МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Ижевский государственный технический университет имени М.Т. Калашникова» (ФГБОУ ВО «ИжГТУ имени М.Т. Калашникова»)

Марков Е. М.

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

к выполнению лабораторных работ по дисциплине
«Объектно-ориентированное программирование»
Второй семестр

Ижевск 2020

Рег. Номер

Методические рекомендации по организации самостоятельной работы студентов составлены в соответствии с рабочей программой учебной дисциплины, разработанной на основе Федерального государственного образовательного стандарта по направлению 09.03.01 «Информатика и вычислительная техника» профиль «Вычислительные машины, комплексы, системы и сети» при изучении дисциплины «Объектно-ориентированное программирование».

Рецензент: Марков М.М., к.т.н., доцент кафедры «Радиотехника»

Составители: Марков Е.М., к.т.н., доцент, доцент каф. ВТ

Рекомендовано Ученым советом факультета для использования в учебном процессе в качестве учебно-методических материалов для студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» профиль «Вычислительные машины, комплексы, системы и сети» при изучении дисциплины «Объектно-ориентированное программирование»

Оглавление

ВВЕДЕНИЕ

Ни для кого не секрет, что значительная часть современных радиотехнических устройств построена на базе микропроцессорной техники. Самыми яркими примерами являются носимая техника, автомобильные сигнализации, устройства управления различными процессами и т.д. Разрабатываются многочисленные устройства, работающие совместно с прочно вошедшими в повседневную жизнь персональными компьютерами. Поэтому современному системотехнику жизненно необходимо владеть навыками программирования.

Язык программирования C++ – компилируемый строго типизированный язык программирования общего назначения. Поддерживает разные парадигмы программирования: процедурную, обобщённую, функциональную; наибольшее внимание уделено поддержке объектно-ориентированного программирования. В 1990-х годах язык стал одним из наиболее широко применяемых языков программирования общего назначения.

При создании C++ стремились сохранить совместимость с языком C. Большинство программ на C будут исправно работать и с компилятором C++. Язык C++ имеет синтаксис, основанный на синтаксисе C. Язык возник в начале 1980-х годов, когда сотрудник фирмы «Bell Laboratories» Бьёрн Страуструп придумал ряд усовершенствований к языку C под собственные нужды. До начала официальной стандартизации язык развивался в основном силами Страуструпа в ответ на запросы программистского сообщества.

В 1998 году был ратифицирован международный стандарт языка C++: ISO/IEC 14882:1998 «Standard for the C+ + Programming Language»; после принятия технических исправлений к стандарту в 2003 году нынешняя версия этого стандарта – ISO/IEC 14882:2003. Название «C++» происходит от C, в котором оператор «++» обозначает приращение.

Овладение практическими навыками является важнейшим принципом в программировании. Важнейшую роль при этом играет лабораторный практикум. Считается, что научиться программировать можно только в результате **практической деятельности**.

Лабораторная работа № 1 Библиотека STL, последовательные контейнеры и алгоритмы

Основное содержание работы.

Цель. Получить практические навыки работы с возможностями стандартной библиотеки шаблонов (STL), на примере последовательных контейнеров и алгоритмов.

Написать программу, в которой данные в последовательном контейнере будут обрабатываться алгоритмами из STL.

Краткие теоретические сведения.

Основные концепции STL

STL – Standard Template Library, стандартная библиотека шаблонов (**шаблоны** - средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам, подробнее https://ru.wikipedia.org/wiki/Шаблоны_C%2B%2B) состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. Контейнеры STL реализуют основные структуры данных, используемые при написании программ.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам: поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов first, last, задающая диапазон обрабатываемых элементов.

Итераторы – это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать (++), как обычные указатели, для последовательного продвижения по контейнеру, а также разыменовывать для получения или изменения значения элемента (*).

Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные (рис. 1).

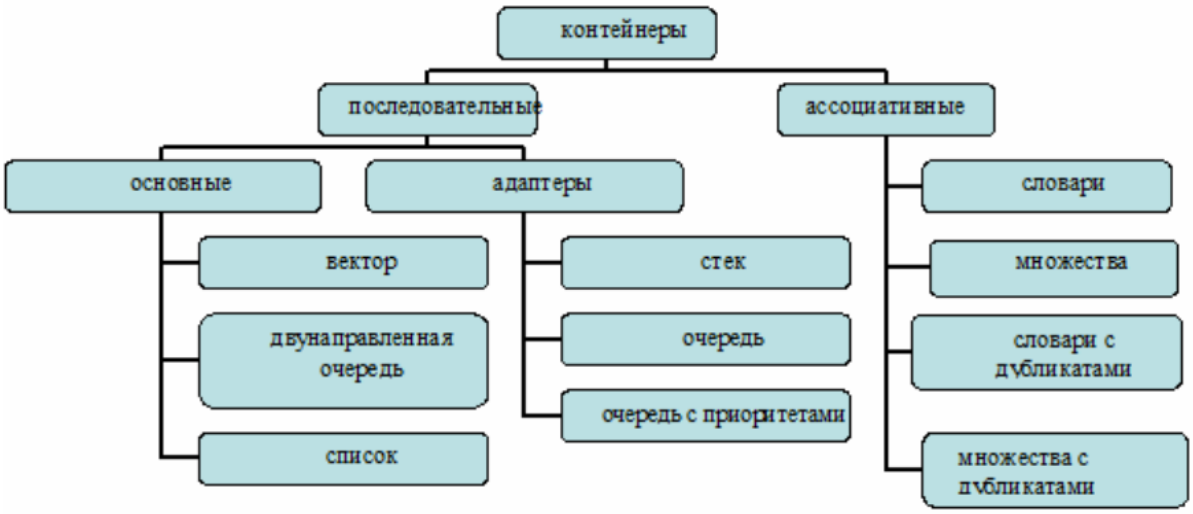

Рис. 1. Контейнерные классы

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся:

- векторы (`vector`)
- списки (`list`)
- двусторонние очереди (`deque`).

Есть еще специализированные контейнеры (или адаптеры контейнеров), реализованные на основе базовых

- стеки (`stack`)
- очереди (`queue`)
- очереди с приоритетами (`priority_queue`).

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
#include <vector>
#include <list>
#include “person.h”
…..
vector<int> v;
list<person> l;
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров:

- словари (`map`)
- словари с дубликатами (`multimap`)
- множества (`set`)
- множества с дубликатами (`multiset`)
- битовые множества (`bitset`).

В последних стандартах C++ появились и другие типы контейнеров, например: неупорядоченные ассоциативные, потокобезопастные и другие.

Итераторы

Рассмотрим, как можно реализовать шаблон функции для поиска элементов в массиве, который хранит объекты типа Data:

```
template <class Data>
Data* Find(Data*mas, int n, const Data& key)
{
    for(int i=0;i<n;i++)
     if (*(mas + i) == key)
      return mas + i;
    return 0;
}
```

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден. Несмотря на приведенный выше пример, использовать указатели с более сложными контейнерами, нежели простые массивы, довольно затруднительно. Во-первых, если элементы контейнера хранятся не последовательно в памяти, а сегментировано, то методы доступа к ним значительно усложняются. Мы не можем в этом случае просто инкрементировать указатель для получения следующего значения. Например, при движении от элемента к элементу в связном списке мы не можем по умолчанию предполагать, что следующий является соседом предыдущего. Приходится идти по цепочке ссылок.

Подход, используемый в STL, заключается как раз в том, чтобы сделать интеллектуальные указатели полностью независимыми и даже дать им собственное имя — **итераторы**. На самом деле, они представляют собой целое семейство шаблонных классов. Для создания итераторов необходимо определять их в качестве объектов таких классов.

Тип iterator определен для всех контейнерных классов STL, однако, реализация его в разных классах разная. К основным операциям, выполняемым с любыми итераторами, относятся:

- Разыменование итератора: если p — итератор, то *p — значение объекта, на который он ссылается.
- Присваивание одного итератора другому.
- Сравнение итераторов на равенство и неравенство (== и !=).
- Перемещение его по всем элементам контейнера с помощью префиксного (++p) или постфиксного (p++) инкремента.

Так как реализация итератора специфична для каждого класса, то при объявлении объектов типа итератор всегда указывается область видимости в форме имя_шаблона : : , например:

```
vector<int>::iterator iterl;
List<person>::iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если i — некоторый итератор, то вместо привычной формы используется следующая:

```
for (i = first; i != last; ++i),
```

где `first` - значение итератора, указывающее на первый элемент в контейнере, а `last` — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Типовая работа с итератором:

```
list<int>::iterator iter; //итератор для целочисленного списка
for(iter = theList.begin(); iter != theList.end(); iter++)
    cout << *iter << ' '; //вывести список

iter = theList.begin();
while( iter != theList.end() )
    cout << *iter++ << ' ';
```

Для удобства работы с итераторами методы и функции, например `diatance()` возвращает количество элементов между итераторами.

```
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    std::vector<int> v{ 3, 1, 4 };
    std::cout << "distance(first, last) = "
              << std::distance(v.begin(), v.end()) << '\n'
              << "distance(last, first) = "
              << std::distance(v.end(), v.begin()) << '\n';
               //the behavior is undefined (until C++11)
}
```

Результат:

```
distance(first, last) = 3
distance(last, first) = -3
```

Использование последовательных контейнеров

К основным последовательным контейнерам относятся вектор (vector), список (list) и двусторонняя очередь (deque).

Чтобы использовать последовательный контейнер, нужно включить в программу соответствующий заголовочный файл:

```
#include <vector>;
#include <list>;
#include <deque>;
using namespace std;
```

Контейнер **вектор** является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Однако **вставка элемента в любую позицию, кроме конца вектора, неэффективна**. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер список организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список **не поддерживает произвольного доступа к своим элементам**: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-1 элементов.

Контейнер двусторонняя очередь во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vecl;
list<double> listl;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<int> vecl(100);
list<double> listl(20);
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<int> vecl(100, 0); deque<float> decl(300, 0.0);
```

4. Создать контейнер и инициализировать его элементы значениями диапазона (first, last) элементов другого контейнера:

```
int arr[7] = (15, 2, 19, -3, 28, 6, 8);
vector<int> vl(arr, arr + 7);
list<int> lst(vl.beg() + 2, vl.end());
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого однотипного контейнера:

```
vector<int> vl;
// добавить в vl элементы
vector<int> v2(vl);
```

Методы стандартных контейнеров (перечень методов зависит от контейнера):

- `begin`() Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
- `end`() Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
- `rbegin`() Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)
- `rend`() Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены

- `push_back()` – добавление в конец
- `pop_back()` – извлечение с удаление из конца
- `insert()` – вставка в произвольное место
- `erase()` – удаление из произвольного места
- `pop_front()` – извлечение с удалением из начала
- `swap()` – обмен контейнеров
- `clear()` – очистить контейнер
- `splice()` – сцепка списков
- `[] at` – доступ к произвольному элементу
- `push()` – добавление в конец очереди;
- `pop()` – удаление из начала очереди;
- `front()` – получение первого элемента очереди;
- `back()`- получение последнего элемента очереди;
- `empty()` – проверка пустая очередь или нет;
- `size()` – получение размера очереди.

Алгоритмы

Алгоритмы STL реализуют некоторые распространенные операции с контейнерами, которые не реализуются методами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов и прочие). Такие операции являются универсальными для любого из контейнеров и поэтому находятся вне этих контейнеров. Зная, как устроены алгоритмы, можно писать необходимые дополнительные алгоритмы обработки, которые не будут зависеть от контейнера.

Каждый алгоритм представлен шаблоном функции или набором шаблонов функций. Все стандартные алгоритмы находятся в пространстве имен `std`, а их объявления – в библиотечных файлах `<algorithm>,.<cmath>, <numbers>` и других.

Можно выделить три основные группы алгоритмов:

1. **Немодифицирующие** алгоритмы, те, которые извлекают информацию из контейнера (о его устройстве, об элементах, которые там есть и т. д.), но никак не модифицируют сам контейнер (ни элементы, ни порядок их расположения). Примеры:
   - `find()` – поиск первого вхождения элемента с заданным значением
   - `count()` – количество вхождений элемента с заданным значением
   - `for_each()` – для применения некоторой операции к каждому элементу, не изменяющей элементы контейнера
   - `min_element()` – для поиска минимального элемента, возвращает итератор на этот элемент
   - `max_element()` – для поиска максимального элемента, возвращает итератор на этот элемент

2. **Модифицирующие** алгоритмы, которые каким-либо образом изменяют содержимое контейнера. Либо сами элементы меняются, либо их порядок, либо их количество. Примеры:

- `generate_n`() – для генерации n элементов, правило генерации можно задать функтором
- `transform`() – для применения некоторой операции к каждому элементу, изменяющей элементы контейнера в отличие от алгоритма `for_each`
- `reverse`() – переставляет элементы в последовательности
- `copy`() – создает новый контейнер

3. **Сортировка**. Примеры:

- `sort`() – простая сортировка
- `stable_sort`() – сохраняет порядок следования одинаковых элементов
- `merge`() – объединяет две отсортированные последовательности

Для работы алгоритмов в необходимы операции ==, <, =, [], и подобные, для стандартных типов данных они определены. Если необходимо использовать в контейнере нестандартные типы, то необходимо перегрузить соответствующие операции (например для работы `min_element` или `sort` необходима перегруженная операция «<») или указать **функциональный объект** (функтор, лямбда выражение).

Функциональный объект

Одним из популярнейших применений функциональных объектов является передача их в качестве аргументов алгоритмам. Тем самым можно регулировать поведение последних.

Давайте вспомним, что такое функциональный объект. Это просто-напросто функция, которая может быть оформлена отдельно или в классе. При оформлении в виде класса может не быть компонентных данных, а есть только один метод или несколько методов: перегружаемая операция (), или другая операция для алгоритма. Класс часто делают шаблонным, чтобы можно было работать с разными типами данных. Типовые шаблонные функциональные объекты:

Функциональный объект	Возвращаемое значение
T = plus(T, T)	x+y
T = minus(T, T)	x -y
T = times(T, T)	x*y
T = divide(T, T)	x/y
T = modulus(T, T)	x%y
T = negate(T)	-x
bool = equal_to(T, T)	x == y
bool = not_equal_to(T, T)	x != y
bool = greater(T, T)	x > y

bool = less(T, T)	x < y
bool = greater_equal(T, T)	x >= y
bool = less_equal(T, T)	x <= y
bool = logical_and(T, T)	x && y
bool = logical_or(T, T)	x || y
bool = logical_not(T, T)	!x

Пример функционального объекта:

```
class MyFo
{
 public: void operator ()(int _x) const { cout << _x << " "; }
};
```

Пример лябмда выражения

```
int result = count_if(srcVec.begin(), srcVec.end(), [] (int _n)
     {
          return (_n % 2) == 0;
     });
```

Подробнее о лямбда выражениях и функциональных объектах можно почитать тут https://habr.com/ru/post/66021/

Исключения (exception)

Бывают ситуации, когда обработать ошибки стандартным способом через условия сложно или невозможно, (обработка ошибок в чужом коде, в конструкторе, …) в этом случае используют исключения.

Используя исключения, мы сможем избежать таких проблем. Чтобы «прикрутить» исключение к этому примеру, надо познакомиться со следующими командами C++: **throw** (в переводе — обработать, запустить), **try** (попытка), **catch** (поймать, ловить). В примере ниже, исключение сработает так: программа получает конкретное указание от программиста — если значение определённой переменной в определённом участке кода (в try-блоке) будет равно 0, то в этом случае пусть генерируется исключение throw. Это исключение автоматически передастся catch-блоку в виде параметра и выполнится код этого блока.

```
try //код, который может привести к ошибке, располагается тут
{
   if (num2 == 0)
   {
      throw 123; //генерировать целое число 123
   }
    cout << num1 / num2 << endl;
}
catch(int i)//сюда передастся число 123
{
  cout << "Ошибка №" << i
       << " - на 0 делить нельзя!!!!" << endl;
}
```

Блоков **catch** может быть несколько на различные аргументы, еще есть специальный блок **finaly** (окончательно) который выполнится после одного из catch, если сработало исключение.

Методические указания.

1. Для определения контейнера можно использовать следующую запись:
```
vector<bitset<32> > bitVector;
```
2. Для случайной генерации использовать генератор из библиотеки случайных чисел <random>.
https://docs.microsoft.com/ru-ru/cpp/standard-library/random
3. Для обработки элементов использовать алгоритмы STL, find(), sort(), max_element() и другие. Пример использования
```
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>
// функциональный объект для модуля
static bool abs_compare(int a, int b)
{
    return (std::abs(a) < std::abs(b));
}

int main()
{
    std::vector<int> v{ 3, 1, -14, 1, 5, 9 };
    std::vector<int>::iterator result;

    result = std::max_element(v.begin(), v.end());
    std::cout << "индекс максимального элемента: " <<
std::distance(v.begin(), result) << '\n';

    result = std::max_element(v.begin(), v.end(),
abs_compare);
    std::cout << "индекс максимального (по модулю)
элемента: " << std::distance(v.begin(), result);
}
```
4. Для работы большинства алгоритмов STL с контейнером bitset понадобиться функциональный объект для операции меньше, вариант его реализации с учетом знака значения:
```
bool BitComp(bitset<32>& l, bitset<32>& r)
{
  return int(l.to_ulong()) < int(r.to_ulong());
}
```

Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.

2. Постановка задачи.
3. Описание понятия шаблона и исключений.
4. Описание использованного контейнера и алгоритмов STL.
5. Скриншоты приложения
6. Листинг демонстрационной программы.

Задание и варианты.

1. Разработать программу по заданию, сама обработка выполняется через алгоритмы STL, обязательно использовать обработку исключений.
2. Использовать подходящий для обработки последовательный контейнер с элементами `bitset` 32 бита (заполняется случайно, все 32 разряда, использовать генератор из класса `<random>`), допускается использовать произвольные типы или стандартные по согласованию с преподавателем.
3. В зависимости от задания массив выводиться до и после обработки в двоичном, десятичном и шестнадцатеричном виде.

Варианты

1. В контейнере поменять местами максимальный и минимальный элементы.
2. В контейнере отсортировать числа между максимальным и минимальным элементом.
3. В контейнере отсортировать числа до максимального значения.
4. В контейнере отсортировать числа после максимального значения.
5. В контейнере отсортировать числа до минимального значения.
6. В контейнере отсортировать числа после минимального значения.
7. В контейнере заменить минимальный элемент на первый элемент в контейнере.
8. В контейнере заменить минимальный элемент на последний элемент в контейнере.
9. В контейнере заменить максимальный элемент на первый элемент в контейнере.
10. В контейнере заменить максимальный элемент на первый элемент в контейнере.
11. Найти максимальный элемент и добавить его в начало контейнера.
12. Найти минимальный элемент и удалить его из контейнера.
13. Найти минимальный элемент и добавить его в начало контейнера.
14. Найти максимальный элемент и удалить его из контейнера.
15. Найти максимальный элемент и добавить его в конец контейнера.
16. Найти минимальный элемент и добавить его в конец контейнера.
17. Почитать количество элементов в контейнере больше среднего максимума и минимума.

18. Почитать количество элементов в контейнере меньше среднего максимума и минимума.
19. В контейнере удалить элементы до максимума.
20. В контейнере удалить элементы после максимума.
21. В контейнере удалить элементы до минимума.
22. В контейнере удалить элементы после минимума.
23. В контейнере заменить все элементы до максимума на минимум.
24. В контейнере заменить все элементы после максимума на минимум.
25. В контейнере заменить все элементы до минимума на максимум.
26. В контейнере заменить все элементы после минимума на максимум.

Лабораторная работа № 2 Библиотека STL, ассоциативные контейнеры и потоковая итераторы

Основное содержание работы.

Цель. Получить практические навыки работы с возможностями стандартной библиотеки шаблонов (STL), на примере ассоциативных контейнеров и потоковых итераторов.

Написать программу, в которой данные в ассоциативном контейнере будут обрабатываться алгоритмами из STL.

Краткие теоретические сведения.

Ассоциативные контейнеры

Данные в таких контейнерах располагаются не последовательно, доступ к ним осуществляется посредством ключей. **Ключи** — это просто номера строк, они обычно используются для выстраивания хранящихся элементов в определенном порядке и модифицируются контейнерами автоматически. Изменение значения ключей обычно не допускается.

В STL имеется два основных типа ассоциативных контейнеров: **множества** и **отображения**. И те, и другие контейнеры хранят данные в виде дерева, что позволяет осуществлять быструю вставку, удаление и поиск данных.

Множества проще и, возможно, из-за этого более популярны, чем отображения. Множество хранит набор элементов, содержащих ключи — атрибуты, используемые для упорядочивания данных. Некоторые авторы называют ключом весь объект, хранящийся в множестве, но мы будем употреблять термин **ключевой объект**, чтобы подчеркнуть, что атрибут, используемый для упорядочивания (ключ), — это не обязательно весь элемент данных. Правило упорядочивания элементов может быть задана отдельно через функтор.

```
set<int> intSet;     //создает множество значений int
```

Следует отметить два специальных метода для работы с множествами:

- метод `lower_bound()` берет в качестве аргумента значение того же типа, что и ключ. Он возвращает итератор, указывающий на первую запись множества, значение которой не меньше аргумента (что значит «не меньше», в каждом конкретном случае определяется конкретным функциональным объектом, используемым при определении множества).
- метод `upper_bound()` возвращает итератор, указывающий на элемент, значение которого больше, чем аргумент.

Отображение хранит пары объектов. Один кортеж в такой «базе данных» состоит из двух «атрибутов»: ключевой объект и целевой (ассоциированный) объект. То есть ключевой объект служит индексом, а

целевой — значением этого индекса. По ключу объекты упорядочены, если добавляется новый элемент в отображение он записывается сразу в нужную позицию. Само упорядочивание по умолчанию по возрастанию, либо можно задать другое через функтор.

```
map<int, string> strMap; // отображение с ключем int и значением string
```

Контейнеры отображение и множество разрешают сопоставлять только один ключ данному значению. С другой стороны, **мулътиотображения** и **мультимножества** позволяют хранить несколько ключей для одного значения. Например, слову «ключ» в толковом словаре может быть сопоставлено несколько статей.

```
multiset<employee> machinists; //создает мультимножество
```

Потоковые итераторы.

Потоковые итераторы позволяют интерпретировать файлы и устройства ввода/вывода (потоки cin и cout), как итераторы. А значит, можно использовать файлы и устройства ввода/вывода в качестве параметров алгоритмов.

Основное предназначение входных и выходных итераторов — как раз-таки поддержка классов потоковых итераторов. С их помощью можно осуществлять применение соответствующих алгоритмов напрямую к потокам ввода/вывода.

Потоковые итераторы — это, на самом деле, объекты шаблонных классов разных типов ввода/вывода.

Существует два потоковых итератора: ostream_iterator и istream_iterator.

Класс ostream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, который имеет дело с выходным итератором. В следующем небольшом примере мы используем его как параметр copy().

```
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
int main()
{
    int arr[] = { 10, 20, 30, 40, 50 };
    list<int> theList;
    for(int j=0; j<5; j++) //перенести массив в список
      theList.push_back( arr[j] );
    ostream_iterator<int> ositer(cout, ", "); //итератор
    //ostream
    cout << "\nСодержимое списка: ";
    copy(theList.begin(), theList.end(), ositer); //вывод списка
    cout << endl;
    return 0;
}
```

Аналогично для файлов

```
ofstream outfile("ITER.DAT"); //создание файлового объекта
ostream_iterator<int> ositer(outfile, " "); //итератор
//записать список в файл
copy(theList.begin(), theList.end(), ositer);
```

Класс istream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, работающего с входным итератором. На примере, как такие объекты могут являться сразу двумя аргументами алгоритма copy().

Введенные с клавиатуры (поток cin) числа в формате с плавающей запятой сохраняются в контейнере типа список.

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
int main()
{
    list<float> fList(5); // неинициализированный список
    cout << "\nВведите 5 чисел (типа float): ";
    // итераторы istream
    istream_iterator<float> cin_iter(cin); // cin
    istream_iterator<float>  end_of_stream;  //eos  (конец
потока)
    // копировать из cin в fList
    copy( cin_iter, end_of_stream, fList.begin() );
    cout << endl; // вывести fList
    ostream_iterator<float> ositer(cout, "--");
    copy(fList.begin(), fList.end(), ositer);
    cout << endl;
    return 0;
}
```

Аналогично для файлов

```
list<int> iList; // пустой список
ifstream infile("ITER.DAT"); // создать входной файловый
объект
istream_iterator<int> file_iter(infile); // файл
istream_iterator<int> end_of_stream; // eos (конец потока)
// копировать данные из входного файла в iList
copy( file_iter, end_of_stream, back_inserter(iList) );
```

Методические указания.

1. Для определения контейнера можно использовать следующую запись:

```
set<int> dSet;
set<double> dSet;
map <int, string> strMap;
map <int, double> strMap;
```

2. Для заполнения множества рекомендуется использовать ограниченное количество значений. Для генерации строк или слов можно использовать простое правило, генерируется длина, а после по длине генерируются символы (например, английские буквы коды 97 - 122).
3. Стоит обратить внимание что изменение элементов в множестве недопустимо, их можно удалять и добавлять нужные. По обработку всех элементов по заданию можно сделать через промежуточное множество, а затем сделать `swap()`.
4. Пример работы с множествами:

```
set<string, less<string> > organic;
// итератор множества
set<string, less<string> >::iterator iter;
organic.insert("Curine"); // вставка компонентов organic
organic.insert("Xanthine");
organic.insert("Curarine");
organic.insert("Melamine");
organic.insert("Cyanimide");
organic.insert("Phenol");
organic.insert("Aphrodine");
organic.insert("Imidazole");
organic.insert("Cinchonine");
organic.insert("Palmitamide");
organic.insert("Cyanimide");
iter = organic.begin(); // вывод множества
while( iter != organic.end() )
    cout << *iter++ << '\n';
string lower, upper; // вывод значений из диапазона
cout << "\nВведите диапазон (например, C Czz): ";
cin >> lower >> upper;
iter = organic.lower_bound(lower);
while( iter != organic.upper_bound(upper) )
    cout << *iter++ << '\n';
```

5. Пример работы с отображениями:

```
string name;
int pop;
string states[] = { "Wyoming", "Colorado", "Nevada",
"Montana", "Arizona", "Idaho"};
int pops[] = { 470, 2890, 800, 787, 2718, 944 };
map<string, int, less<string> > mapStates; //отображение
map<string, int, less<string> >::iterator iter; //итератор
for(int j=0; j<6; j++)
{
    name = states[j]; //получение данных из массивов
    pop = pops[j];
    mapStates[name] = pop; //занесение их в отображение
}
cout << "Введите название штата: "; //получение имени штата
cin >> name;
pop = mapStates[name]; //найти население штата
cout << "Население: " << pop << " 000\n";
cout << endl; //вывод всего отображения
for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
    cout << (*iter).first << ' ' << (*iter).second << "000\n";
```

Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи.
3. Описание использованных контейнеров и алгоритмов STL.
4. Скриншоты приложения.
5. Листинг демонстрационной программы.

Задание и варианты.

1. Разработать программу по заданию, сама обработка выполняется через алгоритмы STL.
2. Использовать для обработки как множество, так и отображение, данные заполняются случайным образом, как в первой работе. **Задание выполняется как для множества, так и для отображения.** В качестве элемента контейнера допускается использовать стандартные типы данных (рекомендуется использовать string, самый простой вариант это int), для тех кто хочет лучше изучить рекомендуется использовать свои типы данных, например как в предыдущей работе `bitset<32>`.
3. В зависимости от задания массив выводиться до и после обработки в удобном виде, как на **экран**, так и **файл**, **через потоковые итераторы**.

Варианты

Для работы с данными отображения, выполняется то же задание, что и в первой работе.

Для множества выполнить следующую обработку:

- Удалить элемент по позиции;
- Удалить элемент по значению;
- Увеличить все элементы на значение первого элемента.

Лабораторная работа № 3 Проектирование приложений, SOLID, паттерны проектирования

Основное содержание работы.

Цель. Получить практические навыки проектирования приложений, на примере веб-приложения.

Написать программу веб приложение, в которой используются современные подходы к проектированию приложений.

Краткие теоретические сведения.

Проектирование приложений

Проектирование приложений помогает разным специалистам, работающим над проектом (менеджер, руководитель, дизайнер, разработчик приложений), одинаково четко понимать задачи и представлять конечный продукт.

Создание прототипа позволяет структурировать идеи, предотвратить ошибки и избежать лишней работы на ранних стадиях разработки. А это, в свою очередь, ускорит процесс разработки приложения и уменьшит финансовые затраты. Подробнее с паттернами проектирования можно ознакомиться по ссылкам https://habr.com/ru/post/276593/, c# - https://metanit.com/sharp/patterns/.

При проектировании лучше всего ориентироваться на современные технологии, желательно проверенные временем. **Грамотные разработчики должны следить за изменением и развитием технологий и подходов в отрасли в которой работают**. Информационные технологии очень быстро меняются, появляются, развиваются и устаревают языки программирования, платформы разработки, фреймворки и т.д., но подходы к разработке сформированные и проверенные временем изменяются редко. Один из таких подходов к ООП разработке это SOLID. (В качестве шутки был написан Hello word с выполнением большинства требований https://gist.github.com/lolzballs/2152bc0f31ee0286b722 примерно 150 строк)

SOLID

SOLID - это аббревиатура от пяти принципов объектно-ориентированного программирования и проектирования, названия которых так удачно зашифровал в этом слове Роберт Мартин, в начале 2000-х.

- Принцип единственности ответственности (The Single Responsibility Principle)
- Принцип открытости/закрытости (The Open Closed Principle)
- Принцип замещения Лисков (The Liskov Substitution Principle)
- Принцип разделения интерфейса (The Interface Segregation Principle)
- Принцип инверсии зависимости (The Dependency Inversion Principle)

Подробнее можно почитать по ссылкам:
http://blog.muradovm.com/2012/03/solid.html
http://sergeyteplyakov.blogspot.com/2014/10/solid.html
https://ru.wikipedia.org/wiki/SOLID_(объектно-ориентированное_программирование)

Интерфейс

Интерфейс — **программная/синтаксическая структура**, определяющая отношение между объектами, которые разделяют определённое поведенческое множество и не связаны никак иначе. При проектировании классов, разработка интерфейса тождественна разработке спецификации (множества методов, которые каждый класс, использующий интерфейс, должен реализовывать).

C++ поддерживает множественное наследование и абстрактные классы, поэтому, как уже упоминалось выше, **отдельная синтаксическая конструкция для интерфейсов в этом языке не нужна**. Интерфейсы определяются при помощи абстрактных классов, а реализация интерфейса производится путём наследования этих классов, с использованием ключевого слова `virtual`. Пример определения интерфейса:

```
class iOpenable
{
    public:
    virtual ~iOpenable(){}

    virtual void open()=0;
    virtual void close()=0;
};
```

Использование интерфейса:

```
class Door: public iOpenable
{
    public:
    Door(){std::cout << "Door object created" << std::endl;}
    virtual ~Door(){}
    //Конкретизация методов интерфейса iOpenable для класса Door
    virtual void open(){std::cout << "Door opened" << std::endl;}
    virtual void close(){std::cout << "Door closed" << std::endl;}

    //Специфические для класса Door свойства и методы
    std::string mMaterial;
    std::string mColor;
    //...
};
```

Для чего нужны интерфейсы, главное это разделить доступные возможности (public методы) от их реализации. Главное отличие класса от интерфейса — в том, что класс состоит из интерфейса и реализации. Если с

классом может быть связана сущность, которая может быть реализована множеством способов, нужен интерфейс.

Для примера выше интерфейс «открываемый» `iOpenable` может быть у двери, бутылки, коробки и так далее.

В Java и C# не позволяет наследовать больше одного класса. В качестве альтернативы множественному наследованию существуют интерфейсы. В этой концепции класс может реализовать любой набор интерфейсов.

https://habr.com/ru/post/30444/

https://ru.wikipedia.org/wiki/Интерфейс_(объектно-ориентированное_программирование)

Фабрика

Фабричный метод, или виртуальный конструктор — порождающий шаблон проектирования, предоставляющий подклассам (дочерним классам) интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какого класса будет создан объект. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

Данную концепцию можно реализовать обычными if-else/switch-case, но идея в том, чтобы сделать производство объектов гибче. Подробнее можно почитать по ссылкам:

https://habr.com/ru/post/129202/

https://ru.wikipedia.org/wiki/Фабричный_метод_(шаблон_проектирования)

MVC

Model-View-Controller (MVC, «Модель-Представление-Контроллер», «Модель-Вид-Контроллер») — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

- **Модель** (Model) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.
- **Представление** (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели.
- **Контроллер** (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Концепция MVC была описана Трюгве Реенскаугом в 1978 году. Позже появились другие концепции.

Model-View-Presenter (**MVP**) — шаблон проектирования, производный от MVC, который используется в основном для построения пользовательского интерфейса. Элемент Presenter в данном шаблоне берёт на себя функциональность посредника (аналогично контроллеру в MVC) и отвечает за управление событиями пользовательского интерфейса (например, использование мыши) так же, как в других шаблонах обычно отвечает представление. https://ru.wikipedia.org/wiki/Model-View-Presenter

Model-View-ViewModel (**MVVM**) — шаблон проектирования архитектуры приложения. Представлен в 2005 году Джоном Госсманом (John Gossman) как модификация шаблона Presentation Model. Ориентирован на современные платформы разработки, такие как Windows Presentation Foundation (WPF), Silverlight от компании Microsoft, ZK framework. https://ru.wikipedia.org/wiki/Model-View-ViewModel

https://habr.com/ru/post/215605/

Разработка веб приложения

Веб-приложение — клиент-серверное приложение, в котором клиент взаимодействует с веб-сервером при помощи браузера. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются межплатформенными службами. Подробнее https://ru.wikipedia.org/wiki/Веб-приложение, https://habr.com/ru/post/450282/.

Так как интернет и браузер есть уже в большинстве персональных устройств, веб-приложения широко распространены. Для разработки веб приложений можно использовать C++, но это неудобно (можно использовать nginx и FastCGI https://habr.com/ru/post/236865/), как один из близких языков можно выделить C#. На связке ASP.NET и C# довольно удобный и доступный инструментарий разработки, например, VisualStudio.

Ресурсы для изучения ASP.NET Core и C#:

- https://metanit.com/sharp/aspnet5/
- https://metanit.com/sharp/mvc5/

Методические указания.

1. Для начала работы рекомендуется поставить Visual Studio 2019 или Visual Studio Code. Можно просто скачать виртуальную машину с уже установленной средой разработки https://developer.microsoft.com/ru-ru/windows/downloads/virtual-machines/.
2. В качестве шаблона для работы можно использовать ASP.NET Core Web Application и сам тип проекта Web Application или Web Application (MVC). Дополнительно можно выбрать стандартную аутентификацию. Подробнее https://metanit.com/sharp/aspnet5/1.2.php
3. В качестве подобных инструкций по созданию приложения по заданию можно использовать:
   https://metanit.com/sharp/aspnet5/3.1.php
   или с базой данных https://metanit.com/sharp/aspnet5/12.1.php

Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи.
3. Описание использованных технологий.
4. Скриншоты приложения
5. Листинг демонстрационной программы.

Задание и варианты.

1. Разработать веб-приложение по заданию на основе ASP.NET или любой другой, с использованием MVC.
2. Приложение должно содержать страницу с перечнем созданных записей (объектов) и, хотя бы одно действие для каждого из них, например, «подробнее», «купить» или подобное, открывающее другую страницу.
3. (дополнительно, не обязательно для выполнения) Хранить данные об записях (объектах) в базе данных.

Варианты

1. СТУДЕНТ имя – string курс – int пол – int(bool)	2. СЛУЖАЩИЙ имя – string возраст – int рабочий стаж – int	3. КАДРЫ имя – string номер цеха – int разряд – int
4. ИЗДЕЛИЕ имя – string шифр – string	5. КНИГА имя – string автор – string	6. ЭКЗАМЕН имя студента – string дата – int

количество – int	стоимость – float	оценка – int
7. АДРЕС имя – string улица – string номер дома – int	8. ТОВАР имя – string количество – int стоимость – double	9. КВИТАНЦИЯ номер – int дата – int сумма – double
10. КОМПАНИЯ имя – string начальник – string количество работающих – int	11. ДВИГАТЕЛЬ имя – string мощность – int стоимость – double	12. АВТОМОБИЛЬ марка – string мощность – int стоимость – double
13. СТРАНА имя – string форма правления – string площадь – double	14. ЖИВОТНОЕ имя – string класс – string средний вес – int	15. КОРАБЛЬ имя – string водоизмещение – int тип – string

Можно выполнить альтернативное задание по согласованию с преподавателем.

Лабораторная работа № 4 Разработка мобильных приложений

Основное содержание работы.

Цель. Получить практические навыки разработки мобильных приложений.

Написать программу для портативных устройств – мобильное приложение.

Краткие теоретические сведения.

Мобильные приложения

Мобильное приложение — программное обеспечение, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах, разработанное для конкретной платформы (iOS, Android, Windows Phone и т. д.). Многие мобильные приложения предустановлены на самом устройстве или могут быть загружены на него из онлайновых магазинов приложений, таких как App Store, Google Play, и других.

Мобильное приложение – это программный пакет, функционал и дизайн которого «заточен» под возможности мобильных платформ. Перечислим несколько основных плюсов приложения:

- Интерфейс программы создан конкретно под работу на мобильном устройстве через сенсорный экран или кнопки;
- Удобная и понятная для пользователей гаджетов навигация, мобильное меню;
- Лучшее взаимодействие с пользователем через сообщения, пуш-уведомления, напоминания. Приложение может выполнять функции даже в фоновом режиме, чего нельзя сказать о сайте. Для работы с программой не нужно открывать браузер, а многие приложения поддерживают ряд функций и при отключенном интернете;
- Хранение персональных данных пользователя. Эта функция расширяет возможности персонализации приложений. Например, вызывает такси на дом (прописка), записывает на прием к врачу по медицинскому полису и другие преимущества;
- Более гибкая обратная связь с компанией, сервисом;
- Можно задействовать больше ресурсов. Например, подключить геолокацию и вызывать машину в любую точку города;
- Приложения могут учитывать биологические ритмы человека и оповещать его о необходимости следовать режиму.

На самом деле функционал мобильных приложений уже давно превзошел адаптированные сайты. Сегодня можно скачать и установить на

смартфон программы для бизнеса, обучения, органайзеры с опциями напоминания, развлекательный контент, различные сервисные службы.

Принцип работы мобильного приложения

Мобильное приложение можно разделить на два больших блока – это **front-end** и **back-end**. Соответственно в часть Front-end входят компоненты и опции программы, с которой взаимодействует пользователь. Например, панель выбора, дашборд, настройки опций и прочее. Back-end – это скрытая часть, «задник». С этими компонентами взаимодействует разработчик посредством серверного софта. Иными словами, мобильное приложение напоминает сплит-систему, в которой одна часть находится на стороне пользователя – это Front-end, а другая на стороне разработчика – это Back-end. Само приложение запускается в виртуальной машине, предоставляющей API для разработчика, а виртуальная машина запускается на Linux или UNIX ядре. Огромное значение имеет безопасность данных пользователей, что следует учитывать при разработке.

Создание простейшего приложения состоит из нескольких этапов:

- проект в среде разработки;
- создание пользовательского интерфейса;
- добавление активностей, навигации и действий;
- тестирования приложения в эмуляторе;
- тестирования приложения на реальном устройстве.

Android-приложение состоит из четырёх компонентов. Каждый компонент — это точка входа, через которую система или пользователь может получить доступ.

- **Активность** (activity) — элементы интерактивного пользовательского интерфейса. Одна активность задействует другую и передаёт информацию о том, что намерен делать пользователь, через класс Intent (**намерения**). Активности подобны веб-страницам, а намерения — ссылкам между ними. Запуск приложения — это активность Main.
- **Сервис** (service) — универсальная точка входа для поддержания работы приложения в фоновом режиме. Этот компонент выполняет длительные операции или работу для удалённых процессов без визуального интерфейса.
- **Широковещательный приемник** (broadcast receiver) транслирует нескольким участникам намерения из приложения.
- **Поставщик содержимого** (content provider) управляет общим набором данных приложения из файловой системы, базы данных SQLite, интернета или другого хранилища.

Среда разработки

Сред разработки и инструментария, как и документации очень много, в том числе есть среда для визуального программирования, где не требуется знать конкретный язык программирования. Основные среды разработки описаны ниже.

Android Studio

Это официальная IDE (интегрированная среда разработки) для Android, созданная компанией Google. В состав входит множество инструментов, эмулятор, удаленный отладчик и так далее. Поддерживаются языки программирования: Java, C++, а также Kotlin, который с недавних пор тоже стал официальным языком Android.

AIDE

AIDE расшифровывается как “Android IDE” и он уникален тем, что работает на самом Android. Это означает, что вы можете создавать приложения, используя свой телефон или планшет, а затем тестировать их на этом же устройстве.

Xamarin в Visual Studio

Visual Studio — это IDE от Microsoft, поддерживающий ряд языков, включая C#, VB.net, JavaScript и многое другое. С помощью фреймворка Xamarin, который входит в Visual Studio, можно создавать кроссплатформенные приложения с помощью C#, а затем тестировать их на нескольких устройствах, подключенных к облаку. Это хороший и бесплатный выбор, если вы планируете выпустить приложение и для Android, и для IOS, но не горите желанием писать свой код дважды. Также он является отличным выбором для тех, кто уже знаком с C# и/или Visual Studio. Минусом является то, что Xamarin неудобен в использовании Java библиотек и, как и с любой другой альтернативой Android Studio, вы теряете поддержку Google и расширенные встроенные функции.

Eclipse

До появления Android Studio, в качестве основного инструмента для разработки Android-приложений, разработчики использовали Eclipse. Этот IDE поддерживает несколько различных языков программирования, в том числе и Java с Android SDK. В отличие от Android Studio, Eclipse не предлагает встроенную поддержку и требует более тщательной настройки. На данный момент Google отключила официальную поддержку, поэтому, на самом деле, нет никаких оснований использовать Eclipse, вместо Android Studio.

Intel XDK

Intel XDK позволяет легко разрабатывать кроссплатформенные мобильные приложения; включает в себя инструменты для создания, отладки и сборки ПО, а также эмулятор устройств; поддерживает разработку для Android, Apple iOS, Microsoft Windows 8, Tizen; поддерживает языки разработки: HTML5 и JavaScript.

Scratch

Скретч — визуальная событийно-ориентированная среда программирования, созданная для детей и подростков. Скретч 3.0 (текущая

версия) сделан на HTML5, используя движок WebGL, что даёт ему возможность работать на мобильных устройствах и планшетах. В 2008 году Скретч был портирован для микроконтроллерного модуля Arduino и носит название S4A.

Xcode

Xcode — интегрированная среда разработки (IDE) программного обеспечения для платформ macOS, iOS, watchOS и tvOS, разработанная корпорацией Apple. Пакет Xcode включает в себя изменённую версию свободного набора компиляторов GNU Compiler Collection и поддерживает языки C, C++, Objective-C, Swift, Java, AppleScript, Python и Ruby с различными моделями программирования, включая (но не ограничиваясь) Cocoa, Carbon и Java.

Методические указания.

1. Для начала работы рекомендуется поставить Visual Studio 2019 или Visual Studio Code. Можно просто скачать виртуальную машину с уже установленной средой разработки https://developer.microsoft.com/ru-ru/windows/downloads/virtual-machines/.
   Также можно использовать **Android Studio** и писать приложение на языке C++ https://developer.android.com/studio
2. В качестве шаблона для работы можно использовать **Mobile App (Xamarin.Forms** https://metanit.com/sharp/xamarin/1.1.php**)** и сам тип проекта Master Detail. Подробнее https://metanit.com/sharp/xamarin/2.1.php.
3. В качестве подобных инструкций по созданию приложения по заданию можно использовать:
   https://metanit.com/sharp/xamarin/3.16.php
   или с базой данных https://metanit.com/sharp/xamarin/7.2.php

Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи.
3. Описание использованных технологий.
4. Скриншоты приложения
5. Листинг демонстрационной программы.

Задание и варианты.

1. Разработать мобильное-приложение по заданию на основе Xamarin

или любой другой, для любой мобильной платформы (Android, iOS, …).

2. Приложение должно содержать активность (экран приложения) с перечнем созданных записей (объектов) и, хотя бы одно действие для каждого из них, например, «подробнее», «купить» или подобное, открывающее другую активность.
3. (дополнительно, не обязательно для выполнения) Хранить данные об записях (объектах) в базе данных.

Варианты

1. СТУДЕНТ имя – string курс – int пол – int(bool)	2. СЛУЖАЩИЙ имя – string возраст – int рабочий стаж – int	3. КАДРЫ имя – string номер цеха – int разряд – int
4. ИЗДЕЛИЕ имя – string шифр – string количество – int	5. КНИГА имя – string автор – string стоимость – float	6. ЭКЗАМЕН имя студента – string дата – int оценка – int
7. АДРЕС имя – string улица – string номер дома – int	8. ТОВАР имя – string количество – int стоимость – double	9. КВИТАНЦИЯ номер – int дата – int сумма – double
10. КОМПАНИЯ имя – string начальник – string количество работающих – int	11. ДВИГАТЕЛЬ имя – string мощность – int стоимость – double	12. АВТОМОБИЛЬ марка – string мощность – int стоимость – double
13. СТРАНА имя – string форма правления – string площадь – double	14. ЖИВОТНОЕ имя – string класс – string средний вес – int	15. КОРАБЛЬ имя – string водоизмещение – int тип – string

Можно выполнить альтернативное задание по согласованию с преподавателем.

Литература

1. Р. Лафоре. Объектно-ориентированное программирование в C++. - Питер, 2011 г., 928 с.
2. Б. Страуструп: Язык программирования C++. Специальное издание, - Бином, 2015 1136 с.
3. Бертран Мейер Объектно-ориентированное конструирование программных систем, - Русская Редакция, 2005 г., 1204 с.
4. Гамма Э.,Хелм Р., Джонсон Р. , Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования., - Питер, 2016 г., 366 с.
5. Вайсфельд М. Объектно-ориентированное мышление., - Питер, 2014 г., 304 с.
6. https://metanit.com/
7. https://habr.com