Язык программирования Си

Бикулов Д.А., Иваницкая Н.В., Иванов А.П.

Семинар 4. Функции, передача параметров по значению. Рекурсия. Глобальные и статические переменные, правила видимости переменных. Модульный подход в программировании и раздельная компиляция.

1 Функции, передача параметров по значению

Функция – самостоятельная единица программы, спроектированная для решения конкретной задачи. Функции повышают уровень модульности программы, облегчают ее чтение, внесение изменений и исправление ошибок.

Определение любой функции имеет следующий вид:

```
<тип_результата> <имя_функции>(<тип1 аргумент1> [, другие аргументы...] )
{
  // тело функции
  <выполняемые операторы>;
}
```

Первая строка, в которой описывается имя функции, тип возвращаемого значения и принимаемые функцией аргументы называется *заголовком функции*.

Блок в фигурных скобках, который следует за заголовком функции и описывает то, что именно функция делает – называется *телом функции*.

Пример. Создадим функцию, вычисляющую модуль числа, а в **`main()`** устроим проверку ее работы:

```
#include <iostream>
using namespace std;

int iabs(int x)   //функция описывается до своего вызова
//  Здесь x – формальный параметр, его значение неизвестно,
//  известен только тип этого аргумента.
{
  if( x < 0 )
    return -x;
  return x;
}

void main()
{
  int a=10, b=0, c= -2;
  int d, e, f;
  d=iabs(a); //вызываем функцию iabs для фактического входного параметра a
  e=iabs(b);
  f=iabs(c);
  cout << d << e << f << endl << flush;
}
```

Функцию можно вызывать только после того, как она объявлена, то есть, заголовок и тело функции в программе должны идти перед ее первым вызовом из другой функции. Одну функцию внутри другой определять нельзя – все функции глобальны в программе.

Оператор **`return`** завершает выполнение функции и передает управление следующему оператору в вызывающей функции, а его аргумент – переменная, которая будет возвращена в качестве результата работы функции.

`main()` – точно такая же функция, как все остальные, ее отличие лишь в том, что при старте программы она вызывается самой первой и определяет – что дальше будет делать программа.

Обратите внимание, в примере выше функция **`main()`** описана как возвращающая тип **`void`** («ничего»), то есть, она не возвращает никакого значения в данном примере. Подобным же образом может быть описана любая функция, если не требуется возврата вычисленного значения (например, функция только распечатывает что-либо на экране).

В этом случае оператор **`return`** указывается без аргумента, причем, если он самый последний в теле функции – то его можно не указывать вовсе:

```
void print10( int A[10] )
{
  for( int i = 0; i < 10; i++ )
    cout << "[i]: " << A[i] << endl;
  cout << flush;
}
```

1.1 Локальные переменные

Все переменные, объявленные в функции являются ее внутренними переменными и вызывающая функция о них ничего не знает, следовательно – никак не может с ними работать (ни прочитать значение, ни изменить). Аналогично переменные вызывающей функции не известны вызываемой функции – все объявленные в функциях переменные *локальны*, то есть с ними можно работать только в пределах данной функции – начиная от места их объявления.

1.2 Передача параметров по значению

Более того, аргументы функции, объявленные в ее заголовке (формальные параметры) – это такие же локальные переменные данной функции, вызывающая функция их инициализирует (присваивает им начальное значение), но дальше никак на них повлиять не может:

```
#include <iostream>
using namespace std;

int sum(int x, int y)
{
  int a = y + 3;
  x += a;
  return x;
}

void main()
{
  int b = 1, c = 2, d;
  d = sum(b,c);
  cout << b << " " << c << " " << d << endl << flush;
}
// Будет напечатано: 1 2 3
```

Поэтому такой способ передачи параметров называется «*передача по значению*», что подчеркивает копирование значений при вызове функции.

1.3 Передача массивов в функции

Массивы в функции тоже можно передавать в качестве аргументов, однако тут есть очень существенная особенность: для достижения большей эффективности работы массив в языке Си определяется как машинный адрес начального элемента. Следовательно, если в функцию передать массив (адрес начального элемента) – то любые изменения элементов этого массива в вызываемой функции отразятся также и на вызывающей стороне:

```
#include <iostream>
using namespace std;

void print_array(int size, const int A[])
{
  for( int i = 0; i < size; i++ )
    cout << "[i]: " << A[i] << endl;
  cout << flush;
}

void cumulative(int size, int A[])
{
  for( int i = 1; i < size; i++ )
    A[i] += A[i-1];
}

void main()
{
  int X[5] = { 10, 20, 30, 40, 50 };
  cumulative(5,X);
  print_array(5,X);
}
// Будет напечатано в столбик: 10 30 60 100 150
```

Единственный способ для функции гарантировать вызывающей стороне, что переданный массив в ходе ее выполнения не будет изменен – это объявить массив *константным*, как это сделано в функции **`print_array()`**.

2 Рекурсия

Язык Си разрешает рекурсивные вызовы функций, то есть функция может вызвать сама себя:

```
#include <iostream>
#include <math.h>
using namespace std;

double factorial(int n)
{
  if ( n <= 0 ) return 1.0;
  return n* factorial(n-1);
}

void main()
{
  int n = 0;
  cout << "Input n: " << flush;
  cin >> n;
  cout << endl << sqrt(factorial(n)) << endl << flush;
}
```

3 Глобальные и статические переменные, правила видимости переменных

3.1 Глобальные переменные

Иногда бывает нужно создать глобальную переменную, то есть такую переменную, которая существует не только в пределах той или иной функции, но доступна в пределах всей программы (то есть нужно, чтобы любая функция программы могла с ней работать).

Такие переменные объявляются вне тела любой функции и инициализируются до начала работы самой первой функции (т.е. функции **`main()`**). Обычные локальные переменные инициализируются независимо и одинаково при каждом вызове функции.

Пример.

```
#include <iostream>
using namespace std;

int A[1000];
int size;

void fill_a(int k)
{
  for( int i = 0; i < k; i++ )
    A[i] = 3*k;
  size = k;
}

int proc_a()
{
  cumulative(size,A);
  cumulative(size,A);
  cumulative(size,A);
  cumulative(size,A);
  return A[size-1];
}

void main()
{
  int dim = 1;
  cout << "Input size: " << flush;
  cin >> dim;
  fill_a(dim);
  int result = proc_a();
  cout << endl << "Sum4 = " << result << endl << flush;
  print_array(size,A);
}
```

В этом примере первая функция заполняет глобальный массив указанным количеством элементов и запоминает размер заполненной части массива в другой глобальной переменной. А вторая функция обрабатывает заполненную часть массива при помощи функции, определенной в одном из предыдущих примеров.

3.2 Статические переменные

Статические локальные переменные не уничтожаются между вызовами функции, инициализируются они только при первом вызове, затем их измененное значение сохраняется до следующего вызова. То есть, ведут они себя почти так же, как и глобальные переменные, но работает с ними только одна единственная функция, та самая, в которой они объявлены.

Пример.

```
#include <iostream>
using namespace std;

void trystat()
{
  int fade=1;
  static int stay=1;
  cout << "fade=" << fade << " & stay=" << stay << endl << flush;
  fade++; stay++;
}

void main()
{
  for( int count=1; count <= 3; count++ ) {
    cout << "Iteration " << count << endl << flush;
    trystat();
  }
}
```

Здесь функция **`trystat()`** увеличивает каждую переменную после печати ее значения. Но результаты для двух ее переменных будут различны:

```
Iteration 1:
fade=1 & stay=1
Iteration 2:
fade=1 & stay=2
Iteration 3:
fade=1 & stay=3
```

Статическая переменная **`stay`** «помнит», что ее значение было увеличено на 1, в то время как для переменной **`fade`** начальное значение устанавливается каждый раз заново: stay инициализируется только один раз при компиляции функции **`trystat()`.**

4 Модульный подход в программировании и раздельная компиляция

Как правило, программы на языке Си состоят из большого числа небольших функций. Рано или поздно становится неудобно помещать все функции в один файл – удобнее разбить связанные между собой функции по разным файлам исходных текстов. Эти файлы будут компилироваться раздельно, но потом будут объединены в одну программу на последнем этапе сборки проекта. При необходимости на этом последнем этапе сборки будут добавлены и функции стандартных системных библиотек языка Си.

Предположим, что компоненты программы имеют большие размеры, и мы хотим распределить их по нескольким файлам. Пускай функция **`main()`** находится в отдельном файле, и каждая функция также находится в отдельном файле. В этом случае возникнет ошибка компиляции – к моменту вызова какой-либо функции она еще не будет нигде определена.

Чтобы эта ошибка не возникала в файле с исходным текстом вызывающей функции достаточно поместить заголовок вызываемой функции без указания тела и закончить заголовок точкой с запятой:

```
#include <iostream>
using namespace std;

void trystat();

void main()
{
  for( int count=1; count <= 3; count++ ) {
    cout << "Iteration " << count << endl << flush;
    trystat();
  }
}
```

Однако, это неудобно: в каждом файле с вызывающими функциями дублировать все заголовки вызываемых функций. Поэтому обычно поступают так: выносят все заголовки функций в отдельный заголовочный файл (традиционно он имеет расширение **.h**), который затем включают во все исходные тексты программы при помощи оператора **#include**. В этот же заголовочный файл выносят определения констант и общие для всей программы операторы препроцессора.

Заголовочные файлы системных библиотек типа **math.h** ничем не отличаются от ваших собственных заголовочных файлов – в них точно также содержатся все общие для данной библиотеки объявления. Разница заключается только в том, что свои файлы вы указываете оператору **#include** в двойных кавычках, а системные заголовочные файлы указываются в угловых скобках.

Пример. Файл **main.cpp**:

```
#include "solve.h"

void main()
{
  double xmax = solvemax();
  cout << "The biggest root is: " << xmax << endl << flush;
}
```

Файл **solve.h**:

```
#ifndef _SOLVE_H_
#define _SOLVE_H_
#include <iostream.h>      //подключаем все нужные библиотеки
#include <math.h>
using namespace std;

double solvemax();
double equation( double x );

#endif //_SOLVE_H_
```

Обратите внимание – здесь блок операторов препроцессора применяется для того, чтобы все определения, которые есть в этом заголовочном файле реализовались бы во включающем файле строго один раз – вне зависимости от того, сколько раз этот заголовочный файл будет включен в него. Это довольно распространенный прием в языке Си.

Файл **equation.cpp**:

```
#include"solve.h"

double equation( double x )
{
  const double a = 1.0, b = -2.0;
  return a*x*x + b*x;
}
```

Файл **solver.cpp**:

```
#include"solve.h"

double solvemax()
{
  if ( fabs(equation(0.0)) > 1.0E-10 ) return 1.0E+10;
     // проверка допустимости применения данного метода

  double y = equation(1.0), z = equation(-1.0);
  double r = (z+y) / (z-y);
  if ( r < 0 ) return 0;
  return r;
}
```

Важно: заголовочные файлы нужно включать в проект Microsoft Visual Studio не в папку «Source files», а в папку «Header files», либо не включать в проект вовсе – они появятся в нужной папке автоматически после первой же компиляции программы.

4.1 Внешние переменные

Для того, чтобы получить возможность воспользоваться глобальными переменными, которые определены за пределами данного программного модуля (файла с исходным текстом) нужно их объявить внешними глобальными переменными:

```
// main.cpp

#include <iostream>
using namespace std;

extern double a, b, c; // объявлены как внешние глобальные переменные
double f(double x);    // заголовок функции из другого модуля

void main()
{
  double x = 1.0;
  cout << "Input a, b, c, x: " << flush;
  cin >> a >> b >> c >> x;
  cout << endl << "f(x) = " << f(x) << endl << flush;
}
```

```
// func.cpp

double a = 1, b = 0, c = 0; // тут это обычные глобальные переменные

double f( double x )
{
  return a*x*x + b*x + c;
}
```

Если нет желания использовать объявленные глобальные переменные во всем файле **main.cpp** (например, в других функциях должны быть переменные с теми же именами) – то можно объявить внешние переменные локально внутри нужной функции:

```
void main()
{
  extern double a, b, c; // объявлены как внешние глобальные переменные,
                         // но только внутри данной функции
  double x = 1.0;
  cout << "Input a, b, c, x: " << flush;
  cin >> a >> b >> c >> x;
  cout << endl << "f(x) = " << f(x) << endl << flush;
}
```

4.2 Статические глобальные переменные

Наконец, бывает необходимость «закрыть» глобальную переменную, объявленную в данном модуле от любого использования в других модулях одной и той же программы. Для этого используется такое же ключевое слово **static**, что и для объявления локальных статических переменных, но смысл его в данном контексте другой:

```
// file1.cpp

static double eps = 0.001;

double f( double x )
{
  if( x < eps ) return 0.0;
  return 1 / x;
}

...
```

```
// file2.cpp

static double eps = 0.00001;

double f( double x )
{
  if( x < eps ) return 0.0;
  return 1 / x / x;
}

...
```

В каждом из этих двух файлов переменная **eps** – своя, она может меняться совершенно независимо и ни одна из функций одного файла никак не может получить доступ к переменной **eps** из другого файла. Попытка использовать оператор **extern** в данном случае также ни к чему не приведет – доступ будет получен только к своей собственной переменной.

5 Вычисление определенных интегралов

Для вычисления определенного интеграла $\int_a^b f(x)dx$ можно использовать один из следующих численных методов, различающихся по своей точности, достигаемой при одинаковых затратах вычислительных ресурсов.

5.1 Методы нижних, верхних и средних прямоугольников

Будем рассматривать определенный интеграл как площадь под графиком подынтегральной функции. Разобьем весь интервал интегрирования на **N** отрезков одинаковой длины. Тогда аппроксимацией для площади под графиком функции может служить сумма площадей прямоугольников, ширина которых равна длине одного отрезка разбиения, а высота – значению функции **f(x)** в данном месте графика функции. При этом метод нижних прямоугольников будет приближать значение интеграла снизу:

$I = \sum \Delta F_i,\ \Delta F_i = h \cdot \min(f(x_i), f(x_i+h)),$ где $h = x_{i+1} - x_i$

Метод верхних прямоугольников будет приближать значение интеграла сверху:

$I = \sum \Delta F_i,\ \Delta F_i = h \cdot \max(f(x_i), f(x_i+h)),$ где $h = x_{i+1} - x_i$

Нужно обратить внимание, что методы нижних и верхних прямоугольников не тождественны методам правых и левых прямоугольников, так как функции на различных участках могут и возрастать и убывать. Также нужно принимать во внимание то, что истинное значение интеграла будет лежать строго между результатами, посчитанными методами нижних и верхних прямоугольников, то есть, вычисляя два этих метода одновременно – можно оценить и точность получившегося решения. При этом затраты вычислительных ресурсов не сильно вырастут, если устранить повторные вычисления функции в одной и той же точке.

Метод средних прямоугольников будет давать результаты, средние между описанными выше двумя методами:

$I = \sum \Delta F_i,\ \Delta F_i = h \cdot f(x_i + h/2),$ где $h = x_{i+1} - x_i$

5.2 Метод трапеций

Метод трапеций – чуть точнее метода прямоугольников, он заключается в том, что вычисляется площадь трапеции, а не прямоугольника:

$I = \sum \Delta F_i,\ \Delta F_i = h \cdot (f(x_i) + f(x_i+h)) / 2,$ где $h = x_{i+1} - x_i$

Нужно обратить внимание, что этот метод не совпадает с методом средних прямоугольников, хотя и похож на него.

5.3 Метод парабол (метод Симпсона)

Большой выигрыш в точности вычислений при эквивалентной затрате вычислительных ресурсов дает метод Симпсона, при котором на каждом сегменте разбиения находятся коэффициенты параболы по трем точкам, находящимся на границах сегмента и в его центре, интеграл от квадратичной функции легко вычисляется аналитически, что дает в результате следующую формулу для метода парабол:

$I = \sum \Delta F_i,\ \Delta F_i = h \cdot (f(x_i) + 4 \cdot f(x_i+h/2) + f(x_i+h)) / 6,$ где $h = x_{i+1} - x_i$

Типовое задание: написать программу, вычисляющую заданный определенный интеграл двумя разными методами на интервале, границы которого вводит пользователь. Точность расчетов тоже вводится пользователем, а контролируется точность вычисления следующим образом:

- Выбирается некоторое начальное разбиение интервала на отрезки, например, количество отрезков указывает пользователь.
- После подсчета определенного интеграла одним из методов удваиваем количество отрезков, разбивающих интервал и повторяем расчет.

- Если разница между двумя значениями интеграла при двух разных разбиениях меньше заданной пользователем точности – считаем задачу решенной.

- Если же эта разница больше требуемой точности – еще раз удваиваем количество разбивающих интервал отрезков и повторяем весь расчет.

В проекте должно быть три файла: **`main.cpp`** (ввод-вывод параметров и результатов), **`f.cpp`** (подынтегральная функция), **`int1.cpp`** (интеграл, посчитанный первым методом), **`int2.cpp`** (интеграл, посчитанный вторым методом, либо - аналитически) и заголовочный файл **`integral.h`**, в котором описаны заголовки всех функций и который включается во все файлы проекта через оператор **`#include`**.

Желательно (но не обязательно) создать отдельную функцию, вычисляющую заданный интеграл аналитически.

Программа должна распечатать:

1. подынтегральную функцию, введенные значения точности, начальное разбиение и границы интервала интегрирования;
2. аналитическое значение интеграла (если вычислялось);

Для каждого из методов:

3. значение интеграла, полученное численным методом;
4. достигнутая точность;
5. конечное количество разбиений;
6. количество итераций (удвоений разбиения), которые понадобились для достижения требуемой точности.

1. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних прямоугольников и методом трапеций. $f(x) = \sin(x)\sin(2x)\sin(3x)$

2. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних и нижних прямоугольников. $f(x) = \frac{1}{\cos(x)(1+\cos(x))}$

3. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом средних прямоугольников. $f(x) = x\sin(2x)$

4. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом Симпсона. $f(x) = \sqrt{1-x^2}$

5. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом верхних прямоугольников. $f(x) = l\ln(x)$

6. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом верхних прямоугольников. $f(x) = (3+5x)^3$

7. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = \frac{x^2}{1-x}$

8. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних прямоугольников и методом средних прямоугольников. $f(x) = \frac{2}{x(7+3x)}$

9. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = \frac{1}{x^2(1-3x)}$

10. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних и нижних прямоугольников. $f(x) = \frac{x}{(1+x)^2}$

11. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом Симпсона. $f(x) = \sin^2(x)$

12. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом нижних прямоугольников прямоугольников и методом Симпсона. $f(x) = \frac{1}{\cos^2(x)}$

13. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = \sqrt{\frac{1+x}{1-x}}$

14. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних и нижних прямоугольников. $f(x) = \frac{1}{x^2\sqrt{x^2-1}}$

15. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом средних прямоугольников и методом Симпсона. $f(x) = \sqrt{1-x^2}$

16. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом нижних прямоугольников и методом трапеций. $f(x) = \frac{x}{\sqrt{1+x^2}}$

17. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = \frac{1}{1+\cos(x)}$

18. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = x\, e^x$

19. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних и нижних прямоугольников. $f(x) = e^{3x}\cos(x)$

20. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом верхних прямоугольников. $f(x) = \frac{1}{x\ln(x)}$

21. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом средних прямоугольников. $f(x) = x^2\ln(x)$

22. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом средних прямоугольников. $f(x) = \frac{e^x - e^{-x}}{2}$

23. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом Симпсона. $f(x) = f\ \frac{e^x - e^{-x}}{e^x + e^{-x}}$

24. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом трапеций и методом нижних прямоугольников. $f(x) = \sin\left(\frac{1}{x}\right)$

25. Вариант

Вычислить определенный интеграл $\int_a^b f(x)dx$ двумя методами: методом верхних и нижних прямоугольников. $f(x) = \frac{\sin(x)}{x}$