Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания  
 к лабораторной работе № 2\_6  
 по курсу «Основы алгоритмизации   
и программирования»**

«Модульное программирование»

Преподаватель: Магеров В.В.

Составитель: Войтехович   
Агния Витольдовна

Полоцк, 2017ЦЕЛЬ РАБОТЫ

Изучить принципы построения модулей на языке С++, а также реализацию программ, включающих несколько модулей.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

# 1 Модульное программирование

Изначальной и, вероятно, все еще самой общей парадигмой программирования является следующая: «Решите, какие процедуры вы желаете; используйте лучшие из алгоритмов, которые можете найти». При этом внимание фокусируется на определении процедуры: выбор алгоритма, необходимого для выполнения желаемых вычислений.

Процедурное программирование использует функции для создания порядка в лабиринте алгоритмов. До сих пор Вы фактически изучали процедурное программирование.

Одной из важнейших составляющих процедурного программирования является **модульное программирование**. Главную идею модульного программирования можно сформулировать следующим образом.

***При решении задач выполняются одни и те же действия. Поэтому разумно реализовать такие операции один раз в виде некоторых модулей и в дальнейшем иметь возможность присоединять эти модули к своей программе.***

Как правило, модуль представляет собой совокупность функций и данных, используемых для решения задач определенного класса или реализующих набор операций над некоторым понятием. Например, в одном модуле могут быть определены функции обеспечивающие ввод-вывод. В другом модуле могут содержаться функции, позволяющие выполнять операции над строками. Третий модуль поддерживает выполнение операций над геометрическими объектами. А еще один, например, обеспечивает доступ к информации в некоторой базе данных.

Если при разработке программы мы обнаружим, что нам необходимо выполнять операции над строками и работать с геометрическими объектами, то нет необходимости реализовывать эти подпрограммы заново. Достаточно подключить соответствующие модули к нашей программе. Ну и, разумеется, необходимо знать, а каким образом надо обращаться к этим модулям, чтобы решить наши задачи.

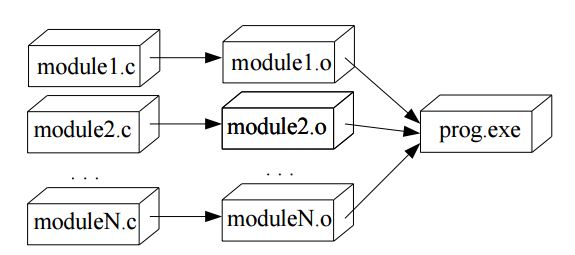
Каждый модуль, входящий в состав программы, решает задачи из своей области. Поэтому модули могут и должны разрабатываться отдельно друг от друга. Это позволяет вести коллективную разработку программной системы. Отдельные программисты или небольшие коллективы независимо от других разрабатывают свою часть программной системы. Кроме того, такой подход обеспечивает возможность модульной отладки, при которой отлаживается не вся программная система в целом, а отдельные ее компоненты. Такая технология обеспечивает быструю локализацию и устранение ошибок, что приводит к ускорению разработки программного продукта.

Основным стержнем, на котором держится модульное программирование, является аппарат подпрограмм (функций). Внутри модулей действия реализуются посредством определения соответствующих подпрограмм. В вызывающей программе содержатся фактически только вызовы необходимых подпрограмм.

Для того чтобы воспользоваться средствами, предоставляемыми модулем, мы должны иметь некую дополнительную информацию по этому модулю. Во-первых, мы должны знать, какие подпрограммы какие действия выполняют. Во-вторых, мы должны знать, какую информацию мы должны передать этим подпрограммам для решения задачи, и каким образом она должна быть передана. Таким образом, важным моментом при использовании модульного программирования является обеспечение взаимодействия между модулями, или как говорят, определение **интерфейса**. Описание интерфейса, как правило, приводят в специальном **заголовочном файле**. Передача информации между подпрограммами (функциями) модулей в большинстве случаев осуществляется через список параметров при вызове соответствующей подпрограммы. Иногда допускается использование глобальных переменных, но каждый такой случай должен быть обоснован.

Для рассматриваемой технологии характерна еще одна задача, а именно задача сборки программы из модулей. Существует несколько способов её решения, некоторые из которых рассматриваются ниже.

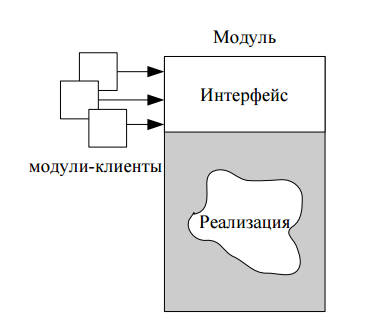
Наиболее распространённым способом сборки программы является сборка на уровне объектных файлов. Суть его состоит в том, что для каждого разработанного модуля, посредством компилятора с языка, получают отдельно объектный файл, а затем, используя компоновщик, объединяют их в единый исполняемый образ программы. Такие известные языки программирования как язык ассемблера, Си, C++, Фортран поддерживают данную технологию. Однако, например, не менее известный язык Паскаль в его классическом варианте такой возможности не предусматривает. На рисунке 1 приведена иллюстрация рассмотренного метода.



**Рисунок 1** – Сборка на уровне объектных файлов

# 2 . Характеристики модулей

Важнейшим принципом разработки модуля является ***принцип информационной закрытости.*** Этот принцип утверждает, что содержание модулей должно быть скрыто друг от друга. Модуль должен определяться и проектироваться так, чтобы его содержимое (функции и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам). Этот принцип иллюстрируется на рисунке 2.



**Рисунок 2** – Информационная закрытость модуля

Информационная закрытость означает, что все модули независимы и обмениваются только необходимой для работы информацией. Кроме этого доступ к операциям и структурам данных модуля извне его ограничен.

Следование принципу информационной закрытости даёт определённые преимущества. С одной стороны обеспечивается возможность разработки модулей независимыми разработчиками, что уменьшает время разработки программной системы. Но он даёт преимущества и в малых проектах, когда количество разработчиков невелико, или он вообще один. Это преимущество заключается в лёгкой модификации системы. При использовании этого принципа вероятность распространения ошибок сильно уменьшается, так как многие данные и функции локализованы внутри модуля и скрыты от других частей системы.

В этом случае модуль играет роль «чёрного ящика». Его содержимое (реализация) недоступно другим модулям, а управление им (интерфейс) простое. Изменение реализации модуля никоим образом не влияет на его пользователей.

Для оценки уровня информационной закрытости модуля можно использовать его внутреннюю характеристику - ***связность***. Под связностью модуля понимается мера зависимости его частей. Чем выше связность модуля, тем лучше результат проектирования. Ниже приведены различные типы связности в порядке её уменьшения. При этом первые три свидетельствуют о хорошем качестве модуля, а остальные о недостаточно высоком качестве разработки.

**Функциональная связность**. Части модуля вместе реализуют одну проблемную задачу (операцию). Например: вычисление синуса угла, проверка орфографии, вычисление зарплаты сотрудника, определение места пассажира. Модуль выполняет только то, для чего он предназначен. Поэтому модуль вычисления синуса не должен печатать его значение.

**Информационная связность.** Выходные данные одной части используются как входные данные в другой части модуля. Порядок выполнения действий строго определён и подобен конвейеру.

**Коммуникативная связность.** В этом случае части модуля связаны по данным, то есть они работают с одним и тем же типом структуры данных. При этом порядок выполнения действий безразличен.

**Процедурная связность**. Этот тип связности соответствует случаю, когда части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения. При этом зависимость по данным между частями модуля отсутствует.

**Временная связность**. Различные части модуля не связаны между собой ни по данным, ни по порядку выполнения, но необходимы в один и тот же период работы системы.

**Логическая связность.** В этом случае части модуля объединены по принципу функционального подобия. Например, модуль включает в себя различные функции обработки ошибок. К серьёзным недостаткам таких модулей относится сложное сопряжение с другими модулями и большая вероятность внесения ошибок при необходимости изменений.

**Связность по совпадению**. В модуле отсутствуют явно выраженные внутренние связи вообще.

Для оценки меры взаимозависимости модулей по данным применяется другая характеристика – ***сцепление***. Сцепление является внешней характеристикой модуля, которую желательно уменьшать. Выделяют несколько типов сцепления, которые перечислены ниже в порядке его увеличения.

**Сцепление по данным**. В этом случае функции модуля А вызывают функции модуля B, причём все входные и выходные параметры вызываемого модуля являются простыми элементами данных.

**Сцепление по образцу.** Здесь в качестве параметров используются структуры данных.

**Сцепление по управлению**. Один модуль явно управляет функционированием другого модуля с помощью флагов или переключателей, посылая ему управляющие данные.

**Сцепление по внешним ссылкам.** В этом случае оба модуля ссылаются на один и тот же глобальный элемент данных.

**Сцепление по общей области.** Здесь модули разделяют одну и ту же глобальную структуру данных.

**Сцепление по содержанию**. Один модуль прямо ссылается на содержание другого модуля (в обход интерфейсной части вызываемого модуля).

# 3 Соглашения по разработке модулей

За время существования такой дисциплины как программирование, в ней были выработаны определенные правила оформления программных модулей. Подавляющее большинство разработчиков программного обеспечения стремятся их соблюдать. Следование этим правилам позволит получить хорошо структурированную, легко читаемую программу. Использование одних и тех же правил оформления модулей позволяет легче и быстрее разобраться в них, значительно уменьшает количество «необязательных» ошибок. Те же, кто эти правила совершенно игнорируют, всех этих преимуществ лишены. На этапе обучения следование приведённым ниже правилам **обязательно**. Вместе с тем нельзя рассматривать их как некие догмы. Это всего лишь рекомендации. В процессе трудовой деятельности вы вынуждены будете их несколько адаптировать в соответствии с требованиями трудовой организации, компании.

– **Проектируйте модуль со связностью не ниже коммуникативной**. Он не должен быть просто совокупностью никак не связанных между собой функций и данных.

– Модуль должен охватывать, как можно большую часть задач рассматриваемой области, и в то же время **набор доступных подпрограмм следует сделать минимальным и простым в использовании.**

– **Добейтесь, чтобы сцепление модуля было по данным или по образцу**. В противном случае изменение некоторого модуля может вызвать лавинообразные изменения в других модулях.

– При разработке модуля с именем ***Name***, как правило, создается два файла: файл-заголовок ***Name.h*** и файл-реализация ***Name.cpp***. В файле-заголовке определяется интерфейс к данному модулю. Он включает в себя определение констант, новых типов данных, описаний функций с обязательным указанием их параметров и описаний глобальных переменных, определенных в модуле и доступных для других частей программы. Если модуль «***A***» обращается к модулю «***B***», то в модуле «***A***» должен быть подключен файл-заголовок модуля «***B***». Поскольку файл-заголовок может подключаться несколько раз, то во избежание дублирования используют конструкцию следующего вида:

#ifndef name\_h

#define name\_h

//...

//описания функций, глобальных переменных, констант

//...

#endif

Файл-реализация ***Name.cpp*** содержит определения глобальных и статических переменных модуля, определение новых типов данных, используемых только внутри этого модуля и реализации функций. Очевидно, что в этом модуле должно производиться подключение файла-заголовка.

– **Минимизируйте использование глобальных переменных. Наилучшим решением является отказ от их использования**. Поскольку глобальные переменные будут доступными из любого модуля, то чрезвычайно трудно контролировать их использование. Настоятельно рекомендуется передавать в функцию все необходимые ей данные в качестве параметров.

– **Избегайте слишком больших реализаций функций.** Например, можно потребовать, чтобы вся реализация функции умещалась в определённое количество строк (количество строк на экране монитора, в пределах одного листа при печати и т .д.). Кроме этого важно избегать глубоких вложений внутри функции (не более 2 вложенных операторов). Игнорирование этой рекомендации приводит к более трудному восприятию программного текста, реализующего данную функцию, а что еще более важно - к повышению вероятности совершить ошибку при кодировании алгоритма.

– Каждая строка программы должна содержать не более одного оператора. Также старайтесь избегать объявления нескольких переменных на одной строке.

– Программный код пишется для чтения не компилятором, а человеком. Программный код должен сам себя документировать и средства языка C позволяют обеспечить выполнение этого требования. Как минимум, все используемые в программах имена должны быть содержательными. Понять смысл оператора **digitCount = digitCount + 2** гораздо проще, чем **c = c + 2.**

– Хорошая практика программирования состоит в том, чтобы **всем явным константам давать символические имена и их использовать в программе**. Указанное правило не относится к константам, которые даже в принципе не могут поменяться. Например, индексация элементов в векторе начинается с 0, поэтому определение для этой константы символического имени бессмысленно и только запутывает программный код.

– **Используйте отступы для выделения** операторов и их блоков внутри условного оператора, оператора цикла и выбора. Следование этой рекомендации позволяет избежать ошибок, связанных, например, с расстановкой скобок { и }. Но еще более важно, что при таком подходе четко прослеживается структура программы.

# 4 Пример: математические операции

В качестве примера, иллюстрирующего «хороший стиль» программирования рассмотрим следующую задачу. Пусть требуется реализовать набор функций для выполнения операций над двумя целыми числами. Первая функция – это сумма двух целых чисел, вторая – результат деления первого числа на второе (возвращает вещественное число), третья возвращает среднее арифметическое из двух чисел, умноженное на число π, а четвертая функция возводит первое число в степень, равную второму числу.

Конечная программа будет состоять из четырех модулей. Основной модуль – ***main*** – именно с него начинается работа программы и из этой функции происходит вызов функций из дополнительных модулей. Данный модуль будет представлен файлом **main.cpp**.

Дополнительный модуль первого уровня – ***menu***. Именно этот модуль будет подключен к main.c и в нем будет реализовано пользовательское меню. Этот модуль будет представлен заголовочным файлом **menu.h** и файлом-реализацией **menu.cpp.**

Оставшийся недостающий функционал можно разделить на 2 категории:

- математические операции;

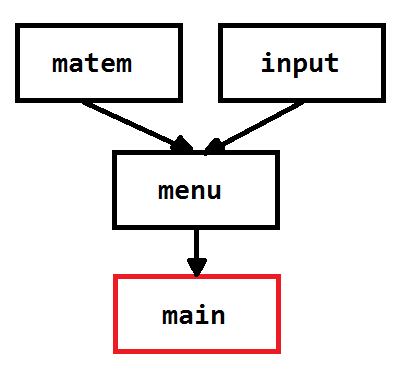
- операции ввода/вывода данных.

Для реализации этих функций создадим два модуля второго уровня:

- модуль ***matem***, состоящий из двух файлов – **matem.h** и **matem.cpp**. в котором реализованы математические функции;

- модуль ***input***, состоящий из файлов **input.h** и **input.cpp,** в которых будут реализованы функции форматированного ввода исходных данных и вывода результатов.

На рисунке 3 представлена модульная схема разрабатываемой программы.



**Рисунок 3** – Модульная схема разрабатываемой программы

Далее представлены листинги всех вышеперечисленных файлов.

## 4.1 main.cpp

#include <clocale>

#include <iostream>

#include "menu.h"

using namespace std;

int main(void)

{

setlocale(LC\_ALL,"");

cout << "Вас приветствует модульная программа"<< endl;

menu();

cout << "Спасибо за внимание!"<<endl;

}

## 4.2 menu.h

#ifndef MENU\_H

#define MENU\_H

using namespace std;

void menu (void);

#endif

## 4.3 menu.cpp

#include "matem.h"

#include "input.h"

void menu (void){

int q = 9;

while (q>0)

{ int a=0, b=0;

cout << "выберите функцию: \n\t1-(a+b)\n\t2-(a/b)"<< endl;

cout<<"\t3-((a+b)/2)\*pi\n\t4-a^b\n\t0- выход"<<endl;

cin >> q;

switch(q){

case 1: {

input(&a,&b);

outputInt(sum(a,b), "sum");

break;

}

case 2: {

input(&a,&b);

outputFloat(del(a,b), "del");

break;

}

case 3: {

input(&a,&b);

outputFloat(func1(a,b), "func1");

break;

}

case 4: {

input(&a,&b);

outputInt(func2(a,b), "func2");

break;

}

default: cout<<"Неверный ввод!"<<endl;

} }

return; }

## 4.4 matem.h

#ifndef MATEM\_H

#define MATEM\_H

#include <iostream>

using namespace std;

#define PI 3.1415

int sum (int a, int b) ;

float del (int a, int b) ;

float func1(int a,int b) ;

int func2(int a,int b);

#endif

## 4.5 matem.cpp

#include "matem.h"

int sum (int a, int b) {

return (a+b);

}

float del (int a, int b) {

float res;

res = a/(float) b;

return res;

}

float func1(int a,int b) {

return (del(sum (a,b), 2) \* PI);

}

int func2(int a,int b) {

int res = 1;

for (int i = 1; i<=b ; i++)

{

res \*= a;

}

return res;

}

## 4.6 input.h

#ifndef INPUT\_H

#define INPUT\_H

#include <iostream>

using namespace std;

void input (int\* a, int\* b);

void outputInt(int x, char\* str );

void outputFloat(float x, char\* str );

#endif

## 4.7 input.cpp

#include "input.h"

void input (int\* a, int\* b) {

cout << "Введите а и b: ";

cin >> \*a >> \*b;

}

void outputInt(int x, char\* str ){

cout << "результат выполнения функции " << str;

cout << " : " << x << endl<<endl;

}

void outputFloat(float x, char\* str ){

cout << "результат выполнения функции " << str;

cout << " : " << x << endl<<endl;

}

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ № 2\_6

**Задание 1.1** Создайте новый модуль для меню.

**Задание 1.2** Создайте новый модуль для функций ввода/вывода данных.

**Задание 1.3** Создайте новый модуль для функций работы с файлами.

**Задание 2** Перенесите пользовательские функции, реализованные в предыдущих лабораторных работах, в соответствующие модули.

**Задание 3** Оформите заголовочные файлы для всех модулей.

**Задание 4** Подключите модули друг к другу в соответствии с тем, каким образом они будут взаимодействовать.

**Задание 5** Нарисуйте модульную схему вашего приложения.