблема требует более творческого подхода. Хотя и здесь существуют рекомендации, несоблюдение которых приводит либо к невозможности собрать проект, либо к трудностям в его дальнейшем развитии.

Нужно определить, какие части программы выделить в отдельные модули. Чтобы это получилось просто и естественно, программа должна быть правильно спроектирована. О том, как правильно спроектировать программу, написано много книг по методологии программирования. Краткая рекомендация: вся программа должна состоять из слабо связанных фрагментов. Приведем пример из реальной жизни. Периферия компьютера — мышка, клавиатура, монитор — это в некотором роде «черные ящики». Каждый элемент выполняет свою функцию и может быть заменен любым подобным ему объектом, главное — подходящим.

В программировании каждый такой «независимый» фрагмент может быть естественным образом преобразован в отдельный модуль (единицу компиляции). Обратите внимание, что под «фрагментом» подразумевается не просто произвольный кусок кода, а функция, или группа логически связанных функций, или класс, или несколько тесно взаимодействующих классов.

Еще нужно определить интерфейсы для модулей — для этого есть четкие правила.

# Интерфейс и реализация

Когда фрагмент программы выделяется в модуль (единицу компиляции), остальной ее части (а точнее компилятору, который будет ее обрабатывать) надо объяснить, что имеется в этом модуле. Для этого служат *заголовочные файлы*.

Таким образом, модуль состоит из двух файлов: заголовочного (интерфейса) и файла реализации.

Заголовочный файл, как правило, имеет расширение ***.h*** или ***.hpp***, а файл реализации — ***.cpp*** для программ на C++ и ***.c*** — на языке C. Хотя в STL включаемые файлы вообще без расширений, но по сути они являются заголовочными файлами.

Заголовочный файл должен содержать все объявления, которые должны быть видны снаружи. Другие объявления делаются в файле реализации.

*Что может быть в заголовочном файле?*

**Заголовочный файл может содержать только объявления и не должен содержать определения.**

Разделение программы позволяет скрыть реализацию, предоставив “клиентам” лишь интерфейс.

То есть, при обработке содержимого заголовочного файла компилятор не должен генерировать информацию для объектного модуля.

Исключение из этого правила — написание шаблона функции или класса внутри заголовочного файла.

**Заголовочный файл должен иметь механизм защиты от повторного включения.**

Защита от повторного включения реализуется директивами препроцессора:

| #ifndef SYMBOL *// если SYMBOL не объявлен*  #define SYMBOL *// объявить SYMBOL*  *// набор объявлений*  #endif *// конец #ifndef* |
| --- |

Для препроцессора при первом включении заголовочного файла это выглядит так: поскольку условие «символ SYMBOL не определен» истинно, определить символ **SYMBOL** и обработать все строки до директивы **#endif**. При повторном включении — так: поскольку условие «символ SYMBOL не определен» (**#ifndef SYMBOL**) ложно (символ был определен при первом включении), то пропустить все до директивы **#endif**.

В качестве SYMBOL обычно применяют имя самого заголовочного файла в верхнем регистре, обрамленное одинарными или сдвоенными подчеркиваниями. Например, для файла ***header.h*** традиционно используется **#define \_\_HEADER\_H\_\_**. Впрочем, символ может быть любым, но обязательно уникальным в рамках проекта.

В качестве альтернативного способа может применяться директива **#pragma once**. Однако преимущество первого способа в том, что он работает на любых компиляторах.

Заголовочный файл сам по себе не является единицей компиляции.

*Что может быть в файле реализации?*

Файл реализации может содержать как определения, так и объявления. Объявления, сделанные в файле реализации, будут лексически локальны для этого файла — то есть будут действовать только для этой единицы компиляции.

**В файле реализации должна быть директива включения соответствующего заголовочного файла.**

Понятно, что объявления, которые видны снаружи модуля, должны быть доступны и внутри. Данное правило также гарантирует соответствие между описанием и реализацией. При несовпадении, допустим, сигнатуры функции в объявлении и определении компилятор выдаст ошибку.

**В файле реализации не должно быть объявлений, дублирующих объявления в соответствующем заголовочном файле.**

# Практический пример

Рассмотрим программу:

| #include <iostream>  using namespace std;  const int dayDefault = 10; *// глобальная константа*  int celebration = 0; *// глобальная переменная*  int holiday = 0; *// глобальная переменная для increase и decrease*  int increase() {  ++celebration;  return ++holiday;  }  int decrease() {  --celebration;  return --holiday;  }  class Date {  public:  Date() : day(dayDefault) { ++number; }  ~Date() { --number; }  void changeDay(int arg);  int get\_day() const;  int get\_number() const;  private:  int day;  static int number;  };  int Date::number = 0;  void Date::changeDay(int arg) {  day = arg;  }  int Date::get\_day() const {  return day;  }  int Date::get\_number() const {  return number;  }  int main()  {  int days;  days = increase();  days = decrease();  cout << "Days: " << days << " number: " << celebration << endl;  Date d1, d2;  if (d1.get\_day() == dayDefault)  cout << "Ok" << endl;  cout << d2.get\_number() << endl;  return 0;  } |
| --- |

Мы имеем:

* глобальную константу **dayDefault**, которая используется и в классе, и в **main**;
* глобальную переменную **celebration**, которая используется в функциях **increase()**, **decrease()** и **main**;
* глобальную переменную **holiday**, которая используется только в функциях **increase()** и **decrease()**;
* функцию **increase()** и **decrease()**;
* класс **Date**;
* функцию **main**.

Теперь пробуем разделить программу на модули.

Сначала как наиболее связанные сущности (используются во многих местах программы) выносим в отдельную единицу компиляции глобальную константу **dayDefault** и глобальную переменную **celebration**.

Файл **globals.h**:

| #ifndef \_\_GLOBALS\_H\_\_  #define \_\_GLOBALS\_H\_\_  const int dayDefault = 10; *// глобальная константа*  extern int celebration; *// глобальная переменная*  #endif *// \_\_GLOBALS\_H\_\_* |
| --- |

**Файл globals.cpp**

| #include "globals.h"  int celebration = 0; *// глобальная переменная* |
| --- |

Обратите внимание, что глобальная переменная в заголовочном файле имеет спецификатор **extern**. В файле **globals.h** мы только *объявляем* переменную, а *определяем* ее в другом файле. Такое описание означает, что где-то существует переменная с таким именем и указанным типом. А определение этой переменной (с инициализацией) помещено в файл реализации. Константа описана в заголовочном файле.

Если константа тривиального типа, то ее можно объявить в заголовочном файле. В противном случае она должна быть определена в файле реализации, а в заголовочном должно быть ее объявление (как для переменной).

Также обратите внимание на защиту от повторного включения заголовочного файла и на включение заголовочного файла в файле реализации.

Затем выносим в отдельный модуль функции **increase()** и **decrease()** с глобальной переменной **holiday**. Получаем еще два файла:

**Файл funcs.h**

| #ifndef \_\_FUNCS\_H\_\_  #define \_\_FUNCS\_H\_\_  int increase();  int decrease();  #endif *// \_\_FUNCS\_H\_\_* |
| --- |

**Файл funcs.cpp**

| #include "funcs.h"  #include "globals.h"  int holiday = 0; *// глобальная переменная для increase и decrease*  int increase() {  ++celebration;  return ++holiday;  }  int decrease() {  ++celebration;  return --holiday;  } |
| --- |

Поскольку переменная **holiday** используется только этими двумя функциями, ее объявление в заголовочном файле отсутствует. Из этого модуля «на экспорт» идут только две функции.

В функциях используется переменная из другого модуля, поэтому необходимо добавить **#include "globals.h"**.

Наконец, выносим в отдельный модуль класс **Date**:

**Файл Date.h**

| #ifndef \_\_Date\_H\_\_  #define \_\_Date\_H\_\_  class Date {  public:  Date();  ~Date();  void changeDay(int arg);  int get\_day() const;  int get\_number() const;  private:  int day;  static int number;  };  #endif *// \_\_Date\_H\_\_* |
| --- |

**Файл Date.cpp**

| #include "Date.h"  #include "globals.h"  int Date::number = 0;  Date::Date() : day(dayDefault) {  ++number;  }  Date::~Date() {  --number;  }  void Date::changeDay(int arg) {  day = arg;  }  int Date::get\_day() const {  return day;  }  int Date::get\_number() const {  return number;  } |
| --- |

Обратите внимание на следующие моменты.

1. Из объявления класса убрали определения методов, так как интерфейс и реализация должны быть разделены.
2. Класс имеет статический член класса, то есть для всех экземпляров класса эта переменная будет общей. Ее инициализация выполняется не в конструкторе, а в глобальной области модуля.
3. В файл реализации добавлена директивам **#include "globals.h"** для доступа к константе **cint**.

Классы практически всегда выделяются в отдельные единицы компиляции.

В файле ***main.cpp*** оставляем только функцию **main**. И добавляем необходимые директивы включения заголовочных файлов.

**Файл main.cpp**

| #include <iostream>  #include "funcs.h"  #include "Date.h"  #include "globals.h"  using namespace std;  int main()  {  int days;  days = increase();  days = decrease();  cout << "days: " << days << " number: " << celebration << endl;  Date d1, d2;  if (d1.get\_day() == dayDefault)  cout << "Ok" << endl;  cout << d2.get\_number() << endl;  return 0;  } |
| --- |

# Типичные ошибки

**Ошибка 1.** **Определение в заголовочном файле.**

Эта ошибка в ряде случаев может себя не проявлять. Например, когда заголовочный файл с этой ошибкой включается только один раз. Но как только этот заголовочный файл будет включен более одного раза, получим либо ошибку компиляции *«многократное определение символа ...»*, либо ошибку компоновщика (аналогичного содержания), если второе включение было сделано в другой единице компиляции.

**Ошибка 2.** **Отсутствие защиты от повторного включения заголовочного файла.**

Тоже проявляет себя при определенных обстоятельствах. Может вызывать ошибку компиляции *«многократное определение символа ...»*.

**Ошибка 3.** **Несовпадение объявления в заголовочном файле и определения в файле реализации.**

Обычно возникает в процессе редактирования исходного кода, когда в файл реализации вносятся изменения, а про заголовочный файл забывают.

**Ошибка 4.** **Отсутствие необходимой директивы #include.**

Если необходимый заголовочный файл не включен, то все сущности, которые в нем объявлены, останутся неизвестными компилятору. Вызывает ошибку компиляции *«не определен символ ...»*.

**Ошибка 5.** **Отсутствие необходимого модуля в проекте построения программы.**

Вызывает ошибку компоновки *«не определен символ ...»*. Обратите внимание, что имя символа в сообщении компоновщика почти всегда отличается от того, которое определено в программе: оно дополнено другими буквами, цифрами или знаками.

**Ошибка 6.** **Зависимость от порядка включения заголовочных файлов.**

Не совсем ошибка, но таких ситуаций следует избегать. Обычно сигнализирует об ошибках либо в проектировании программы, либо при разделении исходного кода на модули.

# Разработка игры Blackjack

 Класс **Deck** представляет колоду карт и наследует от класса **Hand**.

| class Deck : public Hand  {  public:  Deck();    virtual ~Deck();    *// создает стандартную колоду из 52 карт*  void Populate();    *// тасует карты*  void Shuffle();    *// раздает одну карту в руку*  void Deal(Hand& aHand);    *// дает дополнительные карты игроку*  void AdditionalCards(GenericPlayer& aGenericPlayer);  };  Deck::Deck()  {  m\_Cards.reserve(52);  Populate();  }  Deck::~Deck()  {} |
| --- |

Функция **Populate()** создает стандартную колоду из 52 карт. Функция-член проходит по всем возможным комбинациям значений перечислений **Card::suit** и **Card::rank**. Она использует **static\_cast**, чтобы преобразовать целочисленные переменные в значения перечислений, определенных в классе **Card**.

| void Deck::Populate()  {  Clear();  *// создает стандартную колоду*  for (int s = Card::CLUBS; s <= Card::SPADES; ++s)  {  for (int r = Card::ACE; r <= Card::KING; ++r)  {  Add(new Card(static\_cast<Card::rank>(r),  static\_cast<Card::suit>(s)));  }  }  } |
| --- |

Функция **Shuffle()** тасует колоду карт. Она в случайном порядке переставляет указатели, расположенные в векторе **m\_Cards** с помощью функции **random\_shuffle()** стандартной библиотеки шаблонов. Именно поэтому нужно подключить заголовочный файл **<algorithm>**.

| void Deck::Shuffle()  {  random\_shuffle(m\_Cards.begin(), m\_Cards.end());  } |
| --- |

Функция **Deal()** выдает одну карту из колоды в руку. Она добавляет копию указателя в конец вектора **m\_Cards** с помощью функции-члена **Add()**. Далее она удаляет указатель из конца вектора **m\_Cards**, что, по сути, является перемещением карты. Функция **Deal()** мощная, поскольку она принимает ссылку на объект типа **Hand** — и это означает, что она может работать также с объектами классов **Player** и **House**. Благодаря полиморфизму функция **Deal()** может вызывать функцию-член **Add()** любого объекта из этих классов, не зная его конкретный тип.

| void Deck::Deal(Hand& aHand)  {  if (!m\_Cards.empty())  {  aHand.Add(m\_Cards.back());  m\_Cards.pop\_back();  }  else  {  cout << "Out of cards. Unable to deal.";  }  } |
| --- |

Функция **AdditionalCards()** дает дополнительные карты игроку до тех пор, пока он этого хочет или у него не образуется перебор. Функция-член принимает ссылку на объект типа **GenericPlayer**, поэтому вы можете передать ей объект типа **Player** или **House**. Благодаря полиморфизму функция **AdditionalCards()** может не знать, с объектом какого типа она работает. Она может вызывать функции-члены **IsBusted()** и **IsHitting()**, не зная типа объекта, и при этом будет выполнен корректный код.

| void Deck::AdditionalCards(GenericPlayer& aGenericPlayer)  {  cout << endl;  *// продолжает раздавать карты до тех пор, пока у игрока не случается*  *// перебор или пока он хочет взять еще одну карту*  while (!(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting())  {  Deal(aGenericPlayer);  cout << aGenericPlayer << endl;    if (aGenericPlayer.IsBusted())  {  aGenericPlayer.Bust();  }  }  } |
| --- |

Класс **Game** представляет игру Blackjack.

| class Game  {  public:  Game(const vector<string>& names);    ~Game();    *// проводит игру в Blackjack*  void Play();    private:  Deck m\_Deck;  House m\_House;  vector<Player> m\_Players;  };  *// Конструктор этого класса принимает ссылку на вектор строк, представляющих*  *// имена игроков-людей. Конструктор создает объект класса Player для каждого имени*  Game::Game(const vector<string>& names)  {  *// создает вектор игроков из вектора с именами*  vector<string>::const\_iterator pName;  for (pName = names.begin(); pName != names.end(); ++pName)  {  m\_Players.push\_back(Player(\*pName));  }    *// запускает генератор случайных чисел*  srand(static\_cast<unsigned int>(time(0)));  m\_Deck.Populate();  m\_Deck.Shuffle();  }  Game::~Game()  {}  void Game::Play()  {  *// раздает каждому по две стартовые карты*  vector<Player>::iterator pPlayer;  for (int i = 0; i < 2; ++i)  {  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)  {  m\_Deck.Deal(\*pPlayer);  }  m\_Deck.Deal(m\_House);  }    *// прячет первую карту дилера*  m\_House.FlipFirstCard();    *// открывает руки всех игроков*  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)  {  cout << \*pPlayer << endl;  }  cout << m\_House << endl;    *// раздает игрокам дополнительные карты*  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)  {  m\_Deck.AdditionalCards(\*pPlayer);  }    *// показывает первую карту дилера*  m\_House.FlipFirstCard();  cout << endl << m\_House;    *// раздает дилеру дополнительные карты*  m\_Deck.AdditionalCards(m\_House);    if (m\_House.IsBusted())  {  *// все, кто остался в игре, побеждают*  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)  {  if (!(pPlayer->IsBusted()))  {  pPlayer->Win();  }  }  }  else  {  *// сравнивает суммы очков всех оставшихся игроков с суммой очков дилера*  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end();  ++pPlayer)  {  if (!(pPlayer->IsBusted()))  {  if (pPlayer->GetTotal() > m\_House.GetTotal())  {  pPlayer->Win();  }  else if (pPlayer->GetTotal() < m\_House.GetTotal())  {  pPlayer->Lose();  }  else  {  pPlayer->Push();  }  }  }    }    *// очищает руки всех игроков*  for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); ++pPlayer)  {  pPlayer->Clear();  }  m\_House.Clear();  } |
| --- |

Конструктор этого класса принимает ссылку на вектор строк, представляющих имена игроков-людей. Конструктор создает объект класса **Player** для каждого имени.

Код игры **Blackjack** приведен в отдельном файле (**Blackjack.cpp**). В тексте кода даны краткие пояснения по используемым функциям и классам.

# Практическое задание

1. Создайте класс **Date** с полями день, месяц, год и методами доступа к этим полям. Перегрузите оператор вывода для данного класса. Создайте два "умных" указателя **today** и **date**. Первому присвойте значение сегодняшней даты. Для него вызовите по отдельности методы доступа к полям класса **Date**, а также выведите на экран данные всего объекта с помощью перегруженного оператора вывода. Затем переместите ресурс, которым владеет указатель **today** в указатель **date**. Проверьте, являются ли нулевыми указатели **today** и **date** и выведите соответствующую информацию об этом в консоль.

2. По условию предыдущей задачи создайте два умных указателя **date1** и **date2**.

* Создайте функцию, которая принимает в качестве параметра два умных указателя типа **Date** и сравнивает их между собой (сравнение происходит по датам). Функция должна вернуть более позднюю дату.
* Создайте функцию, которая обменивает ресурсами (датами) два умных указателя, переданных в функцию в качестве параметров.

*Примечание: обратите внимание, что первая функция не должна уничтожать объекты, переданные ей в качестве параметров.*

3. Создать класс **Deck**, который наследует от класса **Hand** и представляет собой колоду карт. Класс **Deck** имеет 4 метода:

* **vold Populate()** - Создает стандартную колоду из 52 карт, вызывается из конструктора.
* **void Shuffle()** - Метод, который тасует карты, можно использовать функцию из алгоритмов **STL random\_shuffle**
* **vold Deal (Hand& aHand)** - метод, который раздает в руку одну карту
* **void AddltionalCards (GenericPlayer& aGenerlcPlayer)** - раздает игроку дополнительные карты до тех пор, пока он может и хочет их получать

*Обратите внимание на применение полиморфизма. В каких методах применяется этот принцип ООП?*

4. Реализовать класс **Game**, который представляет собой основной процесс игры. У этого класса будет 3 поля:

* колода карт
* рука дилера
* вектор игроков.

Конструктор класса принимает в качестве параметра вектор имен игроков и создает объекты самих игроков. В конструкторе создается колода карт и затем перемешивается.

Также класс имеет один метод **play()**. В этом методе раздаются каждому игроку по две стартовые карты, а первая карта дилера прячется. Далее выводится на экран информация о картах каждого игра, в т.ч. и для дилера. Затем раздаются игрокам дополнительные карты. Потом показывается первая карта дилера и дилер набирает карты, если ему надо. После этого выводится сообщение, кто победил, а кто проиграл. В конце руки всех игроков очищаются.

5. Написать функцию **main()** к игре **Блекджек**. В этой функции вводятся имена игроков. Создается объект класса **Game** и запускается игровой процесс. Предусмотреть возможность повторной игры.

# Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования С++.
2. Стивен Прата. Язык программирования С++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в С++.

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Модульное программирование](http://staff.mmcs.sfedu.ru/~dubrov/files/sl_oop_02_module.pdf).
2. Бьерн Страуструп. Программирование. Принципы и практика использования С++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.