

Лекция 8

*Модульность — фундаментальный аспект
всех успешно работающих крупных систем.*

Б. Страуструп

Модульное программирование.

Модульное программирование

При увеличении объема кода программы человеку становится сложно удерживать все подробности работы в своей памяти. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части. В C++ это может быть сделано с помощью функций, описывающих простые и обозримые задачи. Тогда всю программу можно рассматривать в более укрупненном виде – на уровне взаимодействия функций. Это важно, поскольку человек способен одновременно работать с ограниченным количеством фактов. Использование функций – первый шаг к повышению степени абстракции программы и ведет к упрощению понимания ее структуры.

Использование функций позволяет также уйти от избыточности кода, поскольку функцию записывают один раз, а вызывать ее можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. При разработке могут использоваться функции-заглушки. Таким образом создаются более простые в отладке и сопровождении программы.

Модульное программирование

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые отдельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (или разными программистами). Модуль содержит данные и функции их обработки. Другим модулям нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого модуля. Чтобы использовать модуль, нужно знать только его интерфейс. Чем более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Препроцессор языка Си. Директива `#include`

Препроцессором называется первая фаза работы компилятора. Инструкции препроцессора называются **директивами**. Они должны начинаться с символа `#`.

Директива `#include <имя_файла>` вставляет содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы `#include`. Поиск файла, если не указан полный путь, ведется в стандартных каталогах включаемых файлов. Вместо угловых скобок могут использоваться кавычки (`" "`) — в этом случае поиск файла ведется в каталоге, содержащем исходный файл, а затем уже в стандартных каталогах.

Препроцессор языка Си. Директива `#include`

Реальные задачи требуют создания **многофайловых проектов** с распределением решаемых задач по **модулям** (файлам).

Исходные тексты совокупности функций для решения какой-либо подзадачи, как правило, размещаются в отдельном модуле (файле). Такой файл называют исходным (***source file*** -расширение **.c** или **.cpp**).

Прототипы всех функций исходного файла выносят в отдельный **заголовочный файл** (***header file***). Для него принято расширение **.h** или **.hpp**.

Таким образом, заголовочный файл **xxx.h** содержит **интерфейс** для некоторого набора функций, а исходный файл **xxx.c** (**xxx.cpp**) содержит **реализацию** этого набора.

Если некоторая функция из указанного набора вызывается из какого-то другого исходного модуля, то вы обязаны включить в этот модуль заголовочный файл **xxx.h** с помощью директивы **`#include`**.

Негласное правило требует включения этого же заголовочного файла и в исходный файл **xxx.c** (**xxx.cpp**).

Препроцессор языка Си. Директива `#include`

Директива `#include` является простейшим средством обеспечения согласованности объявлений в различных файлах, она **включает в файлы информацию об интерфейсе функций** из заголовочных файлов.

В **заголовочных файлах** принято размещать:

- определения типов, задаваемых пользователем, констант;
- объявления (прототипы) функций;
- объявления внешних глобальных переменных (**`extern`**);
- комментарии.

В заголовочном файле *не должно* быть определений функций и данных. Эти правила не являются требованием языка, а отражают разумный способ использования директивы `#include`.

Обратим внимание, что при использовании директивы `#include` может возникнуть ***проблема повторного включения заголовочных файлов***.

Условная компиляция (директивы `#if`, `#elif`, `#else`)

Самим ходом препроцессирования можно управлять с помощью **условных директив** (`#if`). Они представляют собой средство для выборочного включения того или иного текста в программу в зависимости от значения условия, вычисляемого во время компиляции.

Вычисляется **константное целое выражение**, заданное в строке `#if`. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до `#endif`, или `#elif`, или `#else` (директива препроцессора `#elif` похожа на `else if`). Выражение `defined(имя)`, заданное внутри `#if` есть `1`, если *имя* было определено ранее, и `0` в противном случае.

Условная компиляция (директивы **#if**, **#elif**, **#else**)

Например, *застраховаться от повторного включения* заголовочного файла **hdr.h** можно следующим образом:

```
#if !defined(HDR) /* страж включения */  
#define HDR  
  
        /* здесь содержимое hdr.h */  
  
#endif
```

При первом включении файла **hdr.h** будет определено имя **HDR**, а при последующих включениях препроцессор обнаружит, что имя **HDR** уже определено, и перейдет сразу на **#endif**.

Директивы **#ifdef** и **#ifndef** специально предназначены для проверки того, определено или нет заданное в них имя. И пример, приведенный выше, можно записать и в таком виде:

```
#ifndef HDR /* страж включения */  
#define HDR  
  
        /* здесь содержимое hdr.h */  
  
#endif
```


Условная компиляция (директивы `#if`, `#elif`, `#else`)

Пример цепочки проверок имени **SYSTEM**, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Условная компиляция (директивы `#undef`)

Директива

`#undef имя`

удаляет определение символа. Используется редко.

Например, функции **`getchar`** и **`putchar`** реализованы с помощью макросов. Действие **`#define`** можно отменить с помощью **`#undef`**:

```
#undef getchar
```

...

```
int getchar(void) { ... }
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем.

Реализация

Рассмотрим на примере, пусть написана программа для работы с дробями:

```
#include <stdio.h>
struct drob {
    int ch, zn;
};
//-----
struct drob read_drob() { //чтение дроби с клавиатуры
    drob d;
    printf("введите числитель и знаменатель\n");
    scanf("%d%d", &d.ch, &d.zn);
    return d;
}
//-----
void write_drob (struct drob d) { //печать дроби на экран
    printf("числитель и знаменатель\n");
    printf("%d %d\n", d.ch, d.zn);
}
//-----
int compare(struct drob d1, struct drob d2) { //сравнение
    if (d1.ch*d2.zn==d1.zn*d2.ch)
        return 0;
    else if (d1.ch*d2.zn>d1.zn*d2.ch)
        return 1;
    else return -1;}
}
```

Реализация

```
//-----
struct drob cancat(struct drob d1, struct drob d2) {
    struct drob d;          //сложение двух дробей
    d.zn=d1.zn*d2.zn;
    d.ch=d1.ch*d2.zn+d1.zn*d2.ch;
    return d;
}
//-----
int main() {
    drob d1,d2;
    d1=read_drob();          //чтение дроби d1
    d2=read_drob();          //чтение дроби d2
    write_drob(d1);          //печать дроби d1 на экран
    write_drob(d2);          //печать дроби d2 на экран
    int x=compare(d1,d2);    //сравнение d1 и d2
    if (x==0) printf(" дроби равны");
    else if (x==1) printf("d1<d2");
    else printf("d1>d2");
    printf("\n результат суммы: \n");
    d1=cancat(d1, d2);       //сложение двух дробей
    write_drob(d1);
    return 0;
}
```

Реализация

Разделим нашу программу на несколько файлов:

- в первый вынесем функции ввода-вывода дроби и назовем **io_drob**;

- во-второй вынесем функцию сравнения и сложения двух дробей и назовем **drob_oper**;

- в третий вынесем содержимое функции **main()** и назовем **drob_work**.

Реализация

Содержимое файла: io_drob.h

```
#ifndef IO_DROB_H
#define IO_DROB_H

#include "drob_work.h"

struct drob read_drob();
void write_drob(struct drob d);

#endif
```

Содержимое файла: io_drob.cpp

```
#include "io_drob.h"
#include <stdio.h>

struct drob read_drob() {
    drob d;
    printf("введите числитель и знаменатель\n");
    scanf("%d%d", &d.ch, &d.zn);
    return d;
}

void write_drob (struct drob d) {
    printf("числитель и знаменатель\n");
    printf("%d %d\n", d.ch, d.zn);
}
```

Реализация

Содержимое файла: drob_oper.h

```
#ifndef DROB_OPER_H
#define DROB_OPER_H

#include "drob_work.h"

int compare(struct drob d1, struct drob d2);
struct drob concat(struct drob d1, struct drob d2);

#endif
```

Содержимое файла: drob_oper.cpp

```
#include "drob_oper.h"

int compare(struct drob d1, struct drob d2) {
    if (d1.ch*d2.zn==d1.zn*d2.ch)
        return 0;
    else if (d1.ch*d2.zn>d1.zn*d2.ch)
        return 1;
    else return -1;
}

struct drob concat(struct drob d1, struct drob d2) {
    struct drob d;
    d.zn=d1.zn*d2.zn;
    d.ch=d1.ch*d2.zn+d1.zn*d2.ch;
    return d;
}
```

Реализация

Содержимое файла: drob_work.h

```
#ifndef DROB_MOD_H
#define DROB_MOD_H
```

```
struct drob {
int ch, zn;
};
```

```
void work();
#endif
```

Содержимое файла: drob_work.cpp

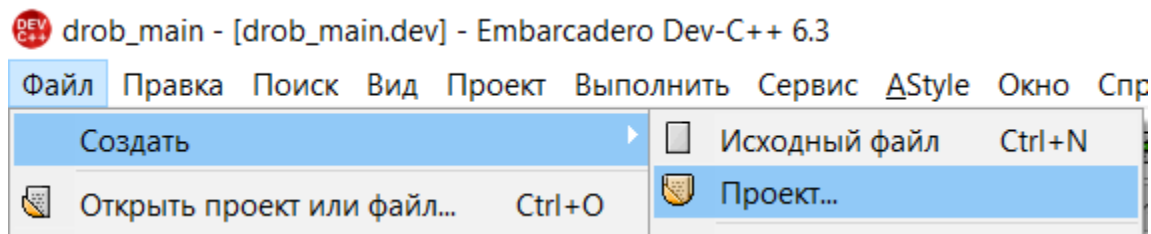
```
#include "io_drob.h"
#include "drob_oper.h"
#include <stdio.h>
```

```
void work() {
    drob d1,d2;
    d1=read_drob();
    d2=read_drob();
    write_drob(d1);
    write_drob(d2);
    int x=compare(d1,d2);
    if (x==0) printf(" дроби равны");
    else if (x==1) printf("d1<d2");
    else printf("d1>d2");

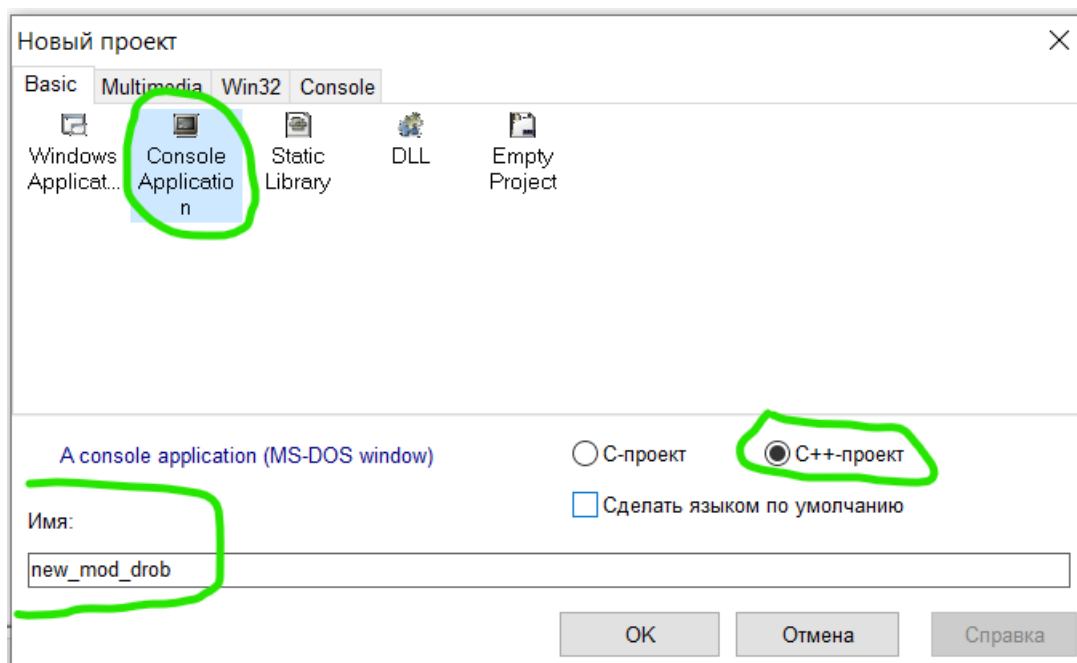
    printf("\n результат суммы: \n");
    d1=cancat(d1, d2);
    write_drob(d1);
    return;
}
```


Реализация

1 шаг: создание проекта



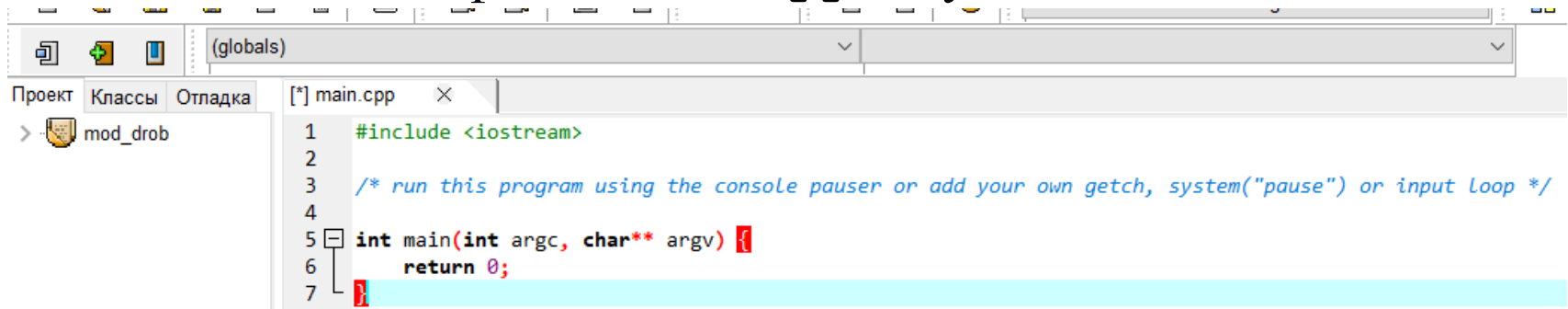
выбираем Console, C++ проект и вводим имя проекта:



после нажатия «ОК» выбираем место создания проекта (в проводнике создайте отдельную папку) и нажимаем «Сохранить», затем

Реализация

после этого создается файл **main.cpp** с пустым телом:

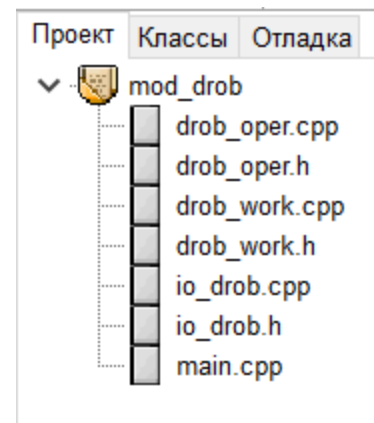
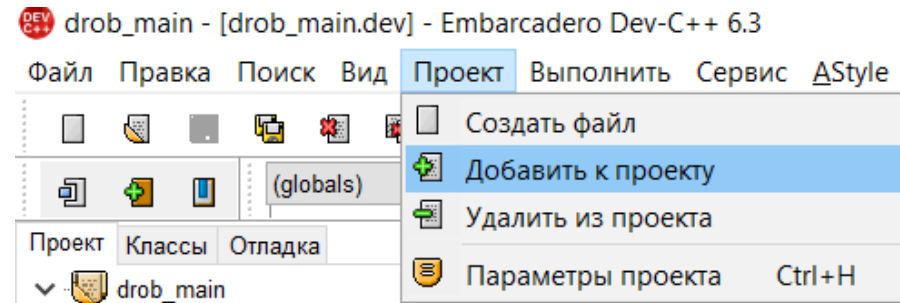


2 шаг: добавление в проект (в созданную папку скопируйте созданные файлы, выберите нужные файлы)

выбираем файлы из проводника:

- **drob_work.cpp**
- **drob_work.h**
- **io_drob.cpp**
- **io_drob.h**
- **drob_oper.cpp**
- **drob_oper.h**

после добавления:



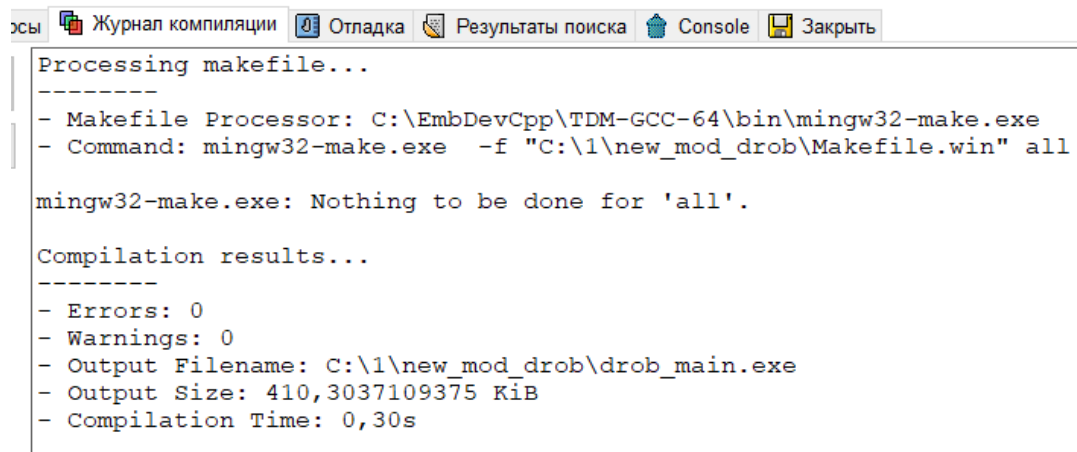
Реализация

3 шаг: заполнение `main.cpp` подключаем «`drob_work.h`» и вызываем функцию `work()`

```
#include "drob_work.h"
/* run this program using the console pauser or add your own
getch, system("pause") or input loop */

int main(int argc, char** argv) {
    work();
    return 0;
}
```

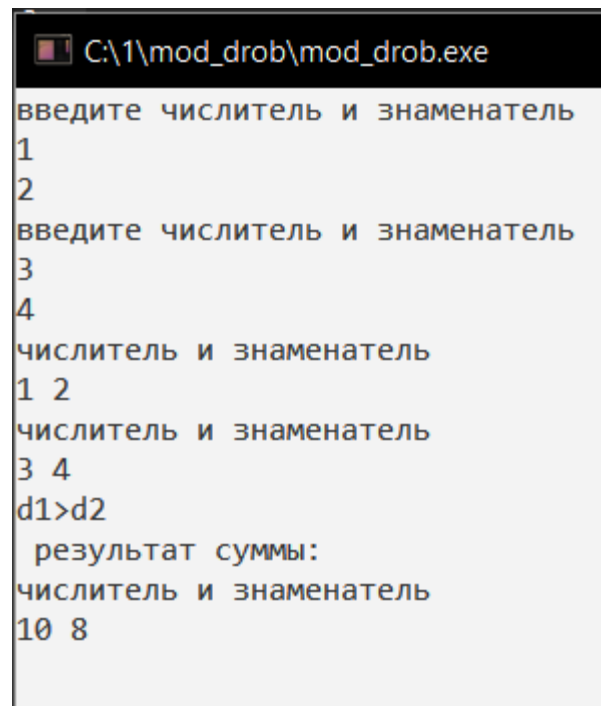
Компилируем и запускаем программу:



```
Processing makefile...
-----
- Makefile Processor: C:\EmbDevCpp\TDM-GCC-64\bin\mingw32-make.exe
- Command: mingw32-make.exe -f "C:\1\new_mod_drob\Makefile.win" all

mingw32-make.exe: Nothing to be done for 'all'.

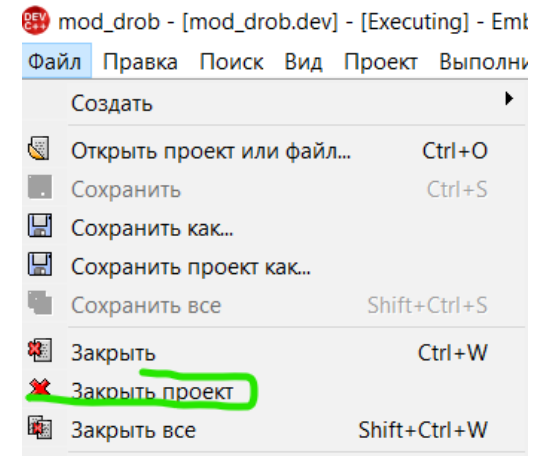
Compilation results...
-----
- Errors: 0
- Warnings: 0
- Output Filename: C:\1\new_mod_drob\drob_main.exe
- Output Size: 410,3037109375 KiB
- Compilation Time: 0,30s
```



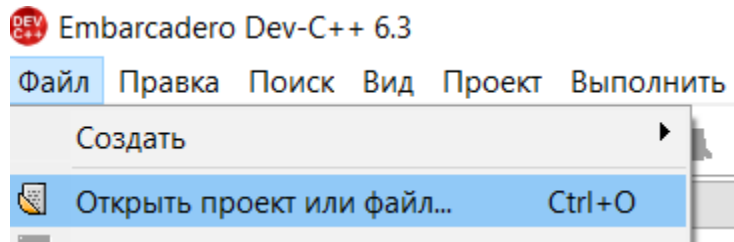
```
C:\1\mod_drob\mod_drob.exe
введите числитель и знаменатель
1
2
введите числитель и знаменатель
3
4
числитель и знаменатель
1 2
числитель и знаменатель
3 4
d1>d2
результат суммы:
числитель и знаменатель
10 8
```

Реализация

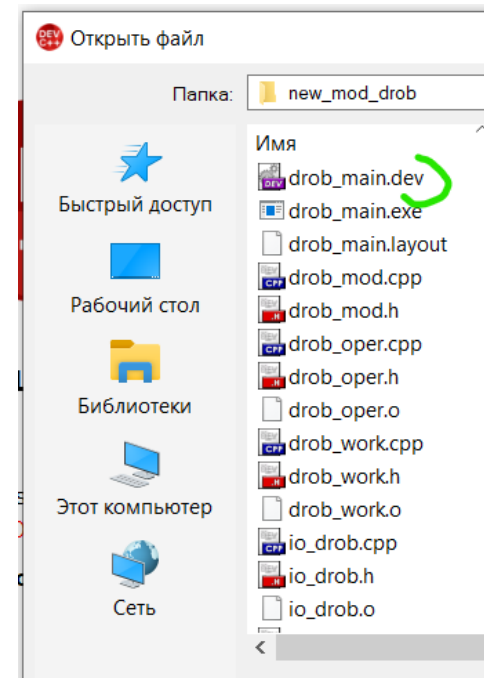
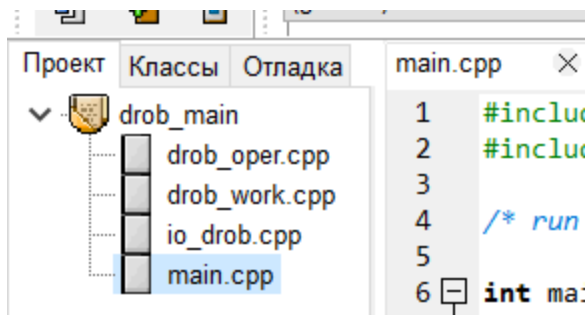
Для закрытия проекта выберите:



Для открытия проекта выберите в меню:



и выберите файл с расширением .dev
и нажмите на **main.cpp**



Структурный подход

Структурный подход к программированию охватывает все стадии разработки проекта:

- спецификацию,
- разработка внутренних структур данных,
- проектирование,
- программирование,
- тестирование.

Задачи, которые при этом ставятся, — уменьшение числа возможных ошибок за счет применения только допустимых структур, возможно более раннее обнаружение ошибок и упрощение процесса их исправления. Ключевыми идеями структурного подхода являются нисходящая разработка, структурное программирование и нисходящее тестирование. Приведенные ниже этапы создания программ рассчитаны на достаточно большие проекты, разрабатываемые коллективом программистов. Для программы небольшого объема каждый этап упрощается, но содержание и последовательность этапов не изменяются.

Программные проекты: как это часто бывает



Так объяснил
заказчик



Так понял менеджер
проекта



Так описал аналитик



Так реализовал
программист



Так презентовал
проект менеджер



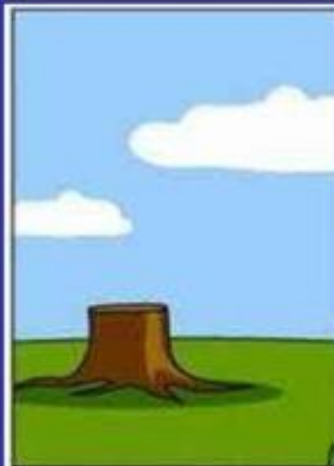
Такой оказалась
документация



Таким оказался
продукт



Такими оказались
затраты



Такой оказалась
работоспособность



Чего хотел заказчик
на самом деле ...

Структурный подход

I этап - спецификация.

Создание любой программы начинается с постановки задачи и завершается созданием технического задания, а затем внешней спецификации программы, включающей в себя:

- описание исходных данных и результатов (типы, форматы, точность, способ передачи, ограничения);
- описание задачи, реализуемой программой;
- способ обращения к программе;
- описание возможных аварийных ситуаций и ошибок пользователя.

Таким образом, программа рассматривается как черный ящик, для которого определена функция и входные и выходные данные.

Структурный подход

II этап – разработка внутренних структур данных.

Большинство алгоритмов зависит от того, каким образом организованы данные, поэтому начинать проектирование программы надо не с алгоритмов, а с разработки структур, необходимых для представления входных, выходных и промежуточных данных. При этом принимаются во внимание многие факторы, например, ограничения на размер данных, необходимая точность, требования к быстродействию программы. Структуры данных могут быть статическими или динамическими.

Структурный подход

III этап - проектирование (определение общей структуры и взаимодействия модулей).

На этом этапе применяется технология нисходящего проектирования программы: разбиение задачи на подзадачи меньшей сложности, которые можно рассматривать отдельно.

На этом же этапе решаются вопросы разбиения программы на модули, главный критерий – минимизация их взаимодействия. Одна задача может реализовываться с помощью нескольких модулей и наоборот, в одном модуле может решаться несколько задач. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы записывают в обобщенной форме – например, словесной, в виде обобщенных блок-схем или другими способами. На этапе проектирования следует учитывать возможность будущих модификаций программы и стремиться проектировать программу таким образом, чтобы вносить изменения было возможно проще.

Структурный подход

IV этап - структурное программирование.

Процесс программирования также организуется по принципу «сверху вниз»: вначале кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, при этом на месте еще не написанных модулей следующего уровня ставятся «заглушки» - временные программы. «Заглушки» в простейшем случае просто выдают сообщение о том, что им передано управление, а затем возвращают его в вызывающий модуль. В других случаях «заглушка» может выдавать значения, заданные заранее или вычисленные по упрощенному алгоритму. Таким образом, сначала создается логический скелет программы, который затем обростает плотью кода.

Рекомендации по записи алгоритмов на C++ (большинство из этих рекомендаций справедливы и для других языков высокого уровня) главные цели — читаемость и простота структуры программы в целом и любой из составляющих ее функций.

При программировании следует отделять интерфейс (функции, модуля, класса) от его реализации и ограничивать доступ к ненужной информации.

Структурный подход

V этап - нисходящее тестирование.

Этот этап записан последним, но это не значит, что тестирование не должно проводиться на предыдущих этапах. Проектирование и программирование обязательно должны сопровождаться написанием набора тестов — проверочных исходных данных и соответствующих им наборов эталонных реакций.

Необходимо различать процессы тестирования и отладки программы. **Тестирование** — процесс, посредством которого проверяется правильность программы. Тестирование носит позитивный характер, его цель — показать, что программа работает правильно и удовлетворяет всем проектным спецификациям.

Отладка — процесс исправления ошибок в программе, при этом цель исправить все ошибки не ставится. Исправляют ошибки, обнаруженные при тестировании.

При планировании следует учитывать, что процесс обнаружения ошибок подчиняется закону насыщения, то есть большинство ошибок обнаруживается на ранних стадиях тестирования, и чем меньше в программе осталось ошибок, тем дольше искать каждую из них. Для исчерпывающего тестирования программы необходимо проверить каждую из ветвей алгоритма.

Структурный подход

Общее число ветвей определяется комбинацией всех альтернатив на каждом этапе. Это конечное число, но оно может быть очень большим, поэтому программа разбивается на фрагменты, после исчерпывающего тестирования которых они рассматриваются как элементарные узлы более длинных ветвей. Кроме данных, обеспечивающих выполнение операторов в требуемой последовательности, тесты должны содержать проверку граничных . Отдельно проверяется реакция программы на ошибочные исходные данные.

Идея нисходящего тестирования предполагает, что к тестированию программы приступают еще до того, как завершено ее проектирование. Это позволяет раньше опробовать основные межмодульные интерфейсы, а также убедиться в том, что программа в основном удовлетворяет требованиям пользователя.