


Раздел «Язык Си» . OOP3-Proj1 :

- Проект или совместная работа над задачей
  - Статическая библиотека.
  - Как собрать проект на C++ и создать статическую библиотеку.
  - Как добавить функции в библиотеку.
  - Make-file для создания библиотеки.
    - Пример простого makefile
  -  Задачи

Проект или совместная работа над задачей

Если Вы решаете небольшую задачу, то ее, конечно, удобнее решать одному.

НО, если:

1. в задаче приходится выполнять множество действий
  2. задача требует особых подходов к реализации функций и эти подходы знают разные люди
  3. у Вас очень мало времени
  4. наработки для этой задачи Вы хотели бы использовать в будущем
- , то разумно представить реализацию задачи как программный проект.

Проект требует более подробную проработку следующих вопросов: какие типы данных, какие инструменты нужны или желательны для решения задачи.

Когда эти вопросы будут решены, необходимо описать типы данных и все функции-инструменты. Все участники проекта должны СТРОГО следовать этому описанию.

Задача о времени

Измерения сделаны в некоторый момент времени. Время может быть представлено в полном виде: **год-месяц-день час:минуты:секунды.наносекунды**, или же в формате **дни часы:минуты:секунды.наносекунды**. В последнем случае предполагаем, что сокращенная форма, например, только секунды и наносекунды не рассматривается.

Необходимо реализовать следующие возможности: Для решения нужно:

1. преобразовывать дату из текстового представления в числовое, желательно совместимое с системным, чтобы можно было пользоваться системными функциями:
  1. **time** – получение текущей даты (с точки зрения системы). Время представляется в секундах с 1970 года (**time\_t**).
  2. **localtime** – заполняется структура **struct tm** с преобразованием секунд в год, месяц, день и т.д.
  3. **difftime** – вычисляется разница времен
  4. **strftime** – записывает дату в текстовую строку в удобном формате.
  5. **mktime** – преобразует дату из структуры **struct tm** в секунды **time\_t**
2. сравнивать два времени (
3. вычислять **разницу времен** в днях, часах и т.д. с точностью до наносекунд.
4. вычислять новую дату исходя из имеющейся и указанной **разницы времен**.
5. умножать **разницу времен** на число.
6. делить **разницу времен** на число.
7. вычислять частное от двух **разницу времен**
8. печатать результат

Эти инструменты можно будет использовать в других программах.

Для этого нужно создать заголовочные файлы, в которых будет описано: все структуры данных и **интерфейс** всех получившихся функций. Кроме того, нужно реализовать все функции так, чтобы можно было ими пользоваться без дополнительной компиляции.

Для решения этой проблемы нужно создать **проект**.

В проекте все создаваемые типы данных и интерфейсы функций описываются в заголовочных файлах. Простейший вариант проекта выглядит так:

Заголовочный файл (prim.h)	Файл реализации функций (prim.c)	Программа, которая использует функции (test.c)
<pre>// Новый :) тип данных typedef int Coord; // Интерфейс функции int add(Coord,Coord);</pre>	<pre>//включить заголовочный файл #include "prim.h" // Реализация функции add int add(int a, int b){     return a + b; };</pre>	<pre>// Использование функции add #include &lt;stdio.h&gt; //включаем НАШ заголовочный файл #include "prim.h"  int main(){     int a, b, c;     scanf("%d%d", &amp;a, &amp;b);     // Используем функцию add     c = add(a, b);     printf("a + b = %d\n",c); }</pre>


Как собрать проект\*

Конечно, все написанное должно быть откомпилировано и собрано.

Можно сразу скомпилировать все нужные файлы и собрать в работающий файл, а можно это сделать по частям.

Перый способ (все вместе)

```
gcc test.c prim.c -o test
```

 Заголовочный файл НИКОГДА в строку компиляции не включается. Компилятор ищет его самостоятельно.

Второй способ (по-отдельности)

```
>gcc -c prim.c
>gcc -c test.c
>gcc prim.o test.o -o test
```

В этом случае получаются два объектных файла (.o). Если с файл не изменялся, то его не обязательно компилировать.

Поиск

Раздел «Язык Си»

Главная

Зачем учить C?

Определения

Инструменты:

Поиск

Изменения

Index

Статистика

Разделы

Информация

Алгоритмы

Язык Си

Язык Ruby

Язык Ассемблера

EI Judge

Парадигмы

Образование

Сети

Objective C

Logon>>

Вернемся к задаче про о времени. Напишем все интерфейсы на языке C++. Будем использовать два представления времени: полная дата – **DateTime** и разница времен – **CTime**.

Рассмотрим сначала реализацию **CTime**:

Описание разницы времен

#### Заголовочный файл (timedat.h)

```
// заголовочные файлы - нужны функции,
// объявленные в них

// ввод/вывод в C++ на консоль
#include <iostream>
// стандартная библиотека
#include <cstdlib>
// работа с файлами в C++
#include <fstream>
// функции времени (системные)
#include <ctime>
// ввод/вывод в C. Иногда удобнее
#include <stdio.h>
// переменная errno для ловли системных
// неудач
#include <errno.h>
// работа со строками в C
#include <string.h>

#define TLimit 315360000
//для сокращенного вызова функций из std
using namespace std;

// название класса
class CTime{
// атрибуты (поля) класса
time_t ctm; // для секунд даты
int nanos; // наносекунды
//используем функцию clock
double start; // начало отсчета
double end; // конец отсчета
public:

// конструктор объекта класса
// всегда называется как класс
// это функция. По умолчанию просто выделяется
// память под объект, но конструктор может после этого
// выполнить еще дополнительные действия
CTime();
// в C++ разные по содержанию функции могут иметь
// одно имя в одном классе или одном пространстве имен
// но они должны отличаться своими параметрами
CTime( const CTime&);

//получение разницы времен в виде строки
void getTime(const char*);
// получение разницы времен как секунд
void getTime(time_t,int);

// в C++ можно ПЕРЕГРУЗИТЬ существующие операторы
// То есть такой оператор теперь будет работать с объектами
// класса CTime (обычный не может) и так как нам нужно

// это оператор сравнения двух объектов CTime
// & - ссылка передает адрес так же как и указатель,
// но работать с объектом, который передан по ссылке
// можно как с обычным (синтаксически)
int operator==(const CTime&);

//это для самостоятельной реализации
int operator>(CTime);
int operator>=(CTime);
int operator<(CTime);
int operator<=(CTime);

// оператор сложения двух разниц времен
CTime operator+(const CTime&);

// разница
CTime operator-(const CTime&);

// умножение на число
CTime operator*(float);

// частное двух объектов CTime
float operator/(const CTime&);
void print();
};
```

#### Файл реализации функций (ctime.cpp)

```
// включаем НАШ заголовок
#include "timedat.h"

// описание конструктора
CTime::CTime(){
// инициализируем все атрибуты

// для ВСЕХ функций класса CTime
// его атрибуты - ГЛОБАЛЬНЫЕ
ctm = 0;
nanos = 0;
};

// иногда нужно получить объект
// копированием, например
// CTime a(b);
// вот для этого такой конструктор
CTime::CTime(const CTime& a){
ctm = a.ctm;
nanos = a.nanos;
};

// получение времени через строку
void CTime::getTime(const char* s){
int day,h,min,sec;

// здесь удобно использовать C-ишный sscanf
sscanf(s,"%d:%d:%d.%d",&day,&h,&min,&sec,&nanos);

// nanos уже получил значения, вычисляем ctm
// nanos и ctm - ГЛОБАЛЬНЫЕ переменные для всех функций
// CTime::функция
ctm = (((day * 24) + h) * 60 + min) * 60 + sec;
};

// печать даты
void CTime::print(){
int day = ctm / 86400;
int h = (ctm / 3600) % 24;
int min = (ctm / 60) % 60;
int sec = ctm % 60;
printf("%d %d:%d:%d.%d ", day, h, min, sec, nanos);
};

CTime CTime::operator+(const CTime& a){
// объект, который будет возвращен
CTime tmp;

//tmp и a - объекты класса CTime, значит
// функции этого класса имеют доступ
// ко ВСЕМ атрибутам и функциям CTime

tmp.nanos = (nanos + a.nanos) % 1000;
tmp.ctm = ctm + a.ctm + (nanos + a.nanos) / 1000;

// возвращаем заполненный объект
// будет возвращена КОПИЯ этого объекта
return tmp;
};
```

= Файл тестирования (использования функций)=

```
#include "timedat.h"

int main(int argc, char** argv){
char ss[100];
FILE *fin;
if(argc > 1){
fin = fopen(argv[1],"r");
if( errno ){
perror("file open: ");
exit(1);
}
}
```

```

    }
    // объявляем три объекта (переменных) CTime
    CTime a,b,c;

    // получаем время как строки
    // Если бы код был написан на C,
    // то функция выглядела бы так:
    // getTime(&a,ss); - два аргумента
    // или так: a = getTime(ss);
    a.getTime(ss);

    // печатаем для проверки
    a.print();
    fgets(ss,99,fin);
    b.getTime(ss);
    b.print();
    // проверяем оператор сложения
    // это сокращенная запись оператора
    // его ПОЛНАЯ запись: c = a.operator+(b)
    c = a + b;
    // печать результат
    c.print();

    return 0;
}

```

## Статическая библиотека.

Допустим, Вы написали и отладили все функции, и они прекрасно работают. Кроме этой задачи есть еще множество других задач, в которых эти функции будут полезны.

Писать их заново или переносить файлы в новый проект хлопотно и неразумно.

Чтобы избежать этого скомпилированные функции обычно включают в **библиотеки**. Рассмотрим использование **статической библиотеки**. Функции статической библиотеки при линковке включаются в исполняемый файл (все функции из библиотеки). Это влияет на размер исполняемого файла. Именно поэтому при создании библиотеки нужно придерживаться принципа "ничего лишнего".

💡 В библиотеку добавляются ТОЛЬКО ОТЛАЖЕННЫЕ ФУНКЦИИ!!!!

При создании библиотеки используется архиватор **ar**.

## Как собрать проект на C++ и создать статическую библиотеку

При создании библиотеки используется архиватор **ar**.

### Создание библиотеки

1. создание объектных файлов реализаций функций
2. создание архива с названием **libназвание\_библиотеки.a**
3. включение объектных файлов в архив
4. ранжирование архива для быстрого поиска функций
5. помещение библиотечного файла в специальный каталог

Что делаем	Комментарий
<code>&gt;g++ -c ctime.cpp</code>	получаем объектный код функций класса
<code>&gt;ar -rc libctime.a ctime.o</code>	создаем архив libctime.a и добавляем в него содержимое ctime.o. Это уже архив, но еще не библиотека
<code>&gt;ranlib libline.a</code>	ранжируем (индексируем) функции в архиве. Теперь это уже статическая библиотека.

Как правило, заголовочные файлы проекта собираются в каталог **include**, а библиотечные – в **lib**.

При компиляции и линковке должны использоваться ключи: **-Iкаталог\_с\_заголовками**, **-Lкаталог\_с\_библиотечными\_файлами** и **-lназвание\_библиотеки**

Далее создаем каталоги для заголовков и библиотек, помещаем их туда. После этого можно уже компилировать программу, которая использует функции из библиотеки.

```

>mkdir lib
>mkdir include
>mv libctime.a lib/
>mv timedat.h include/

```

### Компиляция

```
>g++ ctest.cpp -o ctest -I./include -L./lib -lctime -lm
```

Одна используемая библиотека – наша библиотека **ctime**, а вторая – системная математическая **m**, но это только для примера, функции из нее не нужны сейчас.

## Описание даты (полный формат)

Заголовочный файл (timedat.h) – обавлено описание еще одного класса	Файл реализации функций (dtime.cpp)
<pre> #include &lt;iostream&gt; #include &lt;cstdlib&gt; #include &lt;fstream&gt; #include &lt;ctime&gt; #include &lt;sstream&gt; #include &lt;stdio.h&gt; #include &lt;errno.h&gt; #include &lt;string.h&gt; #define TLimit 315360000 using namespace std; </pre>	<pre> // также включается НАШ // заголовочный файл #include "timedat.h"  // конструктор для класса DateTime DateTime::DateTime(){ // все атрибуты ставим в 0 ctm = 0; nanos = 0; bzero(&amp;tmdat,sizeof(struct tm)); </pre>

```

class CTime{
time_t ctm;
int nanos;
public:
    CTime();
    CTime( const CTime&);

    // для самостоятельного программирования
    // для определения длительности процесса
    void start();
    void stop();

    void getTime(const char*);
    void getTime(time_t,int);
    int operator==(const CTime&);

    // для самостоятельного программирования
    int operator>(CTime);
    int operator<(CTime);

    CTime operator+(const CTime&);

    // для самостоятельного программирования
    CTime operator-(const CTime&);
    CTime operator*(float);
    float operator/(const CTime&);
    void print();
};

// Добавили описание класса DateTime
class DateTime{
    // структура для времени (системная)
    struct tm tmdat;
    // секунды с 1970 г.
    time_t ctm;
    //наносекунды
    int nanos;
public:
    // конструктор
    DateTime();
    // инициализирующий конструктор
    DateTime(time_t, int);
    DateTime(char*);

    // получить дату из строки
    void getData(char*);

    // получить текущую дату с заполнением структуры
    void now();

    // сравнение двух дат
    int operator==(const DateTime&);

    // сложение даты с CTime
    DateTime operator+(const CTime&);

    // для самостоятельного программирования
    //вычитание CTime
    DateTime operator-(const CTime&);

    // для самостоятельного программирования
    // получить разницу времен
    CTime operator-(const DateTime&);
    // печать
    void print();

    // для самостоятельного программирования
    // печать с указанием дня недели
    // как принято в России
    void print(int);
};

```

```

};

// получение даты из строки
void DateTime::getData(char* s){
    int y,mon,day,h,min,sec;
    //здесь удобнее разобрать строку sscanfом
    sscanf(s,"%d-%d-%d %d:%d:%d",&y,&mon,&day,&h,&min,&sec,&nanos);
    // год в структуре хранится начиная с 1900, поэтому его нужно вычесть
    tmdat.tm_year = y - 1900;
    // месяцы начинаюс 0
    tmdat.tm_mon = mon - 1;
    tmdat.tm_mday = day;
    // наше время отличается на 3 часа
    // (есть проблемы с летним европейским временем)
    tmdat.tm_hour = h + 3;
    tmdat.tm_min = min;
    tmdat.tm_sec = sec;
    // преобразование заполненной структуры в
    // секунду. Заодно вычисляются и заполняются
    // оставленные нами поля tmdat (см. man)
    ctm = mktime(&tmdat);
};

// получение даты секунды и наносекунды
DateTime::DateTime(time_t a, int nn){
    ctm = a;
    nanos = nn;
};

// Вычитание двух дат. Сложение двух дат в таком формате
// бессмысленно.
// Результатом сложения является объект CTime - разница времен
CTime DateTime::operator-(const DateTime& a){
    CTime tmp; // для возврата
    // вычисляем разницу в секундах
    time_t sec = ctm - a.ctm - 1;
    int nn = 1000 + nanos - a.nanos;

    // задаем время для объекта CTime секундами и наносекундами
    // к полям ctm и nanos класса CTime
    // функции класса DateTime доступа не имеют!!
    tmp.getTime(sec + nn / 1000, nn % 1000);
    // возвращаем объект
    return tmp;
};

// Печать даты в удобном виде
void DateTime::print(){
    // временная структура
    struct tm tmp;
    // строка ддля представления даты
    char buffer[100];

    // заполнение временной структуры
    // обычно localtime создает динамическую
    // переменную и возвращает указатель на нее.
    // Таким образом все функции времени работают с
    // этой переменной (статическая).
    // Для этой задачи нужна локальная переменная,
    // поэтому сразу получаем ЗНАЧЕНИЕ
    tmp = *localtime(&ctm);
    // получение текстовой строки с датой в нужном формате
    strftime(buffer,80,"%y-%m-%d %X",&tmp);
    // печать результата
    printf("%s.%d\n",buffer,nanos);
};

```

#### Проверка работы функций. Файл ==dtest.cpp==

```

#include "timedat.h"

int main(int argc, char** argv){
    char ss[100];//="13 7:25:10.9876";
    FILE *fin;
    if(argc > 1){
        fin = fopen(argv[1],"r");
        if( errno ){
            perror("file open: ");
            exit(1);
        }
    }
    // в файле две строки с описанием даты
    // строго по формату
    fgets(ss,99,fin);
    DateTime dt;
    dt.getData(ss);
    dt.print();
    fgets(ss,99,fin);
    DateTime dt1(ss);
    dt1.print();
    CTime res;
    // получение результата вычитания
    // здесь предполагается, что первое время больше
    res = dt - dt1;
}

```

```
res.print();
return 0;
}
```

### Как добавить функции в библиотеку.

Вот реализованы новые функции. Можно создать для них другую библиотеку. Но сейчас нужно добавить их в библиотеку **ctime**.

Что делаем	Комментарий
<code>&gt;g++ -c dttime.cpp</code>	получаем объектный код функций класса
<code>&gt;ar -r libctime.a dttime.o</code>	В существующий архив libctime.a и добавляем содержимое dttime.o с ключом <b>-r</b> , но без <b>-с</b> . <b>-r</b> добавляет новые функции и перезаписывает старые с такими же именами. <b>-с</b> – ключ создания архива (см. man).
<code>&gt;ranlib libline.a</code>	Снова ранжируем (индексируем) функции в архиве. Теперь это уже статическая библиотека.

Заголовочный файл в папке **include**, библиотечный переписываем в папку **lib** Компилируем программу:

```
>g++ dtest.cpp -I./include -o dtest -L./lib -lctime
```

### Make-file для создания библиотеки.

Если вы написали задачу со множеством файлов-источников, и этот проект приходится собирать в разных местах, то разумнее автоматизировать этот процесс.

Для этого используется команда **make**. Правда для того, чтобы она выполнила все что нужно, необходимо написать файл с **правилами**. Понятно, что компиляция и сборка проекта из множества файлов будет сложной. При наличии исходных текстов некоторые из них должны быть помещены в библиотеки статические или динамические, некоторые должны быть использованы для создания запускаемых приложений и т.д. Прописать все это словами для человека который не участвовал в разработке проекта будет почти невозможным. Вероятность того, что что-нибудь будет забыто, а что-то перепутано очень велика. Таким образом приложение не соберется.

Для решения этой проблемы существует утилита `==make==` (она существует и для WINDOWS, и для Linux). Эта утилита работает с **makefile**, в котором специальным образом прописаны правила сборки проекта, зависимости и т.д.

Если просто запустить `\textit{make}`, то эта программа постарается найти файл **makefile** и выполнить **правила**, которые записаны в нем. Если же нужно указать другой файл с инструкциями, то тогда

```
>make -f makefile.drugoy
```

В предыдущих разделах мы уже рассматривали как получается готовое приложение: создавали объектные файлы, собирали библиотечные файлы, и потом из всего этого получали готовое приложение.

Желательно, чтобы все это выполнялось автоматически.

Для начала заметим, что в каталоге **src** лежит запакованный файл **myprog.tar**. Этот файл получен с помощью архиватора **tar**.

```
>tar -zcvf prim.tar ctime.cpp dttime.cpp timedat.h dtest.cpp
```

Чтобы распаковать его, нужно сделать следующее:

```
>cd src
>tar -zxf prim.tar
```

Вот теперь имеются все необходимые файлы и каталоги для создания проекта. Осталось их собрать. При отладке приходится часто компилировать и собирать одно и то же, поэтому удобно иметь **makefile**. К тому же он нужен для **vim**.

Напишем инструкции для **makefile**, чтобы сборка прошла автоматически.

В предыдущих разделах мы уже рассматривали как получается готовое приложение: создавали объектные файлы, собирали библиотечные файлы, и потом из всего этого получали готовое приложение.

Желательно, чтобы все это выполнялось автоматически.

Основные части **makefile**:

```
[цель]:зависимости
[tab]команды
....
[tab]команды
```

Цель, это то, что мы хотим, в конечном счете, получить после работы **make**, а зависимости – это то, что необходимо для получения результата.

### Пример простого makefile

```
# Наша главная цель -
# получить исполняемый
# файл test

all: test

# от наличия каких файлов зависит test
test: ctime.o dttime.o

# что нужно выполнить, чтобы получить test, если они есть
# для команд [tab] вначале строки обязательно
# НЕ ПРОБЕЛЫ!!
g++ -o test ctime.o dttime.o dtest.cpp

# из чего получаем ctime.o
ctime.o: ctime.cpp
# как получаем
g++ -c ctime.cpp
```

```
# из чего получаем dtime.o
dtime.o: dtime.cpp
# как получаем
g++ -c dtime.cpp

# файлы .o нужно потом убрать
clean:
rm -rf *.o
```

Вызов:

```
> make
>make clean
```

Допустим теперь нужно создать и автоматически поместить заголовочные и библиотечные файлы на место. Пишем **makefile.lib**.

```
# так назвали проект
# это основная цель работы
all: myprog

# от чего зависит myprog
myprog: libdtime.a

# от чего зависит libdtime.a
libdtime.a: ctime.o dtime.o

# как получить
ar -cr libdtime.a ctime.o dtime.o
ranlib libdtime.a

ctime.o: ctime.cpp
g++ -c ctime.cpp

dtime.o: dtime.cpp
g++ -c dtime.cpp

# инсталляция - помещение в нужные каталоги
install: myprog
mkdir -p include
mkdir -p lib
mv timedat.h ./include
mv libdtime.a ./lib

# очистка
clean:
rm -rf *.o
```

Вызов:

```
> make -f makefile.lib
>make clean
```

### Задачи

1. Написать **makefile** для отладки функций классов.
2. Допisać и отладить оставшиеся функции классов.
3. системная функция **clock** считает такты процессора. Добавить две функции в класс **CTime** чтобы можно было получить разницу времен между началом и концом работы программы. Проверить работу.
4. Написать **makefile.lib** для получения библиотеки из исходников и инсталляции. Создать библиотеку реализованных функций.
5. Поменяться с соседом библиотеками и протестировать ее. За найденные ошибки – плюсы.
6. Написать программу на С по возведению числа в указанную степень разными способами. Использовать написанные библиотеки для определения времени запуска и завершения заботы программ. Эти времена записать в файл. Напечатать самую долгую и самую быструю программы

Примеры использования библиотек и классов для работы с системными файлами в файле **prim.tar**.

-- [TatyanaOvsyannikova2011](#) – 28 Sep 2017

Attachment	Action	Size	Date	Who	Comment
 prim.tar	<a href="#">manage</a>	1.2 K	28 Sep 2017 - 14:19	<a href="#">TatyanaOvsyannikova2011</a>	

(с) Материалы раздела "Язык Си" публикуются под лицензией GNU Free Documentation License.