ФМИ № 2 2015: информатика № вёрстки 3	№ корректуры: Число ошибок: Верстал: *Соболева*	Дата	Подпись:

Хирьянов Тимофей Фёдорович
Преподаватель кафедры информатики МФТИ и портала Foxford.ru.

Функции в C++ и практика их написания

В современной практике программирования большое внимание уделяется объектно-ориентированному программированию, популярность имеют такие языки, как C++, Java, Ruby, Python. Я являюсь большим сторонником ООП, однако мой опыт преподавания информатики студентам МФТИ показывает, что независимо от знания языка программирования, владения продвинутыми алгоритмами или опыта в олимпиадном программировании, студенты не владеют базовой культурой написания программ – парадигмой структурного программирования. Они просто не знают, о чём надо думать во время написания программы, с какой позиции вообще начинать написание кода. В результате возникают программы, которые невозможно ни читать, ни исправлять, ни дописывать, а значит, невозможно использовать в реальной жизни.

Понятие о структурном программировании

Что же такое структурное программирование? Это – парадигма разработки программ с помощью представления их в виде иерархической структуры блоков. Идея структурного программирования появилась в 1970-х годах у учёного Эдсгера Вибе Дейкстры и была популяризована Никлаусом Виртом, создателем широко известного в школах языка Pascal.

В эту парадигму входит всего три пункта:

1. Любая программа состоит из трёх типов конструкций:
   - последовательное исполнение;
   - ветвление;
   - цикл.
2. Логически целостные фрагменты программы оформляются в виде *подпрограмм*. В тексте основной программы вставляется инструкция *вызова подпрограммы*. После окончания подпрограммы исполнение продолжается с инструкции, следующей за командой вызова подпрограммы.
3. Разработка программы ведётся поэтапно методом «сверху вниз».

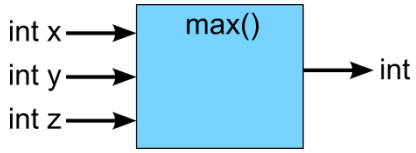
Первый пункт важен скорее не тем, что в нём есть, а тем, чего в

нём нет: в нём нет оператора безусловного перехода **goto**. Именно это отличает структурное программирование от *процедурного* (процедурное программирование – синоним *императивного*). Благодаря пункту два в языках высокого уровня появились новые синтаксические конструкции – функции и процедуры.

Пункт три – самый важный, и он является сутью парадигмы структурного программирования.

Функции в C++

Функция – важнейший элемент структурного программирования. Она является законченной подпрограммой, поэтому у неё есть свои «ввод» и «вывод» – параметры (аргументы) и возвращаемое значение.



С точки зрения внешней программы функция – это «чёрный ящик». Функция определяет собственную (локальную) область видимости, куда входят входные параметры, а также те переменные, которые объявляются непосредственно в теле самой функции.

Главное, что можно сделать с функцией – это возможность её вызвать.

Параметр функции – это принятый функцией аргумент, значение, переданное из вызывающего кода.

Обобщённо определение функции с параметрами выглядит так:

```
[спецификатор-класса-памяти] [спецификатор-типа] имя-функции
([список-формальных-параметров])
{ тело-функции }
```

где [список-формальных-параметров] – это перечисленные через запятую формальные параметры с указанием их типа.

Когда про параметр говорят с точки зрения описания функции, его называют формальным.

Различают:

1. Фактический параметр – значение, передаваемое в функцию при её вызове (при разных вызовах фактические параметры, очевидно, разные).

2. Формальный параметр – аргумент, указываемый при объявлении или определении функции (он имеет имя и тип).

Культура программирования требует, чтобы имя функции (и список её аргументов) были настолько понятными, чтобы для её вызова программисту-пользователю не обязательно было изучать документацию по ней, а тем более вникать в исходный код её реализации.

Например, на картинке выше описана функция, тела которой вы ещё не видели. Можете ли вы себе представить, что она делает? Как её вызвать? Что будет в переменной **a** после вызова **a = max(1, 2, 3)**?

И обратный пример: можно ли представить себе, что делает функция с именем **func(a, b, c)**?

Очевидно, слишком общие названия – всё равно, что их отсутствие.

Перед использованием функция должна быть *объявлена* и соответствующим образом *определена*.

Объявление (declaration) функции содержит список параметров вместе с указанием типа каждого параметра, а также тип возвращаемого функцией значения.

Определение (definition) функции содержит исполняемый код функции.

Вызов функции может встретиться в программе до определения, но обязательно после объявления – это требование компилятора. Чуть ниже вы увидите, что оно *противоречит пункту три структурного программирования*.

Функции, которые не возвращают значений, иногда называют *процедурами*.

Передача параметра по значению. Параметры в Си передаются по значению, то есть вызывающая функция *копирует* в доступную вызываемой функции память (на сегменте стека) непосредственное *фактическое значение*.

Это значит, что если внутри функции с копией переменной что-то случится, например, в неё будет записано новое значение, то с оригиналом переменной ничего не произойдёт, значение в нём останется прежним.

Передача параметра по адресу. Если необходимо именно изменить переменную из внешней, по отношению к вызываемой функции, области видимости, можно копировать **адрес переменной**, подлежащей изменению. Соответственно при вызове функции **f(&x)** приходится использовать операцию взятия адреса **&**.

Можно заметить, что передача параметра по адресу является частным случаем передачи по значению: передаваемым значением является *адрес*, пройдя по которому можно найти и изменить значение переменной **x**.

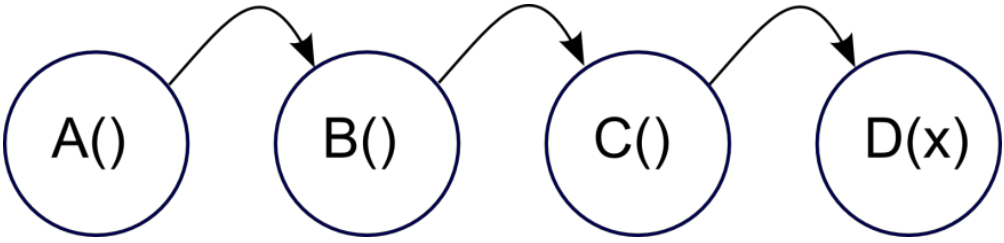
Передача параметра по ссылке. В C++ появилась возможность передачи параметра по ссылке. Фактически это передача по адресу, но скрытая от глаз программиста. Иногда это бывает удобно, однако возврат значений через параметры функции – порочная практика.

Как написать хорошую функцию. Стремитесь написать функцию так, чтобы:

1) у вас получилось **назвать её понятным именем**, отражающим её сущность;
2) она **выглядела законченно**;
3) она была **не длиннее 20 строк**;
4) для работы ей были нужны **только значения входных параметров** и никакие другие данные;
5) **результат возвращался через значение функции**, а не через глобальные переменные или параметры, переданные по адресу.

И ещё скажем про концентрацию внимания. При написании функции нужно *забыть обо всей остальной программе*, а думать только о ней, о её подзадаче. А после написания функции – *забыть о содержимом тела функции* и пользоваться ею так же, как и обычной библиотечной. Тогда ваш мозг будет свободней, и писать программу в целом станет легче.

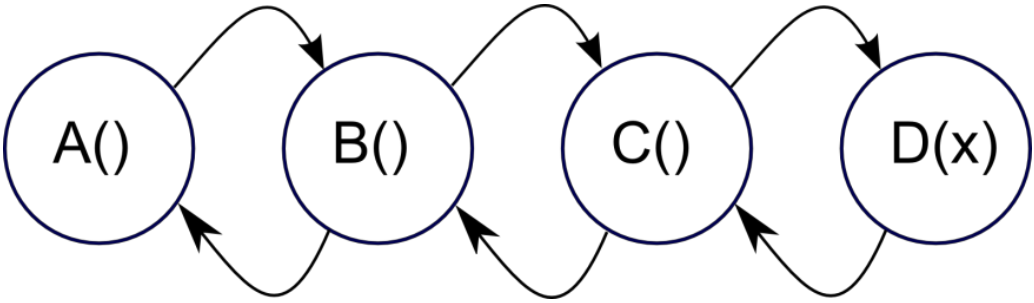
Граф вызовов функций. Для понимания связи функций друг с другом отобразим их связь друг с другом по логике их вызовов в виде *графа вызовов*.



По данному графу видно, что функция A() непосредственно вызывает функцию B(), которая, в свою очередь, вызывает функцию C(), а та пользуется функцией D().

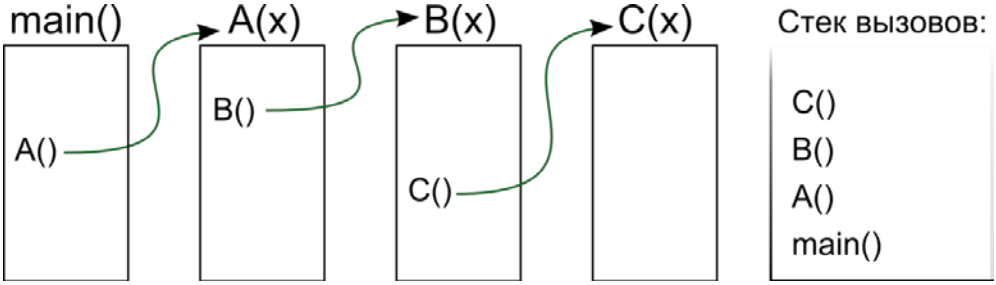
После вызова функцией A() функции B() сама она погружается в ожидание, не делая ничего до завершения подпрограммы B(). Функции A() важен результат B(), и она не может продолжить своё выполнение без него.

Если мы отобразим на графе обратный ход, как возвращается исполнение к функциям вместе с результатами вычислений с более глубоких уровней вызова, то получится вот так:



Стек вызовов. Заметим, что в одну и ту же вершину графа можно было попасть по-разному. Для того чтобы после выполнения вызванной функции вернуться в соответствующее моменту место вызова, важно запомнить, как именно мы в неё попали.

Стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу или подпрограмму (при вложенных или рекурсивных вызовах) называют **стеком вызовов** (call stack).



При вызове подпрограммы в стек заносится **адрес возврата** – адрес в памяти следующей инструкции приостанавливаемой программы, а управление передаётся подпрограмме. При возврате из подпрограммы адрес возврата снимается со стека, и управление передаётся на следующую инструкцию приостановленной программы (или подпрограммы).

Если подпрограмма сама вызывает подпрограмму, то в стек заносится очередной адрес возврата. Так продолжается до тех пор, пока очередная функция не отработает сама, ничего уже не вызывая. Стек заполняется снизу вверх, а освобождается сверху вниз, поэтому возвраты из функций будут идти ровно по той же траектории, как шли вызовы, только в обратном порядке.

Заметим также, что для анализа текущего состояния программы (например, при произошедшей в программе ошибке) граф вызовов малопригоден, а вот стек вызовов крайне полезен.

Проектирование «сверху-вниз»

Техника написания функций и понимание логики их работы очень важны. Однако самая суть структурного программирования – в подходе к проектированию программы. Для понимания этого приведём процесс решения конкретной задачи во времени.

Условие задачи. Допустим (в рамках задачи), что в некоторой стране зарплата сотрудника дорожной патрульной службы состоит из штрафов, накладываемых на водителей за превышение скорости в 60 км/ч, штраф напрямую зависит от номера автомобиля, а рабочий день заканчивается с приездом начальника, при этом начальника штрафовать нельзя. Требуется посчитать зарплату сотрудника за день.

Входные данные подаются построчно, в каждой строке – скорость (целое число) и номер автомобиля (6 символов – 1 буква, 3 цифры и ещё 2 буквы).

Штраф для автомобильных номеров зависит от количества совпадающих цифр: три совпадают – 1000 у. е., две любые цифры совпадают – 500 у. е., все цифры разные – 100 у. е.

Решение №1.

Помечтаем о том, что всё уже написано до нас:

```
   int main()
{
        int zarplata = poschitat_zarplatu_sotrudnika();
        cout << zarplata << endl;
        return EXIT_SUCCESS;
   }
```

Решение №2.

Как жаль, что мечта не компилируется… Так исправим это!

```
   #include <iostream>
   using namespace std;
   /*
   * Функция считает и возвращает зарплату сотрудника ДПС,
   * считывая исходные данные с клавиатуры.
   */
   int poschitat_zarplatu_sotrudnika();

   int main()
{
        int zarplata = poschitat_zarplatu_sotrudnika();
        cout << zarplata << endl;
        return EXIT_SUCCESS;
   }
   int poschitat_zarplatu_sotrudnika()
   {
        return 0; //FIXME! Но пока она этого не делает...
   }
```

Успешная компиляция, commit на github, но у нас возникла «недоделка» – FIXME…

Заметим, что такая функция, которая позволяет коду скомпилироваться, но пока ничего не выполняет, называется *заглушкой* и широко используется в практике программирования (см. технику программирования TDD).

Решение №3.

Что же, время заняться детализацией функции `poschitat_zarplatu_sotrudnika()`.

Размечтаемся, что у нас уже написаны всё необходимые функции, и напишем её так:

```
   int poschitat_zarplatu_sotrudnika();
   {
    int summa_shtrafov = 0;
    int skorost_avtomobilya;
    string nomer_avtomobilya;
    cin >> skorost_avtomobilya >> nomer_avtomobilya;
    while (!detect_nachalnik(nomer_avtomobilya))
    {
        if (skorost_avtomobilya > 60) {
            summa_shtrafov +=
poschitat_shtraf(nomer_avtomobilya);
        }
        cin >> skorost_avtomobilya >> nomer_avtomobilya;
    }
    return summa_shtrafov;
}
```

Заметим, что в этом решении делегировано в отдельные функции всё, что только можно было делегировать. Разве что проверка превышения скорости осталась без собственной функции, но она уж слишком тривиальна, чтобы вокруг неё разводить сантименты.

Заметим, что в групповой разработке ПО старший программист на этом завершил бы свою работу. Ему осталось только сформулировать для каждой из функций её прототип (заголовок) и документирующий комментарий, раздать работу по написанию мелких функций младшим программистам и идти пить чай.

На данном этапе пропустим написание функций-заглушек и сразу приведём окончательное решение.

Окончательное решение задачи

```
   #include <iostream>
   using namespace std;
   /*
   * Функция считает и возвращает зарплату сотрудника ДПС,
   * считывая исходные данные с клавиатуры.
   */
   int poschitat_zarplatu_sotrudnika();

   /*
   * Проверяет, принадлежит ли данный номер начальнику.
   */
   bool detect_nachalnik(string nomer_avtomobilya);
```

```
   /*
   * Вычисляет штраф для нарушителя по его номеру.
   */
int poschitat_shtraf(string nomer_avtomobilya);

   /*
   * Проверяет, является ли номер «крутым» (совпадение трёх цифр)
   */
bool krutoj_nomer(string nomer_avtomobilya);

   /*
   * Проверяет, является ли номер «красивым» (совпадение двух цифр)
   */
bool krasivyj_nomer(string nomer_avtomobilya);

   int main()
{
       int zarplata = poschitat_zarplatu_sotrudnika();
       cout << zarplata << endl;
       return EXIT_SUCCESS;
   }
   /*-— определение тел функций ----*/
int poschitat_zarplatu_sotrudnika()
   {
    int summa_shtrafov = 0;
    int skorost_avtomobilya;
    string nomer_avtomobilya;
    cin >> skorost_avtomobilya >> nomer_avtomobilya;
    while (!detect_nachalnik(nomer_avtomobilya))
    {
        if (skorost_avtomobilya > 60) {
            summa_shtrafov += poschitat_shtraf(nomer_avtomobilya);
        }
        cin >> skorost_avtomobilya >> nomer_avtomobilya;
    }
    return summa_shtrafov;
}

bool detect_nachalnik(string nomer_avtomobilya)
{
    return nomer_avtomobilya == "A999AA";
}

int poschitat_shtraf(string nomer_avtomobilya)
{
    if (krutoj_nomer(nomer_avtomobilya)) return 1000;
    else if (krasivyj_nomer(nomer_avtomobilya)) return 500;
    else return 100;
}

bool krutoj_nomer(string nomer_avtomobilya)
{
    if (nomer_avtomobilya[1] == nomer_avtomobilya[2]
```

```
        && nomer_avtomobilya[1] == nomer_avtomobilya[3]) {
        return true;
    }
    return false;
}

bool krasivyj_nomer(string nomer_avtomobilya)
{
    if (nomer_avtomobilya[1] == nomer_avtomobilya[2]
        || nomer_avtomobilya[2] == nomer_avtomobilya[3]
        || nomer_avtomobilya[1] == nomer_avtomobilya[3]) {
        return true;
    }
    return false;
}
```

Выводы

Вернёмся к требованию компилятора «перед использованием функция должна быть объявлена и соответствующим образом определена» и заметим, что мы поступали ровно наоборот – каждый раз сначала использовали функцию, а уже затем формулировали её прототип, и только в самом конце очередного шага детализации определяли её тело.

Преимущества структурного программирования:

- Выделение логически завершённого кода в отдельную структурную единицу сильно упрощает жизнь как писателя, так и читателя кода.
- Синтаксически выделенные подпрограммы имеют логичные имена, которые делают код самодокументирующимся.

Остаётся только заметить, что умение выделять объекты в ООП сродни умению формулировать функции, поэтому умение программировать в парадигме объектно-ориентированного программирования является прямым продолжением умения программировать структурно.

Дорогие ребята, чтобы стать хорошими программистами, начинайте с самого начала писать исходный код красиво и понятно. Успехов вам в изучении программирования!