[14. Использование функций.](#_Toc515770227)

[14.1. Создание и использование простой функции.](#_Toc515770228)

[14.2. Прототипы функций.](#_Toc515770229)

[14.3. Вызов по значению и вызов по ссылке.](#_Toc515770230)

[14.4. Использование указателей для связи между функциями](#_Toc515770231)

[14.5. Рекурсия.](#_Toc515770232)

[14.5.1. Равноправность функций в языке Си.](#_Toc515770233)

[14.6. Параметры и аргументы функций](#_Toc515770234)

[14.6.1. Формальные и фактические параметры.](#_Toc515770235)

[14.6.2. Аргумент типа void.](#_Toc515770236)

[14.6.3. Символьные параметры.](#_Toc515770237)

[14.6.4. Целочисленные параметры.](#_Toc515770238)

[14.6.5. Параметры в формате чисел с плавающей точкой.](#_Toc515770239)

[14.6.6. Параметры в формате чисел двойной длины.](#_Toc515770240)

[14.6.7. Массивы в качестве параметров.](#_Toc515770241)

[14.6.8. Аргументы по умолчанию](#_Toc515770242)

[14.7. Возвращение значения функцией: оператор return.](#_Toc515770243)

[14.8. Типы функций.](#_Toc515770244)

[14.8.1. Функции типа void.](#_Toc515770245)

[14.8.2. Функции типа char.](#_Toc515770246)

[14.8.3. Функции типа int.](#_Toc515770247)

[14.8.4. Функции типа long.](#_Toc515770248)

[14.8.5. Функции типа float.](#_Toc515770249)

[14.8.6. Функции типа double.](#_Toc515770250)

[14.9. Важные возможности C++.](#_Toc515770251)

[14.9.1. Встраивание (inline).](#_Toc515770252)

[14.9.2. Перегрузка (overloading).](#_Toc515770253)

[14.9.3. Функции с переменным числом параметров. Многоточие (...).](#_Toc515770254)

[14.9.3.1. Задание числа дополнительных параметров с помощью первого параметра.](#_Toc515770255)

[14.9.3.2. Определение конца списка параметров с помощью параметра индикатора.](#_Toc515770256)

[14.9.3.3. Использование специального набора макроопределений.](#_Toc515770257)

[14.9.3.4. Список указателей переменной длины на char (конкатинация строк)](#_Toc515770258)

[14.9.3.5. Изменение параметров по числу и по типу](#_Toc515770259)

[14.9.3.6. Дополнительные примеры функций с произвольным числом параметров](#_Toc515770260)

[14.10. Аргументы функции main().](#_Toc515770261)

[14.10.1. Строки.](#_Toc515770262)

[14.10.2. Целые числа.](#_Toc515770263)

[14.10.3. Числа с плавающей точкой.](#_Toc515770264)

[14.11. Области видимости. Локальные и глобальные переменные](#_Toc515770265)

[14.12. Сложности в правилах области действия (scope rules).](#_Toc515770266)

[14.12.1. Неопределенные символы в программе на С.](#_Toc515770267)

[14.12.2. Использование переменной с файловой областью действия.](#_Toc515770268)

[14.12.3. Приоритет переменных с файловой и локальной областями действия.](#_Toc515770269)

[14.12.4. Проблемы области действия в C++.](#_Toc515770270)

[14.12.5. Операция уточнения области действия в C++.](#_Toc515770271)

[14.13. Математические функции](#_Toc515770272)

[14.14. Указатель на функцию](#_Toc515770273)

[14.14.1. Указатели на функции](#_Toc515770274)

[14.14.2. Указатели на методы](#_Toc515770275)

[14.14.3. Примеры](#_Toc515770276)

[14.14.3.1. Пример 1. Простой указатель на функцию](#_Toc515770277)

[14.14.3.2. Пример 2. Массив указателей на функции](#_Toc515770278)

[14.14.3.3. Пример 3. Указатель на функцию – параметр функции](#_Toc515770279)

[14.14.3.4. Пример 4. Сортировка массива](#_Toc515770280)

[14.14.4. Указатель на функцию и динамическое связывание](#_Toc515770281)

[14.14.5. Таблицы функций, вызов по имени](#_Toc515770282)

[14.14.6. Указатель на функцию как средство параметризации алгоритма](#_Toc515770283)

[14.14.7. Сортировка одной и той же структуры данных по разным критериям](#_Toc515770284)

[14.15. Массив указателей на функции](#_Toc515770285)

[14.16. Шаблоны функций в С++. Основные понятия](#_Toc515770286)

[14.17. Параметры шаблонов функций.](#_Toc515770287)

[14.18. Шаблоны функций. Аргументы по умолчанию](#_Toc515770288)

[14.19. Функции округления](#_Toc515770289)

[14.20. Компиляция программ, состоящих из двух или более функций.](#_Toc515770290)

1. Использование функций.

Принципы программирования на языке Си основаны на понятии функции. В представленных ранее примерах программирования мы уже воспользовались несколькими функциями: printf(), scanf(), getchar(), putchar () и strlen(). Эти функции являются системными, однако мы создали и несколько своих собственных функций под общим именем main(). Выполнение программы всегда начинается с команд, содержащихся в функции main(), затем последняя вызывает другие функции, например getchar(). Теперь мы переходим к вопросу о том, как создавать свои собственные функции и делать их доступными для функции main(), а также друг для друга.

Разработка программного обеспечения на практике является довольно непростым процессом. Программисту требуется учесть все тонкости и нюансы как всего программного комплекса в целом, так и отдельных его частей. Системный подход к программированию основывается на том, что поставленная перед разработчиком задача предварительно разбивается на пару-тройку менее крупных вопросов, которые, в свою очередь, делятся еще на несколько менее сложных задач, и так до тех пор, пока самые мелкие задачи не будут решены с помощью стандартных процедур. Таким образом, осуществляется так называемая функциональная декомпозиция.

Ключевым элементом данной модели для решения конкретной задачи в C++ выступает функция. Функцию можно представить как подпрограмму или некую процедуру, несущую законченную смысловую нагрузку.

Во-первых, что такое функция? Функция — самостоятельная единица программы, спроектированная для реализации конкретной задачи. Функции в языке Си играют ту же роль, какую играют функции, подпрограммы и процедуры в других языках, хотя детали их структуры могут быть разными. Вызов функций приводит к выполнению некоторых действий. Например, при обращении к функции printf() осуществляется вывод данных на экран. Другие же функции позволяют получать некоторую величину, используемую затем в программе. К примеру, функция strlen() «сообщает» программе длину конкретной строки. В общем, функции могут выполнять различные действия и получать значения величин, используемых в программе.

Почему мы пользуемся функциями? Во-первых, они избавляют нас от повторного программирования. Если конкретную задачу необходимо выполнить в программе несколько раз, мы напишем соответствующую функцию только один раз, а затем будем вызывать ее всегда, когда это требуется. Во-вторых, мы можем применять одну функцию, например putchar(), в различных программах. Даже в том случае, если некоторая задача выполняется только в одной программе, лучше оформить ее решение в виде функции, поскольку функции повышают уровень модульности программы и, следовательно, облегчают ее чтение, внесение изменений и коррекцию ошибок. Предположим, например, что мы хотим написать программу, которая делает следующее:

вводит набор чисел

сортирует эти числа

находит их среднее

выводит на печать гистограмму

Соответствующую программу можно записать так:

main()

{

float list [50];

readlist(list);

sort(list);

average(list);

bargraph(list);

}

Разумеется, мы должны были бы запрограммировать четыре функции readlist(), sort(), average() и bargraph(), но... это уже детали. Используя смысловые имена функции, мы четко определяем, что программа делает и как она организована. После этого можно заниматься каждой функцией отдельно и совершенствовать ее до тех пор, пока она не будет правильно выполнять требуемую задачу. Дополнительное преимущество указанного подхода заключается в том, что если мы создадим функции достаточно общего вида, то их можно будет использовать и в других программах.

Благодаря тому, что функции представляют собой самостоятельные, отдельно программируемые модули, программа выглядит как модульная структура. Модульное программирование позволяет разбивать программу на работоспособные блоки, которые в целом составляют законченную программу. Например, одна функция может использоваться для ввода данных, другая — для печати, третья — для записи данных на диск. В действительности, все операции в программах на С и C++ осуществляются в теле некоторой функции, которая называется main() и имеется во всех без исключения программах на С или C++.

Если вы работали с другими языками программирования, то заметите, что функции С очень похожи на программные модули других языков. Например, в Паскале применяются функции и процедуры, а в ФОРТРАНЕ только функции. Правильное проектирование функций в С и C++ в значительной степени определяет эффективность, читаемость и переносимость вашего программного кода.

Многие программисты предпочитают думать о функции, как о «черном ящике»; они задают ее через поступающую информацию (вход) и полученные результаты (выход). Все, что происходит внутри черного ящика, их не касается до тех пор, пока не нужно писать программу, реализующую эту функцию. Когда мы используем, например, функцию printf(), мы знаем, что должны передать ей управляющую строку и, возможно, несколько аргументов. Мы знаем также результат вызова функций printf(). He нужно полагать, что при программировании вам придется заниматься созданием функции printf(). Использование функций указанным выше способом позволяет сконцентрировать внимание на общей структуре программы, а не на деталях.

В этот материал включены многие программные примеры, назначение которых заключается в демонстрации способов создания и применения самых различных функций. Во многих примерах используются также встроенные библиотечные функции С и C++, расширяющие возможности ваших программ функций базируется на концепции прототипов функций, которая к моменту создания стандарта уже широко применялась в C++. В настоящий момент программирование на С переживает переходный период. Если вы читали журнальные статьи и книги, посвященные программированию на С, то наверно заметили, насколько различен в них стиль написания функций. Функции могут соответствовать, а могут и не соответствовать новому стандарту ANSI С, что зависит от того, пытаются ли сами программисты следовать этому стандарту. Компилятор Visual C/C++ использует для функций стандарт ANSI С, но может также компилировать и предыдущие версии. Программы на С здесь соответствуют стандарту ANSI С. Сделана попытка приблизить программы на C++ к стандарту ANSI С языка С, поскольку для C++ такого стандарта еще не существует.

Функции — это краеугольный камень в программировании на С и C++. Здесь излагается концепция функций и использование их прототипов согласно последнему стандарту ANSI С. Многочисленные примеры программ познакомят вас с различными типами функций и способами передачи параметров. Также вы узнаете правила применения стандартных переменных C/C++ argc и argv для передачи аргументов командной строки в функцию main(). Кроме этого, рассматриваются некоторые уникальные возможности, имеющиеся в C++.

## Создание и использование простой функции.

Что нам требуется знать о функциях? Нужно знать, как их можно определять, как к ним обращаться и как устанавливать связи между функцией и программой, ее вызывающей. Чтобы изучить это, мы далее рассмотрим очень простой пример, а затем будем обобщать его, вводя дополнительные характеристики до тех пор, пока не получим полную и ясную картину.

Наша первая скромная цель — создание функции, которая печатает 65 символов » в ряд. Чтобы эта функция выполнялась в некотором контексте, мы включили ее в программу, которая печатает простой титул фирменного бланка. Ниже приведена полная соответствующая программа. Она состоит из функций main() и starbar().

/\* титул фирменного бланка! \*/

#include<stdio.h>

#define NAME "MEGATHINK, INC."

#define ADDRESS "10 Megabuck Plaza"

#define PLACE "Megapolis, CA 94904"

void starbar();

void main()

{

starbar();

printf("%s\n", NAME);

printf("%s\n", ADDRESS);

printf("%s\n", PLACE);

starbar();

}

/\* далее следует функция starbar() \*/

#define LIMIT 65

void starbar()

{

int count;

for (count=1; count<=LIMIT; count++)

putchar('\*');

putchar('\n');

}

Результат работы программы выглядит так:



При рассмотрении этой программы необходимо обратить внимание на следующие моменты:

1. Мы вызвали функцию starbar() (или, можно сказать, обратились к ней) из функции main(), используя только ее имя. Это несколько напоминает заклинание, вызывающее злого духа, но, вместо того чтобы чертить пятиугольник, мы помещаем вслед за именем функции точку с запятой, создавая таким образом оператор:

starbar();

10_01

Это одна из форм вызова функции, но далеко не единственная. Когда в процессе выполнения программы компьютер достигает оператора starbar(), он находит указанную функцию, после чего начинает выполнять соответствующие ей команды. Затем управление возвращается следующей строке «вызывающей программы» — в данном случае main().

1. При написании функции starbar() мы следовали тем же правилам, что и при написании main(): вначале указывается имя, затем идет открывающая фигурная скобка, приводится описание используемых переменных, даются операторы, определяющие работу функции, и, наконец, закрывающая фигурная скобка. Мы даже поместили перед описанием функции starbar() директивы #define и #include, требующиеся для нее, а не для функции main().

10_02

1. Мы включили функции starbar() и main() в один файл. Вообще говоря, можно было создать два отдельных файла. Один файл несколько упрощает компиляцию, а два отдельных файла облегчают использование одной функции в разных программах. Случай двух и более файлов мы обсудим позже, а пока будем держать все наши функции в одном месте. Закрывающая фигурная скобка функции main() указывает компилятору на ее конец. Круглые скобки в имени starbar() говорят о том, что starbar() — это функция. Обратите внимание, что здесь за именем star-bar() не следует символ «точка с запятой»; его отсутствие служит указанием компилятору, что мы определяем функцию starbar(), а не используем ее.

Если рассматривать функцию starbar() как черный ящик, то ее выход — это напечатанная строка, состоящая из символов \*. Какие бы то ни было данные на входе у нее отсутствуют, потому что ей не нужно использовать информацию из вызывающей программы. Вообще, этой функции не требуется связь с вызывающей программой.

Обратимся к случаю, когда такая связь необходима.

## Прототипы функций.

Если вы еще не знакомы с тем, как писать функции на С, то у вас, возможно, возникнут разные вопросы. Как выглядит функция? Где она записывается в программе? Как объявляется? Из чего состоит? Где выполняется проверка типов?

Согласно стандарту ANSI С все функции должны иметь прототипы. Прототипы могут располагаться либо в самой программе на С или C++, либо в заголовочном файле. Большинство прототипов функций находятся в самих программах. Объявление функции в С и C++ начинается с ее прототипа. Прототип функции достаточно прост; обычно он включается в начало программы для того, чтобы сообщить компилятору тип и количество аргументов, используемых некоторой функцией. Использование прототипов обеспечивает более строгую проверку типов по сравнению с той, которая была при прежних стандартах С.

Хотя допустимы и другие стили записи прототипов функций, все же рекомендуется использовать, по возможности, следующий стиль: повторение строки объявления функции с добавлением в конце точки с запятой. Например:

возвращаемый\_тип имя\_функции(тип\_аргумевта(-ов)) (имя\_аргумента(-ов));

Функция может иметь тип void, int, float и так далее и определяется возвращаемым типом. Имя\_функции() — это любое значимое наименование, выбранное вами для определения этой функции. Если в функцию передается некоторая информация, то необходимо также задать тип\_аргумента и затем — имя\_аргумента. Аргументы также могут иметь тип void, int, float и так далее. Можно передавать функции несколько значений, повторяя тип и имя аргумента и отделяя их от остальных аргументов запятой. Допускается перечисление только типов аргументов, однако, такая форма прототипов используется нечасто.

Сама функция содержит в себе некий фрагмент программного кода на С или C++ и обычно следует за описанием функции main(). Функция может иметь следующий вид:

возвращаемый\_тип имя\_функции(типы\_аргументов и имена\_аргументов)

{

.

.

.

(объявления данных и тело функции)

.

.

return();

}

Обратите внимание на то, что первая строка самой функции идентична прототипу, который располагается в начале программы, за одним важным исключением: в конце отсутствует точка с запятой. В следующем примере на С показаны прототип функции и ее использование в программе:

/\*08PROTO.C

Программа на С, иллюстрирующая использование прототипа функции.

Функция складывает два целых числа и возвращает целое число\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

int iadder(int ix, int iy); /\* прототип функции \*/

main()

{

int ia=23;

int ib=13;

int ic;

ic=iadder(ia,ib);

printf("The summ is: %d\n",ic);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iadder(int ix, int iy) /\* объявление функции \*/

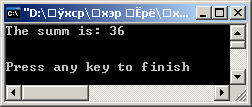
{

int iz;

iz=ix + iy;

return(iz); /\* возврат из функции \*/

}



Функция называется **iadder()**. В прототипе декларируется, что функция имеет два целочисленных аргумента и возвращает целочисленное значение. В действительности, стандарт ANSI С предполагает, что все прототипы функций располагаются в некотором отдельном заголовочном файле. Таким же образом, как вы можете догадаться, связаны заголовочные файлы и соответствующие библиотеки С. Как уже упоминалось, для простых программ допустимо включать прототип функции в тело самой программы.

Написание приведенной выше функции на C++ практически аналогично:

//08PROTO.CPP

//Программа на C++, иллюстрирующая использование прототипа функции.

//Функция складывает два целых числа и возвращает целое число

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

int iadder(int ix, int iy); // прототип функции

main()

{

int ia=23;

int ib=13;

int ic;

ic=iadder(ia,ib);

cout << "The summ is: " << ic << endl;

printf( "This is executed first.\n" );

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iadder(int ix, int iy) // объявление функции

{

int iz;

iz=ix+iy;

return(iz);

}



## Вызов по значению и вызов по ссылке.

В предыдущих примерах аргументы передаются функциям по значению. Когда переменные передаются по значению, в функцию передается копия текущего значения этой переменной. Поскольку передается копия переменной, сама эта переменная внутри вызывающей функции не изменяется. Вызов по значению — наиболее распространенный способ передачи информации (параметров) в функцию, и этот метод в С и C++ задан по умолчанию. Главным ограничением вызова по значению является то, что функция обычно возвращает только одно значение.

При вызове по ссылке в функцию передается не текущее значение, а адрес аргумента. Программе требуется меньше памяти, чем при вызове по значению. Кроме того, переменные в вызывающей функции могут быть изменены. Еще одним достоинством этого метода является то, что функция может возвращать более одного значения.

В результате выполнения операции & определяется адрес ячейки памяти, которая соответствует переменной. Если pooh — имя переменной, то &рooh — ее адрес. Можно представить себе адрес как ячейку памяти, но можно рассматривать его и как метку, которая используется компьютером для идентификации переменной. Предположим, мы имеем оператор

pooh = 24;

Пусть также адрес ячейки, где размещается переменная pooh, — 12126. Тогда в результате выполнения оператора

printf( %d %d\n" , pooh, &pooh);

получим

24 12126

Более того, машинный код, соответствующий первому оператору, словами можно выразить приблизительно так: «Поместить число 24 в ячейку с адресом 12126».

Воспользуемся указанной выше операцией для проверки того, в каких ячейках хранятся значения переменных, принадлежащих разным функциям, но имеющих одно и то же имя.

/\* контроль адресов \*/

#include<stdio.h>

void mikado(int);

void main()

{

int pooh = 2, bah = 5;

printf(" B main(), pooh = %d u &pooh = %u\n" , pooh, &pooh);

printf(" B main(), bah = %d u &bah = %u\n", bah, &bah);

mikado(pooh);

}

void mikado(int bah)

{

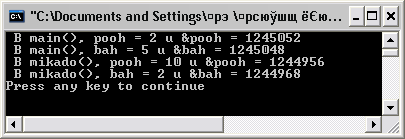
int pooh = 10;

printf(" B mikado(), pooh = %d u &pooh = %u\n", pooh, &pooh);

printf(" B mikado(), bah = %d u &bah = %u\n" , bah, &bah);

}

Мы воспользовались форматом %u (целое без знака) для вывода на печать адресов на тот случай, если их величины превысят максимально возможное значение числа типа int. В нашей вычислительной системе результат работы этой маленькой программы выглядит так:



О чем это говорит? Во-первых, две переменные pooh имеют различные адреса. То же самое верно и относительно переменных bah. Следовательно, как и было обещано, компьютер рассматривает их как четыре разные переменные. Во-вторых, при вызове mikado(pooh) величина фактического аргумента (pooh из main()) передается формальному аргументу (bah из mikado()). Обратите внимание, что было передано только значение переменной. Адреса двух переменных (pooh в main() и bah в mikado()) остаются различными.

Мы коснулись второго вопроса потому, что этот факт оказывается неверным для всех других языков. В той или иной процедуре Фортрана, например, можно использовать переменные вызывающей программы. Кроме того, в такой процедуре переменные могут иметь различные имена, но адреса их при этом будут совпадать. В языке Си подобные механизмы отсутствуют. Каждая функция использует свои собственные переменные. Это более предпочтительно, потому что «исходные» переменные не будут таинственным образом изменяться из-за того, что вызванная функция обладает побочным эффектом. Но это может также приводить и к некоторым трудностям, о чем и будет рассказано отдельно.

Иногда требуется, чтобы одна функция могла изменять переменные, относящиеся к другой. Например, в задачах сортировки часто бывает необходимо осуществлять обмен значениями между двумя переменными. Предположим, у нас есть две переменные х и у и мы хотим, чтобы они обменялись своими значениями. Простая последовательность операторов

х = у;

y = х;

не является решением поставленной задачи, потому что к тому моменту, когда начнет выполняться оператор во второй строке, первоначальное значение переменной X будет потеряно. Чтобы сохранить это первоначальное значение, необходимо дополнить данный фрагмент еще одной строкой:

temp = х;

х = у;

у = temp;

Теперь у нас есть требуемый метод; реализуем его в виде некоторой функции, а также создадим драйвер для ее проверки. Чтобы сделать более ясным, какая переменная принадлежит функции main(), а какая — функции interchange(), мы будем использовать переменные х и у в первой из них, и u и v — во второй.

/\* обмен1 \*/

#include<stdio.h>

void interchange(int , int);

void main()

{

int x=5, y=10;

printf(" At the beginning x = %d and y = %d.\n", x, y);

interchange(x, y);

printf(" And now x = %d and y = %d.\n", x, y);

}

void interchange(int u, int v)

{

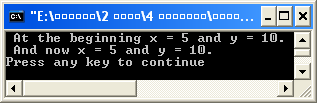
int temp;

temp = u;

u = v;

v = temp;

}



Попробуем выполнить эту программу. Результаты будут выглядеть следующим образом:

Вначале х = 5 и у = 10.

Теперь х = 5 и у = 10.

Не может быть! Значения переменных не поменялись местами! Вставим в программу interchange() несколько операторов печати, чтобы понять причину допущенной ошибки.

/\* обмен2 \*/

#include<stdio.h>

void interchange(int ,int );

void main()

{

int x = 5, y = 10;

printf(" At the beginning x =%d and y = %d.\n", x, y);

interchange(x,y);

printf(" And now x = %d and y = %d.\n", x, y);

}

void interchange(int u,int v)

{

int temp;

printf(" At the beginning u = %d and v = %d.\n" , u, v);

temp = u;

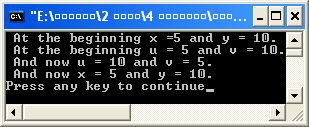
u = v;

v = temp;

printf(" And now u = %d and v = %d.\n" , u, v);

}

Результат работы этой программы выглядит так:



Отсюда видно, что ничего неправильного в работе функции interchange() нет; она осуществляет обмен значениями между переменными u и v. Проблема состоит в передаче результатов обратно в функцию main(). Как мы уже указывали, функции interchange() и main() используют различные переменные, поэтому обмен значениями между переменными u и v не оказывает никакого влияния на х и у! А нельзя ли каким-то образом воспользоваться оператором return? Мы могли бы, конечно, завершить тело функции interchange() строкой

return(u);

и изменить форму вызова в функции main() следующим образом:

х = interchange(x, у);

В результате такого обращения к функции переменная х получит новое значение, но у при этом не изменится.

С помощью оператора return в вызывающую программу можно передать только одну величину. Но нам нужно передать две величины. Это оказывается вполне осуществимым! Для этого нужно лишь воспользоваться «указателями».

В следующем примере используется функция iadder(). На этот раз переменные передаются в функцию по ссылке. Как показано ниже, в С для вызова по ссылке в качестве аргумента функции передается указатель. Этот же метод можно использовать и в C++. /\*

//08CBEF.C

//Программа на С, иллюстрирующая передачу параметров по ссылке

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

int iadder(int \*pix, int \*piy); //прототип функции

main()

{

int ia=23;

int ib=13;

int ic;

ic=iadder(&ia,&ib);

printf("The summ is: %d\n",ic);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iadder(int \*pix, int \*piy) //объявление функции

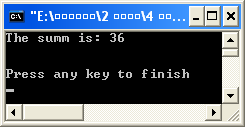
{

int iz;

iz=\*pix + \*piy;

return(iz);

}



Как вы уже знаете, в С в качестве аргументов при объявлении функций можно использовать переменные и указатели. В C++ кроме переменных и указателей имеется третий тип аргументов, называемый ссылочным типом (reference). Ссылочный тип задает адрес, однако не требует операции обращения по адресу. Во многих сложных программах на C++ такой синтаксис упрощает использование переменных-указателей внутри вызываемых подпрограмм. Внимательно изучите синтаксис следующего примера и сравните его с предыдущим:

// 08REFRNC.CPP

// Программа на C++, иллюстрирующая аналогичную передачу параметров

// по ссылке с использованием ссылочного типа C++

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

int iadder(int &rix, int &riy); //прототип функции

main()

{

int ia=23;

int ib=13;

int ic;

ic=iadder(ia,ib);

cout << "The summ is: " << ic << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iadder(int &rix, int &riy) //объявление функции

{

int iz;

iz=rix+riy;

return(iz);

}



Вы заметили отсутствие указателей при просмотре листинга программы на C++? К ссылочному типу данных в примере относятся rix и riy. В C++ запрещены ссылки на ссылки, ссылки на битовые поля и массивы ссылок, а также указатели на ссылки. Вне зависимости от того, используется ли вызов функции по ссылке или же ссылочный тип данных, в любом случае в C++ идет обращение к адресу аргумента.

## Использование указателей для связи между функциями

Мы только прикоснулись к обширному и увлекательному миру указателей. Сейчас нашей целью является использование указателей для решения задачи об установлении связи между функциями. Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных. Посмотрим, как она выглядит, выполним ее, а затем попытаемся понять, как она работает.

/\* обменЗ \*/

#include<stdio.h>

void interchange(int \* ,int \* );

void main()

{

int x = 5, y = 10;

printf(" At the beginning x = %d and y = %d.\n" , x, y);

interchange(&x,&y); /\* передача адресов функции \*/

printf(" And now x = %d and y = %d.\n" , x, y);

}

void interchange(int \*u, int \*v) /\* u и v являются указателями \*/

{

int temp;

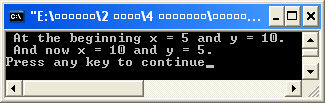
temp = \*u; /\* temp присваивается значение, на которое указывает u \*/

\*u = \*v;

\*v = temp;

}

После всех встретившихся трудностей, проверим, работает ли этот вариант!



Да, программа работает.

Посмотрим, как она работает. Во-первых, теперь вызов функции выглядит следующим образом:

interchange(&x, &y);

Вместо передачи значений х и у мы передаем их адреса. Это означает, что формальные аргументы и и v, имеющиеся в спецификации

interchange(u,v)

при обращении будут заменены адресами и, следовательно, они должны быть описаны как указатели. Поскольку х и у — целого типа, а u и v являются указателями на переменные целого типа, и мы вводим следующее описание:

int \*u, \*v;

Далее в теле функции оператор описания

int temp

используется с целью резервирования памяти. Мы хотим поместить значение переменной х в переменную temp, поэтому пишем

temp = \*u;

Вспомните, что значение переменной u — это &х, поэтому переменная и ссылается на х. Это означает, что операция \*u дает значение х, которое как раз нам и требуется. Мы не должны писать, например, так:

temp = u; /\* неправильно \*/

поскольку при этом происходит запоминание адреса переменной х, а не ее значения; мы же пытаемся осуществить обмен значениями, а не адресами.

Точно так же, желая присвоить переменной у значение переменной х, мы пользуемся оператором

\*u = \*v;

который соответствует оператору

х = у;

Подведем итоги. Нам требовалась функция, которая могла бы изменять значения переменных x и у. Путем передачи функции адресов переменных х и у мы предоставили ей возможность доступна к ним. Используя указатели и операцию \*, функция смогла извлечь величины, помещенные в соответствующие ячейки памяти, и поменять их местами.

Вообще говоря, при вызове функции информация о переменной может передаваться функции в двух видах. Если мы используем форму обращения

function1(х);

происходит передача значения переменной х. Если же мы используем форму обращения

function2(&x);

происходит передача адреса переменной х. Первая форма обращения требует, чтобы определение функции включало в себя формальный аргумент того же типа, что и х:

function1(num)

int num;

Вторая форма обращения требует, чтобы определение функции включало в себя формальный аргумент, являющийся указателем на объект соответствующего типа:

function2(ptr)

int \*ptr;

Пользуйтесь первой формой, если входное значение необходимо функции для некоторых вычислений или действий, и второй формой, если функция должна будет изменять значения переменных в вызывающей программе. Вторая форма вызова уже применялась при обращении к функции scanf(). Когда мы хотим ввести некоторое значение в переменную num, мы пишем scanf("%d", &num). Данная функция читает величину, затем, используя адрес, который ей дается, помещает эту величину в память.

Указатели позволяют обойти тот факт, что переменные функции interchange() являются локальными. Они дают возможность нашей функции «добраться» до функции main() и изменить величины описанных в ней объектов.

Программисты, работающие на языке Паскаль, могут заметить, что первая форма вызова аналогична обращению с параметром-значением, а вторая — с параметром-переменной. У программистов, пишущих на языке Бейсик, понимание всей этой методики может вызвать некоторые затруднения. В этом случае, если материал данного раздела покажется вам поначалу весьма необычным, не сомневайтесь, что благодаря некоторой практике, все обсуждаемые средства станут простыми, естественными и удобными.

## Рекурсия.

Как уже упоминалось ранее, функция может вызывать сама себя. При этом говорят, что возник рекурсивный вызов. Рекурсия бывает:

простой - если функция в теле содержит вызов самой себя;

косвенной – если функция вызывает другую функцию, а та в свою очередь вызывает первую.

При выполнении рекурсии программа сохраняет в стеке значения всех локальных переменных функции и ее аргументов с тем, чтобы в дальнейшем по возвращении из рекурсивного вызова восстановить их сохраненные значения. Рис. 0403 иллюстрирует поведение рекурсивной функции.



**Рис. 0403.** Рекурсивный вызов

В связи с вышеизложенным, применять рекурсию следует с осторожностью, так как ее использование для функции, содержащих количество переменных или слишком большое вызовов, может вызвать переполнение стека, также помнить, что при использовании рекурсивного вызова разработчик обязан предусмотреть механизм возврата в вызывающую процедуру, чтобы не произошло образования бесконечного цикла.

Некоторые задачи на практике могут быть проще и нагляднее решены именно с использованием рекурсивных функций. Например, решение тривиальной задачи нахождения факториала без обращения к рекурсии могло бы выглядеть следующим образом:

#include <iostream>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

int count = 1;

long int result =1;

while(count && count <31){

cout <<"Введите целое число: ";

cin >> count;

for(int i= count; i> 1; i--){

result \*= i;

}

cout << result << '\n';

result = 1;

}

getchar(); getchar();

return 0;

}

Ту же задачу можно решить более элегантно, применив рекурсию:

#include <iostream>

using namespace std;

long int fact(long);

int main()

{

setlocale(LC\_ALL, "Rus");

int count = 1;

while(count && count < 31){

cout << "Введите целое число: ";

cin >> count;

cout << fact(count) << '\n';

}

getchar(); getchar();

return 0;

}

long int fact(long x)

{

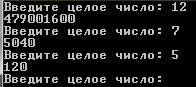
if (x== 0 || x== 1)

return 1;

return x \* fact(x-1);

}

Как видно, тело функции main () во втором примере максимально упростилось и занимается фактически вводом значения и выводом результата, в то время как все вычисления возложены на единственную содержательную строку в рекурсивной функции fact().



Рекурсия возникает в программах, в которых некоторая функция вызывает сама себя. Поначалу рекурсия кажется похожей на бесконечный цикл, однако, это не так. Рекурсия поддерживается как в С, так и в C++. Рекурсивные алгоритмы позволяют получить эффективное, читабельное и компактное решение задачи. Например: в следующей программе рекурсия используется для вычисления факториала числа. Факториал числа определяется как произведение самого числа и всех предыдущих целых чисел. Например:

8\*7\*6\*5\*4\*3\*2\*1= 40320

Необходимо внимательно выбирать тип данных, так как произведение растет очень быстро: факториал 15 равен 1307674368000.

/\*08FACTR.C

Программа на С, использующая рекурсивные вызовы функции.

Вычисляет факториал числа.

Пример: 7! =7x6x5x4x3x2x1= 5040\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

using namespace std;

double dfactorial(double danswer);

main()

{

double dnumber=15.0;

double dresult;

dresult=dfactorial(dnumber);

printf("The factorial of %.0lf is: %.0lf\n",dnumber,dresult);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

double dfactorial(double danswer)

{

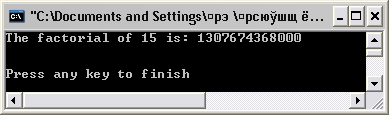
if(danswer <= 1.0)

return(1.0);

else

return(danswer\*dfactorial(danswer-1.0));

}



Возникает рекурсия, так как внутри функции dfactorial() имеется вызов ее самой. Обратите также внимание на то, что в функции printf() используется новый код форматирования для печати значения двойной длины: %...lf. Здесь "l" является модификатором формата "f и обозначает тип double вместо float.

### Равноправность функций в языке Си.

Все функции в программе, написанной на языке Си, равноправны: каждая из них может вызывать любую другую функцию и в свою очередь каждая может быть вызвана любой другой функцией. Это делает функции языка Си несколько отличными от процедур Паскаля, поскольку процедуры в Паскале могут быть вложены в другие процедуры (причем, процедуры, содержащиеся в одном гнезде, являются недоступными для процедур, расположенных в другом).

Нет ли у функции main() какой-то специфики? Безусловно, есть. Она заключается в том, что после «сборки» программы, состоящей из нескольких функций, ее выполнение начинается с первого оператора функции main(). Но этим ее исключительность и ограничивается. Даже функция main() может быть вызвана другими функциями, как показывает приведенный ниже пример:

/\* вызов функции main() \*/

#include <stdio.h>

main()

{

char ch;

printf (" Укажите произвольный символ. Q — признак конца работы.\n");

ch = getchar();

printf ("Так! Вы указали %c!\n", ch);

if (ch != 'Q')

more();

}

more()

{

main();

}

Функция main() вызывает more(), а функция more() вызывает main()! После вызова функции main() ее выполнение начинается с самого начала; мы организовали цикл с взаимным вызовом.

Функция может даже вызывать сама себя. Упростим предыдущий пример следующим образом:

/\* main.main \*/

#include <stdio.h>

main()

{

char ch;

printf (" Укажите произвольный символ. Q признак конца работы.\n");

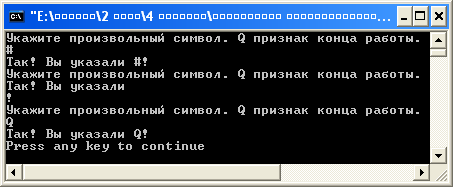
ch = getchar();

printf ("Так! Вы указали %c!\n", ch); if (ch != 'Q')

main();

}

Ниже приводятся результаты одного прогона программы, показывающие, что она работает. Обратите внимание на то, как обрабатывается символ «новая строка», который передается программе при нажатии клавиши [ввод].



Действие, состоящее в том, что функция вызывает сама себя, называется «рекурсией». Цикл, который мы создали, используя рекурсию, отличается от циклов while и do while. Когда функция main() вызывает сама себя, не происходит передачи управления на ее начало. Вместо этого в памяти машины создаются копии всего набора переменных функции main(). Если вы выведете на печать адреса переменных в обычном цикле, то увидите, что эти адреса не изменяются от итерации к итерации. Что же касается рассматриваемого здесь цикла, то в нем адрес используемой переменной меняется, поскольку при каждом выполнении тела цикла создается новая копия переменной ch . Если программа циклически выполняется 20 раз, то будет создано 20 различных копий переменной, каждая из. которых носит имя ch, но имеет свой собственный адрес.

## Параметры и аргументы функций

Каждая функция, которую предполагается использовать в программе, должна быть в ней объявлена. **Обычно объявления функций размещают в заголовочных файлах**, которые затем подключаются к исходному тексту программы с помощью директивы #include. Объявление функции описывает ее прототип (иногда говорят, сигнатура). Прототип функции объявляется следующим образом:

возвр\_тип FuncName(список объявляемых параметров);

Здесь возвр\_тип - возвращаемый функцией тип данных. Если возвращаемый тип данных не указан, то по умолчанию компилятор считает, что возвращаемый функцией тип есть int. Список объявляемых параметров задает тип и имя каждого параметра функции, разделенные запятыми. Допускается опускать имя параметра. Список объявляемых параметров функции может быть пустым. Приведем примеры прототипов функций:

int swap(int, int);

double max (double parl, double par2);

void func();

**Прототип функции может быть пропущен, если определение функции следует до первого ее вызова из любой другой функции**. Последний вариант считается плохим стилем программирования, так как бывают случаи перекрестных вызовов, то есть когда две или несколько функций вызывают друг друга, в результате чего невозможно вызвать одну функцию без предварительного определения другой.

Определение функции состоит из ее заголовка и собственно тела, заключенного в фигурные скобки и несущего смысловую нагрузку. Если функция возвращает значение, отличное от типа void, в теле функции обязательно должен присутствовать оператор return с параметром того же типа, что и возвращаемое значение. В случае если возвращаемое значение не будет использоваться в дальнейшем в программе (void), оператор return следует без параметра или вообще может быть опущен, тогда возврат из функции осуществляется по достижении закрывающейся скобки.

Для того чтобы функция выполнила определенные действия, она должна быть вызвана в программе. При обращении к функции она выполняет поставленную задачу, а по окончании работы возвращает в качестве результата некоторое значение.

Вызов функции представляет собой указание идентификатора функции (ее имени), за которым в круглых скобках следует список аргументов, разделенных запятыми:

имя\_функции ( аргумент\_1,

аргумент\_2,

аргумент\_3,

аргумент\_N);

Каждый аргумент функции представляет собой переменную, выражение или константу, передаваемые в тело функции для дальнейшего использования в вычислительном процессе. Список аргументов функции может быть пустым.

Функция может вызывать другие функции (одну или несколько), а те, в свою очередь, производить вызов третьих и т.д. Кроме того, функция может вызывать сама себя. Это явление в программировании называется рекурсией. Подробнее рекурсия будет рассмотрена ниже.

Как уже было отмечено ранее, любая программа на C++ обязательно включает в себя главную функцию main(). Именно с этой функции начинается выполнение программы.

На рис.0401 показан схематический порядок вызова функций.



**Рис. 0401.** Вызов функции

Программа начинает выполняться с функции main() до вызова функции FUNC1 (х, у). С этого момента управление программой передается в функцию FUNC1 (х, у), причем в качестве значения переменной Radius данная функция использует величину переменной х, а в качестве переменной Symbol передается значение у (рис. 0402 иллюстрирует передачу параметров в функции). Далее до оператора return выполняется тело функции FUNC1(х, у), после чего управление возвращается в тело функции main (), а именно, следующему за вызовом FUNC1 (х, у) оператору. После этого продолжается выполнение функции main() до вызова функции FUNC2(a, b, с) При вызове этой функции переменная а передает значение логической переменной YesNo, переменная b целочисленной переменной Count, а переменная с - короткому целому Key.



**Рис. 0402.** Передача параметров в функции

Функция main () часто не имеет аргументов, однако, если требуется при вызове программы передать ей какие-нибудь параметры, синтаксис функции main() меняется:

int main(argc, argv)

Здесь первый аргумент, argc, указывает количество передаваемых параметров, а второй, argv, является указателем на массив символьных строк, содержащих эти аргументы. Массивы и указатели будут рассмотрены позже.

Рассмотрим несколько примеров объявления, вызова и определения функций.

// Объявление функций:

int MyFunction(int Number, float Point);

char InputSymbol();

void SetBit(short Num);

void EmptySample(int, char);

// Вызов функций:

Result = MyFunction(varbl, 3.14);

symb = InputSymbol();

SetBit(3);

EmptySample (2, smb1);

//Определение функций:

int MyFunction (int Number, float Point)

{

int mу\_х;

…

return my\_x; // my x - типа int

}

char InputSymbol ()

{

char symbol;

cin >>symbol;

return symbol;

}

void SetBit (short number)

{

GlobalBit = GlobalBit | number;

}

void EmptySample (int x, char ch)

{

}

Для того чтобы было более понятно место функции в программе, детально рассмотрим пример вычисления квадрата числа с использованием функции.

В заголовочном файле header. h разместим прототип функции MySquare ():

// header.h

long MySquare(int);

Тогда главный модуль программы будет подключать заголовочный файл, содержать описание и вызов функции MySquare () из функции main().

**MyFile.cpp**

#include "header.h"

int main()

{

setlocale(LC\_ALL, "Rus");

int Variable = 5;

cout << MySquare(Variable);

getchar(); getchar();

return 0;

}

**header.h**

long MySquare(int x)

{

return x \* x;

}



Тот же пример может выглядеть несколько иначе, если вместо подключения заголовочного файла поместить прототип функции MySquare () прямо в файл исходного текста программы:

#include <iostream>

using namespace std;

// Прототип функции

long MySquare(int);

int main()

{

setlocale(LC\_ALL, "Rus");

int Variable = 5;

cout << MySquare(Variable);

getchar(); getchar();

return 0;

}

long MySquare(int x)

{

return x \* x;

}

Результат работы программы не изменится - на печать будет выведено число 25.



В этом разделе вы узнаете о том, как передавать аргументы функции. Эти аргументы могут называться по-разному: некоторые программисты называют их аргументами, другие — параметрами или фиктивными переменными.

Аргументы функции не обязательны. Одни разработанные вами функции могут не иметь аргументов, а у других их может быть много. Тип аргументов одной функции может быть разным; это означает, что для них можно использовать любые стандартные типы данных. В следующих примерах показаны функции с аргументами разных типов. Кроме этого, в этих примерах используются функции из различных библиотек С и C++. Дополнительная информация о библиотечных функциях и их прототипах содержится в справочных руководствах по Visual C/C++.

Титул фирменного бланка выглядел бы несколько лучше, если бы текст был сдвинут к центру. Мы сможем поместить текст в центре, если напечатаем нужное число пробелов перед выводом требуемой строки. Воспользуемся некоторой функцией для печати пробелов. Наша функция space() (давайте назовем ее так) будет очень напоминать функцию starbar(), за исключением того, что на этот раз между функцией main() и функцией space() должна быть установлена связь, так как необходимо сообщить последней функции о требуемом числе пробелов.

Рассмотрим это более конкретно. В строке, состоящей из звездочек, 65 символов, а в строке MEGATHINK, INC. — 15. Поэтому в нашем первом варианте программы вслед за этим сообщением шло 50 пробелов. Чтобы сместить текст к центру, нужно сначала напечатать 25 пробелов, а потом текст, в результате чего слева и справа от данной фразы окажется по 25 пробелов. Следовательно, необходимо иметь возможность передать величину «25» функции, печатающей пробелы. Мы применяем тот же способ, что и при передаче символа '\*' функции putchar(): используем аргумент. Тогда запись space(25) будет означать, что необходимо напечатать 25 пробелов. 25 — это аргумент. Мы будем вызывать функцию sрасе() три раза: один раз для каждой строки адреса. Вот как выглядит эта программа:

/\* титул фирменного бланка2 \*/

#include<stdio.h>

#include<string.h>

#define NAME "MEGATHINK, INC."

#define ADDRESS " 10 Megabuck Plaza"

#define PLACE "Megapolis, CA 94904"

void starbar();

void space(int);

void main()

{

int spaces;

starbar();

space(25); /\* space() использует в качестве аргумента константу \*/

printf("%s\n", NAME);

spaces = (65 - strlen(ADDRESS))/2;

/\* мы заставляем программу вычислять, сколько пропустить пробелов \*/

space(spaces); /\* аргументом является переменная \*/

printf("%s\n", ADDRESS);

space((65-strlen(PLACE))/2); /\* аргументом является выражение \*/

printf("%s\n", PLACE);

starbar();

}

/\* определение функции starbar() \*/

#define LIMIT 65

void starbar()

{

int count;

for (count=1; count<=LIMIT; count++ )

putchar('\*'); putchar('\n');

}

/\* определение функции space() \*/

void space(int number)

{

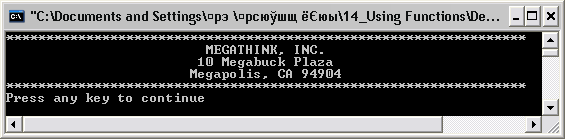
int count; /\* дополнительная переменная описывается после фигурной скобки \*/

for (count=1; count<=number; count++)

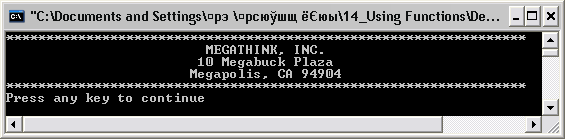
putchar(' ');

}

Обратите внимание на то, как мы экспериментировали при вызовах функции space(): мы задавали аргумент тремя различными способами. Являются ли все они работоспособными? Да — и вот доказательство.



Обратите внимание на то, как мы экспериментировали при вызовах функции space(): мы задавали аргумент тремя различными способами. Являются ли все они работоспособными? Да — и вот доказательство.



Рассмотрим сначала, как определить функцию с одним аргументом, после чего перейдем к вопросу о том, как она используется.

Определение нашей функции начинается с двух строк:

space(number)

int number;

Первая строка информирует компилятор о том, что у функции space() имеется аргумент и что его имя number. Вторая строка — описание, указывающее компилятору, что аргумент number имеет тип int. Обратите внимание: аргумент описывается перед фигурной скобкой, которая отмечает начало тела функции. Вообще говоря, вы можете объединить эти две строки в одну:

space(int number;)

Независимо от формы записи переменная number называется «формальным» аргументом. Фактически это новая переменная, и в памяти компьютера для нее должна быть выделена отдельная ячейка.

Посмотрим, как можно пользоваться этой функцией.

Задача в данном случае состоит в том, чтобы присвоить некоторую величину формальному аргументу number. После того как эта переменная получит свое значение, программа сможет выполнить свою задачу. Мы присваиваем переменной number значение фактического аргумента при вызове функции. Рассмотрим наш первый случай использования функции space():

space(25);

Фактический аргумент здесь 25, и эта величина присваивается формальному аргументу — переменной number, т. е. вызов функции оказывает следующее действие:

number = 25;

Короче говоря, формальный аргумент — переменная в вызываемой программе, а фактический аргумент — конкретное значение, присвоенное этой переменной вызывающей программой. Как было показано в нашем примере, фактический аргумент может быть константой, переменной или даже более сложным выражением. Независимо от типа фактического аргумента он вначале вычисляется, а затем его величина (в данном случае некоторое целое число) передается функции. Рассмотрим, например, наше последнее обращение к функции space().

space((65- strlen(PLACE))/2);

Сначала было вычислено значение длинного выражения, образующего фактический аргумент; оно оказалось равным 26. Затем величина 26 присваивается переменной number. Функция не знает и не «хочет» знать, является ли поступившее число константой, значением некоторой переменной или более общего выражения. В заключение повторим снова, что фактический аргумент — это конкретное значение, которое присваивается переменной, называемой формальным аргументом.

Рассматривая функцию space() как черный ящик, можно сказать, что ее вход — это число пропущенных позиций, а выход — фактический пропуск позиций. Вход связан с функцией через аргумент. С помощью аргумента обеспечивается связь между функциями main() и space(). В то же время переменная count описана внутри тела функции, и другие функции ничего не знают о ней. Указанная переменная является частью механизма, скрытого внутри черного ящика. Это не та же переменная, что count в starbar().

10_03

### Формальные и фактические параметры.

Каждое описание функции содержит некоторый список аргументов, называемый списком формальных параметров. Элементы в этом списке необязательны, поэтому список может быть как пустым, так и содержать комбинацию элементов любого типа данных, например integer, float и character.

Если какая-то программа вызывает функцию, то она передает этой функции список параметров, называемый списком фактических параметров. Если программа соответствует стандарту ANSI С, то списки формальных и фактических параметров полностью совпадают, хотя на практике строгая проверка не выполняется.

Рассмотрим следующий пример на С:

printf("This is hexadecimal %x and octal %o",ians);

В данном примере функции printf() передается только один параметр, хотя ожидаются два. Если список параметров неполон, то недостающие параметры инициализируются произвольными значениями. В C++ эта проблема отчасти решена: в списке формальных параметров можно указывать значения по умолчанию. Если некоторый параметр отсутствует в списке фактических параметров, то автоматически подставляется его значение по умолчанию. Например, в C++ можно записать следующий прототип функции:

int iyourfunction (int it, float fu=4.2, int iv=10)

Если при вызове функции iyourfunction() не указаны параметры fu или iv, то будут использоваться указанные значения (4.2 или 10). В C++ все формальные параметры, имеющие значения по умолчанию, необходимо ставить в конце списка формальных параметров. Другими словами, допустимы вызовы iyourfunction(10) и iyourfiinction(10,15.2). Если же значение fu не указано, то значение iv указывать нельзя.

**Параметры и/или аргументы?**

[В чем разница аргумента и параметра?](https://ru.stackoverflow.com/questions/510254/%d0%92-%d1%87%d0%b5%d0%bc-%d1%80%d0%b0%d0%b7%d0%bd%d0%b8%d1%86%d0%b0-%d0%b0%d1%80%d0%b3%d1%83%d0%bc%d0%b5%d0%bd%d1%82%d0%b0-%d0%b8-%d0%bf%d0%b0%d1%80%d0%b0%d0%bc%d0%b5%d1%82%d1%80%d0%b0)

***Параметром*** (формальным параметром) функции называется переменная в функции, которая будет содержать передаваемое снаружи входное значение.

***Аргументом*** называют фактическую переменную или выражение, значение которого используется как входное значение при вызове функции.

Пример:

double Negate(double x)

{

return -x;

}

Здесь x — параметр функции Negate.

double r = Negate(5.0);

Здесь 5.0 — аргумент вызова функции Negate.

Таким образом, набор параметров (то есть, «переданных снаружи» переменных) функции фиксирован, а вот аргументы (то есть, выражения, значения которых попадут в эти переменные) при каждом из вызовов могут быть свои.

Разница между параметром и аргументом сродни разнице между переменной и ее значением. Впрочем, эти два термина очень часто используют в качестве синонимов. Если вы случайно употребите одно вместо другого (не на экзамене!), вас без проблем поймут.

**Формальные и фактические параметры**

Важно различать:

* формальный параметр — аргумент, указываемый при объявлении или определении [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).
* фактический параметр — аргумент, передаваемый в [функцию](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) при её вызове;

Пример на языке [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)):

// Описание функции. int a - формальный параметр, имя параметра может отсутствовать.

int myfunction(int a);

// Определение функции. int b - формальный параметр, имя параметра может не совпадать с указанным при объявлении функции.

int myfunction(int b)

{

return 0;

}

int main()

{

int c=0;

myfunction(c); // Вызов функции. c - фактический параметр.

return 0;

}

### Аргумент типа void.

В соответствии с ANSI С, отсутствие списка аргументов функции должно быть указано явно при помощи ключевого слова void. В C++ использование void пока не обязательно, но считается целесообразным. В следующей программе имеется простая функция voutput(), не имеющая параметров и не возвращающая никакого значения. Функция main() вызывает voutput(). При выходе из voutput() управление возвращается функции main(). Трудно придумать более простую функцию

/\*08FVOID.C

Программа на С печатает сообщение при помощи функции.

В функции используются параметр типа void и стандартная

библиотечная функция С sqrt()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void voutput(void);

main()

{

/\* Программа определяет квадратный корень \*/

printf("This programm will find the square root\n\n\n");

voutput();

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void voutput(void)

{

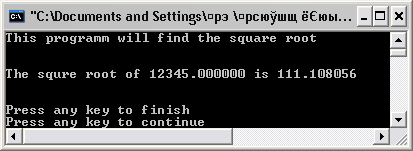
double dt=12345.0;

double du;

du=sqrt(dt);

printf("The squre root of %lf is %lf\n",dt,du);

}



Обратите внимание, что функция voutput() вызывает библиотечную функцию С, называемую sqrt(). Прототип sqrt() находится в файле math.h. У функции один параметр в формате числа двойной длины, и возвращает она результат извлечения квадратного корня тоже в виде числа двойной длины.

### Символьные параметры.

Функции можно передавать символьные значения. В следующем примере в функции main() одиночный символ считывается с клавиатуры и передается функции voutput(). Символ считывается функцией getch(). В стандартной библиотеке С имеются другие функции, тесно связанные с функцией getch(): getc(), getchar() и getcher(). Функция getch() получает символ от стандартного устройства ввода (клавиатуры) и возвращает символьное значение, не отображая его на экране:

//08FCHAR.C

/\*Программа на С считывает символ с клавиатуры, передает его функции

и печатает сообщение, использующее этот символ\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void voutput(char c);

main()

{

char cyourchar;

/\* Введите один символ с клавиатуры \*/

printf("Enter one character from the keyboard. \n");

cyourchar=getch();

voutput(cyourchar);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void voutput(char c)

{

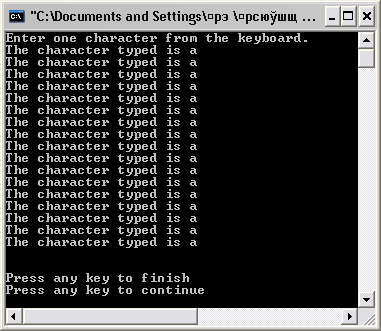
int j;

for(j=0;j<=16;j++)

/\* Введен символ ... \*/

printf("The character typed is %c \n",c);

}



По листингу видно, что функции передается единственный символ. Функция 16 раз печатает сообщение и символ. Формат %с в функции printf() задает печать одиночного символа.

### Целочисленные параметры.

В следующем примере одно целое число вводится с клавиатуры при помощи функции С scanf() и передается функции vside(), в которой на основе полученного значения, означающего длину стороны, вычисляются и печатаются площадь квадрата, объем куба и площадь поверхности куба.

//08FINT.C

/\*Программа на С вычисляет значения на основании введенной длины.

Функция получает параметр типа int, введенный с

клавиатуры при помощи функции scanf()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void vside(int is);

main()

{

int iyourlength=0;

/\* Введите с клавиатуры длину как целое число \*/

printf("Enter the length, as an integer from the keyboard. \n");

scanf("%d",&iyourlength);

vside(iyourlength);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vside(int is)

{

int iarea, ivolume,isarea;

iarea=is\*is;

ivolume=is\*is\*is;

isarea=6\*iarea;

/\* Длина стороны равна \*/

printf("The lenth of a side is %d \n",is);

/\* Квадрат будет иметь площадь \*/

printf("The square would have an area of %d \n",iarea);

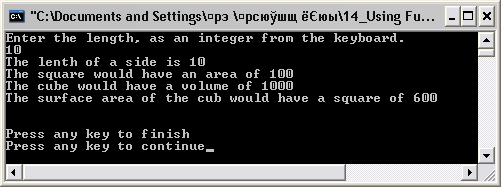
/\* Куб будет иметь объем \*/

printf("The cube would have a volume of %d \n",ivolume);

/\* Площадь поверхности куба \*/

printf("The surface area of the cub would have a square of %d \n",isarea);

}



Обратите внимание на то, что переменная is и все вычисленные значения имеют целый тип. А что произошло бы с типами вычисляемых значений, если бы переменная is представляла собой радиус круга и сферы?

### Параметры в формате чисел с плавающей точкой.

Числа с плавающей точкой так же легко передавать в качестве параметров функции, как и целые значения. В следующем примере на С в функцию, названную vhypotenuse(), передаются два числа с плавающей точкой. Для ввода с клавиатуры обоих чисел используется функция scanf().

//08FFLOAT.C

/\*Программа на С вычисляет гипотенузу правильного треугольника.

Функция использует параметры типа float, которые вводятся с

клавиатуры при помощи функции scanfО\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void vhypotenuse(float fx, float fy);

main()

{

float fxlen,fylen;

/\* Введите основание правильного треугольника \*/

printf("Enter the base of the right triangle from the keyboard. \n");

scanf("%f",&fxlen);

/\* Введите высоту правильного треугольника \*/

printf("Enter the heite of the right triangle from the keyboard. \n");

scanf("%f",&fylen);

vhypotenuse(fxlen, fylen);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vhypotenuse(float ft, float fu)

{

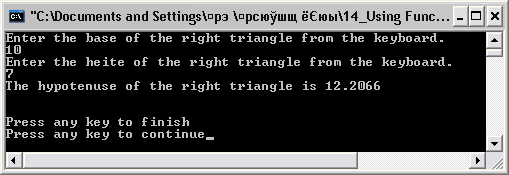
double dresult;

dresult=hypot((double) ft, (double) fu);

/\* Гипотенуза правильного треугольника равна \*/

printf("The hypotenuse of the right triangle is %g \n",dresult);

}



Следует заметить, что оба параметра функции vhypotenuse() преобразуются в тип double при использовании функции hypot() из math.h. Все функции, описанные в заголовочном файле math.h, получают и возвращают значения типа double. В своих программах вы можете использовать и другие математические функции, перечисленные в табл. 8.1. Для получения более подробной информации вы можете также просмотреть содержимое файла math.h.

Таблица. Математические функции, описанные в заголовочном файле Microsoft math.h

|  |  |
| --- | --- |
| acos, acosl | Арккосинус |
| asin, asinl | Арксинус |
| atan, atanl | Арктангенс |
| atan2,atan2l | Арктангенс |
| bessel | Функции Бесселя |
| \_cabs*,* \_cabsl | Абсолютное значение комплексного числа |
| ceil, ceill | Целочисленное максимальное значение |
| \_chgsign | Инвертирование знака |
| \_clear87, clearfp | Чтение и сброс слова состояния числа с плавающей точкой |
| \_control87, \_controlfp | Чтение старого управляющего слова числа с плавающей точкой и установка нового |
| \_copysign | Возвращает число x со знаком числа y |
| cos,cosl | Косинус |
| cosh, coshl | Гиперболический косинус |
| \_dieeetomsbin | Преобразование IEEE-числа двойной точности в двоичный формат Microsoft |
| div | Деление одного целого на другое, возвращается частное и остаток |
| \_dmsbintoieee | Преобразование Microsoft-числа двойной точности в формат IEEE |
| exp, expl | Степенная функция |
| fabs, fabsl | Абсолютное значение |
| \_fieeetomsbin | Преобразование IEEE-числа одинарной точности в двоичный формат Microsoft |
| \_finite | Проверка числа с плавающей точкой на бесконечность |
| floor, floorl | Нахождение наибольшего целого, меньшего или равного аргументу |
| fmod, fmodl | Нахождение остатка |
| \_fmsbintoieee | Преобразование Microsoft-числа одинарной точности в формат IEEE |
| \_fpclass | Возвращает слово состояния с информацией о классе чисел с плавающей точкой |
| *\_*fpieee\_flt | Вызов описанного пользователем обработчика исключительных ситуаций для чисел с плавающей точкой IEEE-стандарта |
| \_fpreset | Повторная инициализация пакета математических функций |
| frexp, frexpl | Вычисление экспоненциального значения |
| \_hypot, \_hypotl | Вычисление гипотенузы правильного треугольника |
| \_isnan | Проверка числа с плавающей точкой на значение “не число” (NAN) |
| ldexp, ldexpl | Произведение от аргумента |
| ldiv | Деление одного целого long на другое, возвращается частное и остаток |
| log, logl | Натуральный логарифм |
| log10, log10l | Десятичный логарифм |
| \_logb | Выделение показателя числа с плавающей точкой |
| \_irotl, \_irotr | Сдвиг числа unsigned long int влево или вправо |
| \_matherr, \_matherrl | Обработка математических ошибок |
| \_max, \_min | Определение большего или меньшего из двух значение |
| modf, modfl | Деление аргумента на целую и дробную части |
| \_nextafter | Определение следующего значения |
| pow, powl | Вычисление значения, возведенного в степень |
| rand | Получение псевдослучайного числа |
| \_rotl, \_rotr | Сдвиг числа unsigned int влево или вправо |
| \_scalb | Степень числа 2, определяемая аргументом |
| sin, sinl | Синус |
| sinh, sinhl | Гиперболический синус |
| sqrt, sqrtl | Квадратный корень |
| srand | Инициализация датчика псевдослучайных чисел |
| \_status87, \_statusfp | Получение слов состояния числа с плавающей точкой |
| tan, tanl | Тангенс |
| tanh, tanhl | Гиперболический тангенс |

### Параметры в формате чисел двойной длины.

Тип чисел двойной длины double обеспечивает очень большую точность чисел с плавающей точкой. Все функции, описанные файле math.h, получают и возвращают числа типа double. В следующей программе два числа двойной длины вводятся с клавиатуры. Функция, названная vpower(), возводит первое число в степень, определяемую вторым числом. Поскольку оба числа имеют тип double, можно, к примеру, вычислить, 45.7 в степени 5.2 равно 428118741.757.

//08FDOUBL.C

/\*Программа на С возводит число в некоторую степень.

Функция использует параметр типа double и функцию pow()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void vpower(double dt, double du);

main()

{

double dtnum,dunum;

/\* Введите основание степени \*/

printf("Enter the base number from the keyboard. \n");

scanf("%lf",&dtnum);

/\* Введите показатель степени \*/

printf("Enter the power from the keyboard. \n");

scanf("%lf",&dunum);

vpower(dtnum, dunum);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vpower(double dt, double du)

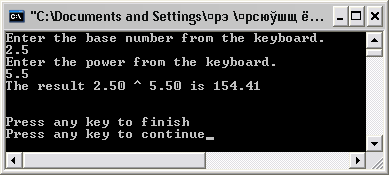
{

double danswer;

danswer=pow(dt, du);

printf("The result %.2f ^ %.2f is %.2f \n",dt,du,danswer);

}



Здесь для возведения числа в некоторую степень используется библиотечная функция pow() с прототипом в файле math.h.

### Массивы в качестве параметров.

В следующем примере содержимое некоторого массива передается в функцию в качестве параметра, вызываемого по ссылке. В этом случае адрес первого элемента массива передается через указатель.

//08FPNTR.C

/\*Программа на С передает функции массив в качестве параметра.

Для передачи информации о массиве используется указатель\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void voutput(int \*pinums);

main()

{

int iyourarray[7]={2,7,15,32,45,3,1};

/\* Передать информацию о массиве в функцию \*/

printf("Send array information to function. \n");

voutput(iyourarray);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void voutput(int \*pinums)

{

int t;

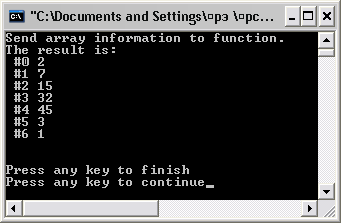
/\* Результат равен \*/

printf("The result is:\n");

for(t=0;t<7;t++)

printf(" #%d %d\n",t,pinums[t]);

}



Обратите внимание на то, что при вызове функции указывается только имя iyourarray. Далее вы узнаете о массивах подробнее. В данном примере при указании имени массива в функцию передается адрес первого его элемента. Так как iyourarray — массив целых чисел, массив можно передать, задавая указатель соответствующего типа.

Также допустимо передать информацию по адресу и для безразмерного массива. В следующем примере показано, как это можно сделать на C++. (Такой же подход возможен и в С.) Информация, содержащаяся в массиве iyourarray, передается путем указания адреса его первого элемента.

**// 08FARRAY.CPP**

// Программа на C++ вызывает функцию и передает ей массив.

// Функция вычисляет среднее значение элементов массива

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void avg(float fnums[]);

main()

{

float iyourarray[8]={(float) 12.30,(float) 25.70,

(float) 82.10,(float) 6.00,(float) 7.01,

(float) 0.25,(float) 4.20,(float) 6.28};

// Передать информацию усредняющей функции

cout << "Send array information to function.\n";

avg(iyourarray);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void avg(float fnums[])

{

int iv;

float fsum=0.0;

float faverage;

cout << "The result is:\n";

for(iv=0;iv<8;iv++)

{

fsum+=fnums[iv];

cout << "number " << iv+1 << " is " << fnums[iv] << endl;

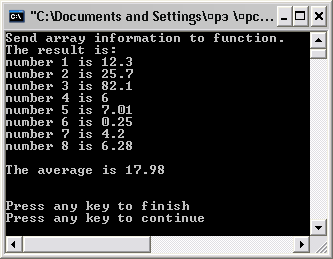
}

faverage=fsum/iv;

// Среднее значение равно

cout << "\nThe average is " << faverage << endl;

}



Среднее значение определяется при помощи суммирования всех элементов и деления на их общее число. Для форматированного вывода на экран используется поток cout.

### Аргументы по умолчанию

C++ допускает при вызове функций опускать некоторые ее параметры. Достигается это указанием в прототипе функции значений аргументов по умолчанию. Например, функция, прототип которой приведен ниже, может при вызове иметь различный вид в зависимости от ситуации.

// Прототип функции:

void ShowInt (int i,

bool Flag = true,

char symbol = '\n');

//Вызовы функции ShowInt:

ShowInt (1, false, ‘a’);

ShowInt (1, false);

ShowInt(3);

В первом случае все три аргумента заданы явно, поэтому работа функции осуществляется в обычном режиме. Во втором вызове функции передается два параметра из трех, причем вместо последнего аргумента подставляется значение по умолчанию, а именно символ '\n'. Третий вариант обращения к функции сообщает только один целочисленный параметр, а в качестве остальных аргументов. Используются значения по умолчанию: логическая переменная со значением true и символьная переменная со значением '\n'.

Для используемых параметров по умолчанию существует обязательное правило - все параметры справа от аргумента по умолчанию должны иметь значение по умолчанию. Так, в приведенном выше прототипе нельзя было бы, указав значение параметра по умолчанию для целочисленной переменной i, пропустить определение любого из остальных аргументов по умолчанию.

Рассмотрим пример, в котором осуществляется вывод знакового числа двойной точности с указанием количества значащих символов Другими словами, определим функцию, принимающую в качестве одного из параметров число выводимых знаков. Для решения поставленной задачи можно воспользоваться функцией возведения числа в степень pow() и функцией взятия модуля от длинного числа с плавающей точкой fabs1 (), прототипы которых содержатся в заголовочном файле math. h:

#include <math.h>

#include <iostream>

using namespace std;

void Out (double Numb, double Sig=1, bool Flg=true);

int main()

{

double Mpi = -3.141592654;

Out(Mpi, 4, false);

Out(Mpi, 2);

Out(Mpi);

getchar(); getchar();

return 0;

}

void Out(double numb, double sig, bool fig)

{

if(!fig)

numb = fabsl(numb);

numb = (int) (numb \* pow(10, sig));

numb = numb / pow(10,sig);

cout << numb << '\n';

}

В теле программы производится вызов одной и той же функции Out () с различным числом параметров для вывода значения переменной двойной точности Mpi. В результате работы программы на печать будут выведены следующие значения:

3,1415

-3,14

-3,1



Величина, указываемая в аргументах по умолчанию, может быть не только константным выражением - она может быть глобальной переменной или значением, возвращаемым некоторой функцией.

## Возвращение значения функцией: оператор return.

Создадим функцию, вычисляющую абсолютную величину числа. Абсолютная величина числа — это его значение (если отбросить знак). Следовательно, абсолютная величина 5 равна 5, а абсолютная величина -3 равна 3. Мы назовем эту функцию abs(). Входом Для abs() может быть любое число, для которого мы хотим найти Абсолютную величину. Выходом функции будет соответствующее неотрицательное число. Входная величина может обрабатываться благодаря наличию аргумента; выходная величина возвращается (т. е. выдается), как вы увидите ниже, при помощи ключевого слова языка Си — return. Поскольку функция abs() должна быть вызвана другой функцией, мы создадим простую программу main(), основной целью которой будет проверка, работает ли функция abs(). Программа, спроектированная для того, чтобы проверять работу функции именно таким образом, называется «драйвером». Драйвер подвергает функцию последовательным проверкам. Если результаты оказываются удовлетворительными, то ее можно поместить в программу, заслуживающую большего внимания. (Термин «драйвер» обычно относится к программам, управляющим работой устройств.) Приведем далее наш драйвер и функцию, вычисляющую абсолютную величину числа:

/\* abs.драйвер \*/

#include<stdio.h>

int abs(int );

void main()

{

int a= 10, b= 0, c= -22;

int d, e, f;

d = abs(a);

e = abs(b);

f = abs(c);

printf(" %d %d %d\n", d, e, f);

}

/\* функция, вычисляющая величину числа \*/

int abs(int x)

{

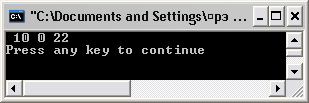
int y;

y = (x < 0) ? - x : x; /\*вспомните операцию ? : \*/

return (y); /\* возвращает значение у вызывающей программе \*/

}

Результат работы программы выглядит так:



Сначала вспомним операцию условия ?:. Эта операция в функции abs() выполняется следующим образом: если х меньше 0, у полагается равным — х; в противном случае у полагается равным x. Это как раз то, что нам нужно, поскольку если х равен —5, то у равен -(-5), т. е. 5.

Ключевое слово return указывает на то, что значение выражения, заключенного в круглые скобки, будет присвоено функции, содержащей это ключевое слово. Поэтому, когда функция abs() впервые вызывается нашим драйвером, значением abs(a) будет число 10, которое затем присваивается переменной d.

Переменная у является внутренним объектом функции abs(), но значение у передается в вызывающую программу с помощью оператора return. Действие, оказываемое оператором

d = abs(a);

по-другому можно выразить так:

abs(a);

d = у;

Можно ли в действительности воспользоваться такой записью? Нет, так как вызывающая программа даже не подозревает о том, что переменная у существует.

Возвращаемое значение можно присвоить переменной, как в нашем примере, или использовать как часть некоторого выражения, например, следующим образом:

answer = 2\*abs(z) + 25;

printf(" %d\n" , abs(—32 + answer));

Оператор return оказывает и другое действие. Он завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Это происходит даже в том случае, если оператор return является не последним оператором тела функции. Следовательно, функцию abs() мы могли бы записать следующим образом:

/\* функция, вычисляющая абсолютную величину числа, вторая версия \*/

abs(x)

int x;

{

if (x < 0)

return(-x);

else

return(x);

}

Эта версия программы проще, и в ней не используется дополнительная переменная у. Для пользователя, однако, обе версии неразличимы, поскольку у них имеется один и тот же вход и они обеспечивают один и тот же выход. Только внутренние структуры обеих функций различны. Даже версия данной программы, приведенная ниже, работает точно так же:

/\* функция, вычисляющая абсолютную величину числа, третья версия \*/

abs(x)

int(x);

{

if (х < 0)

return(-x);

else

return(x);

printf(" Профессор Флеппард — болван. \n");

}

Наличие оператора return препятствует тому, чтобы оператор печати printf() когда-нибудь выполнился в программе. Профессор Флеппард может пользоваться в своих программах объектным кодом, полученным в результате компиляции данной функции, и никогда не узнает об истинных чувствах своего студента-программиста.

Вы можете также использовать просто оператор

return;

Его применение приводит к тому, что функция, в которой он содержится, завершает свое выполнение и управление возвращается в вызывающую функцию. Поскольку у данного оператора отсутствует выражение в скобках, никакое значение при этом не передается функции.

## Типы функций.

В следующем разделе приводятся примеры программ на С и C++ для каждого из важнейших типов данных, возвращаемых функциями. Тип функции показывает тип возвращаемого ею значения. Ни в одном из примеров в предыдущих разделах функции не возвращали информацию, и поэтому они имели тип void (пустой).

Тип функции определяется типом возвращаемого ею значения, а не типом ее аргументов. Если указание типа отсутствует, то по умолчанию считается, что функция имеет тип int. Если значения функции не принадлежат типу int, то необходимо указать ее тип в двух местах.

* Описать тип функции в ее определении:

char pun(ch, n) /\* функция возвращает символ \*/

int n;

char ch;

float raft(num) /\* функция возвращает величину типа float \*/

int num;

* Описать тип функции также в вызывающей программе. Описание функции должно быть приведено наряду с описаниями переменных программы; необходимо только указать скобки (но не аргументы) для идентификации данного объекта как функции.

main()

{

char, rch, pun();

float raft;

Запомните! Если функция возвращает величину не типа int, указывайте тип функции там, где она определяется, и там, где она используется.

### Функции типа void.

Раз уж во всех предыдущих примерах использовался тип void, то в данном разделе пример такого типа будет более развернутым. Как вы уже знаете, в С и C++ числовую информацию можно вводить и выводить в шестнадцатеричном, десятичном и восьмеричном форматах — но не в двоичном. Представление данных в двоичном формате удобно для двоичной арифметики или для создания битовых масок. Функция vbinary() преобразует десятичное число, введенное с клавиатуры, в двоичное и выводит его на экран. Двоичные цифры не упаковываются как одно двоичное число, а хранятся отдельно в массиве. Таким образом, для просмотра двоичного числа необходимо распечатать содержимое массива.

//08VOIDF.C

/\*Программа на С иллюстрирует использование функции типа void.

Программа печатает двоичное представление числа\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

void vbinary(int ivalue);

main()

{

int ivalue;

/\* Введите число с основанием 10 для преобразования в двоичное \*/

printf("Enter a decimal number to conversion to binary.\n");

scanf("%d",&ivalue);

vbinary(ivalue);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vbinary(int idata)

{

int t=0;

int iyourarray[50];

while(idata != 0)

{

iyourarray[t]=(idata %2);

idata/=2;

t++;

}

t--;

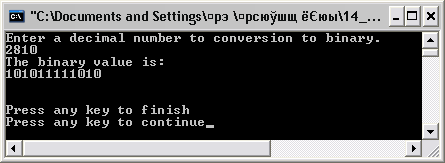
printf("The binary value is:\n");

for(;t>=0;t--)

printf("%1d",iyourarray[t]);

printf("\n");

}



Процесс преобразования чисел от большего основания к меньшему представляет собой достаточно простой математический алгоритм. Например, десятичные числа можно преобразовать в другую систему счисления при помощи последовательного деления числа на значение нового основания. Если выполняется переход от основания 10 к основанию 2, то число с основанием 10 несколько раз делится на 2. Получаются частное и остаток. Частное становится делимым для каждого последующего деления. Остаток становится очередной цифрой в преобразованном числе. В случае двоичного преобразования остаток может быть или 1, или 0. Вот пример того, как число 13 преобразуется в двоичное:

08_01

В функции vbinary() цикл while выполняется до тех пор, пока в результате арифметических операций значение переменной idata не станет равным нулю. Операция деления по модулю определяет остаток и запоминает очередной бит в массиве. Затем происходит деление переменной idata, причем сохраняется только целая часть. Этот процесс повторяется до тех пор, пока частное (та же самая переменная idata) не станет равным нулю.

Отдельные биты в массиве, образующие двоичный результат, должны выводится из массива в обратном порядке. Это можно увидеть по листингу программы. Взгляните на цикл for, используемый в функции. Можете ли вы придумать какой-нибудь алгоритм данного преобразования, чтобы двоичное представление хранилось не в массиве, а в переменной?

### Функции типа char.

В этом разделе вы увидите слегка измененный вариант описанного ранее примера. Функция С clowercase() имеет символьный параметр и возвращает также символьное значение. В данном примере некоторая заглавная буква вводится с клавиатуры и передается в функцию, в которой для преобразования символа в строчную букву используется библиотечная функция tolower() (из стандартной библиотеки с прототипом в файле ctype.h). Близкие tolower() функции: toascii() и toupper().

//08CHARF.C

/\*Программа на С иллюстрирует использование функции типа char.

Функция получает символ в верхнем регистре и преобразует его

в нижний регистр\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

char clowercase(char c);

main()

{

char clowchar,chichar;

/\* Введите символ в верхнем регистре \*/

printf("Enter an uppercase character.\n");

chichar=getchar();

clowchar=clowercase(chichar);

printf("Entering character is %c\nConverting character is %c\n",chichar,clowchar);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

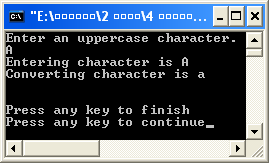
}

char clowercase(char c)

{

return(tolower(c));

}



### Функции типа int.

Следующая функция имеет целочисленные параметры и возвращает целые значения. Функция icube() получает из main() некоторое число (0, 2, 4, 6, 8, 10 и так далее), возводит его в куб и возвращает целое значение в main(). Исходное число и его куб выводятся на экран.

//08INTF.C

/\*Программа на С иллюстрирует использование функции типа int.

Функция считывает по очереди целые числа и возвращает

их значения, возведенные в куб\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

int icube(int ivalue);

main()

{

int k,inumbercube;

for(k=0;k<20;k+=2)

{

inumbercube=icube(k);

/\* Куб числа ... равен ... \*/

printf("The cube of the number %2d is %d\n",k,inumbercube);

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

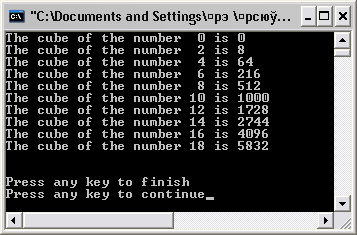
}

int icube(int ivalue)

{

return(ivalue\*ivalue\*ivalue);

}



### Функции типа long.

Следующий пример представляет собой программу на C++, получающую в качестве параметра целочисленную переменную и возвращающую значение типа long. Данный тип, используемый в Visual C/C++ и в других распространенных компиляторах, не является стандартным типом языка ANSI С. Приведенная функция возводит число 2 в некоторую целую степень.

// 08LONGF.CPP

// Программа на C++, иллюстрирующая использование функции типа long.

// Эта функция получает целые числа и возвращает значения,

// равные заданной степени числа 2

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

long lpower(int ivalue);

main()

{

int k;

long lanswer;

for(k=0;k<21;k++)

{

lanswer=lpower(k);

// 2 в степени ... равно ...

cout << "2 raised to the " << k

<< " power is " << lanswer << endl;

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

long lpower(int ivalue)

{

int t;

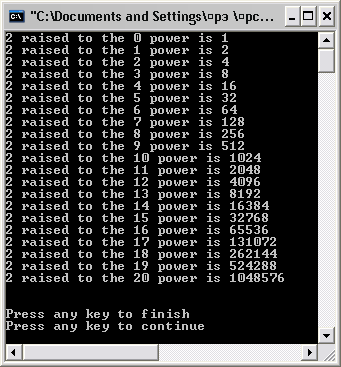
long lseed=1;

for(t=0;t<ivalue;t++)

lseed\*=2;

return(lseed);

}



Эта функция умножает исходное число само на себя столько раз, сколько нужно для возведения в заданную степень. Например, для возведения числа 2 в 6-ю степень (2 в 6-й степени) программа выполняет следующее умножение:

2\*2\*2\*2\*2\*2 = 64

Могут ли функции, описанные в файле math.h, дать тот же результат? Ответ можно найти из ранее приведенной таблицы.

### Функции типа float.

В следующем примере массив типа float передается в качестве параметра в функцию, которая возвращает значение типа float. В этом примере на C++ определяется произведение всех элементов массива.

// 08FLOATF.CPP

// Программа на C++ иллюстрирует использование функции типа float.

// Функция получает массив чисел типа float и возвращает их

// произведение в виде числа с плавающей точкой

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

float fproduct(float farray[]);

main()

{

float fmyarray[7]={(float) 4.3,(float) 1.8,(float) 6.12,(float) 3.19,

(float) 0.01,(float) 0.1,(float) 9876.2};

float fmultiplied;

fmultiplied=fproduct(fmyarray);

// Произведение всех введенных чисел равно

cout << "The product of all array enteris is: "

<< fmultiplied << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

float fproduct(float farray[])

{

int i;

float fpartial;

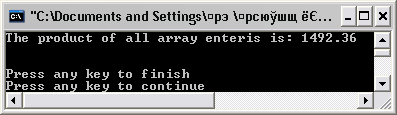
fpartial=farray[0];

for(i=1;i<7;i++)

fpartial\*=farray[i];

return(fpartial);

}



Поскольку вычисляется произведение всех элементов массива, значение его первого элемента должно быть присвоено переменной fpartial до начала цикла for. Обратите внимание на то, что цикл в функции fproduct() начинается с 1, а не с обычного нулевого значения.

### Функции типа double.

В следующем примере на С передаются и возвращаются значения типа double. Функция dtrigcosine() преобразует значение угла, выраженное в градусах, в косинус этого угла.

//08DOUBLE.C

/\*Программа на С иллюстрирует использование функции типа double.

Функция считывает целые числа от 0 до 90 и возвращает значение

косинуса каждого числа\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <math.h>

using namespace std;

const double dPi=3.14159265359;

double dtrigcosine(double dangle);

main()

{

int j;

double dcosine;

for(j=0;j<91;j++)

{

dcosine=dtrigcosine((double) j);

/\* Косинус ... градусов равен \*/

printf("The cosine of %d degrees is %19.18lf \n",j,dcosine);

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

double dtrigcosine(double dangle)

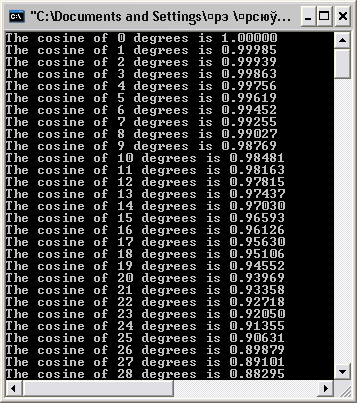
{

double dpartial;

dpartial=cos((dPi/180.0)\*dangle);

return(dpartial);

}



Следует заметить, что функция dtrigcosine() для получения результата использует функцию cos(), описанную в файле math.h. Для всех тригонометрических функций значение угла должно быть преобразовано из градусов в радианы. Напомним, что pi радианов равно 180 градусам.

## Важные возможности C++.

Для создания функций на C++ можно использовать некоторые специфические возможности языка. Одним из его достоинств является возможность написания встроенных функций. Код некоторой встроенной функции воспроизводится в том месте программы, в котором она вызывается. Поскольку компилятор располагает этот код в точке вызова функции, то при использовании коротких, часто вызываемых функций, сокращается время выполнения программы.

Помимо этого, в C++ функции можно перегружать. Перегрузка позволяет использовать одно и то же имя функции для нескольких ее прототипов. Затем различные прототипы распознаются не по имени, а по типу и списку аргументов. Перегрузка очень полезна в функциях, предназначенных для работы с разными типами данных.

### Встраивание (inline).

В результате работы компилятора каждая функция представляется в виде машинного кода. Если в программе вызов функции встречается несколько раз, в местах таких обращений генерируются коды вызова уже реализованного экземпляра функции. Однако выполнение вызовов требует некоторой затраты времени. Таким образом, если тело функции небольшого размера и обращение к ней в программе происходит довольно часто, на практике можно указать компилятору вместо вызовов функции в соответствующих местах генерировать все ее тело. Осуществляется это с помощью ключевого слова inline. Teм самым увеличивается производительность реализованного кода, хотя, конечно, размер программы может увеличиваться. Компиляторы различных фирм накладывают свои ограничения на использование встраиваемых функций, поэтому перед использованием inline-функций необходимо обратиться к руководству компилятора.

Ключевое слово inline должно предшествовать первому вызову встраиваемой функции (например, содержаться в ее прототипе).

#include <iostream>

using namespace std;

// Прототип встраиваемой функции:

inline int Sum(int, int);

int main()

{

int A= 2, B= 6, C= 3;

char eol = '\n';

// Вызовы встраиваемой функции,

// генерируют все тело функции

cout << Sum (A, B) << eol;

cout << Sum (B, C) << eol;

cout << Sum (A, C) << eol;

getchar(); getchar();

return 0;

}

int Sum(int x, int y)

{

return x + y;

}



В приведенном примере в каждом месте вызова функции Sum () будет сгенерирован код тела всей функции.

Ключевое слово inline можно рассматривать как директиву или, лучше сказать, как указание компилятору C++ на встраивание этой функции. По разным причинам компилятор может проигнорировать это указание. Например, функция может быть слишком длинной. Встраиваемые функции используются в первую очередь для сокращения времени выполнения программы при частом вызове коротких функций.

// Программа на C++ иллюстрирует использование встроенных функций.

// Встроенные функции работают лучше всего в тех случаях, когда часто

// повторяются короткие функции. В этом примере некоторое сообщение

// выводится на экран несколько раз

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

// Это - встроенная функция

inline void voutput(void)

{

cout << "This is an inline function!" << endl;

}

main()

{

int t;

// Программа печатает сообщение несколько раз

cout << "This program prints a message several times." << endl;

for(t=0;t<3;t++)

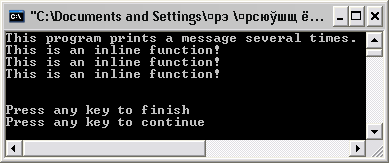
voutput();

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



### Перегрузка (overloading).

Следующий пример иллюстрирует перегрузку функции. Обратите внимание на то, как описаны прототипы двух функций с одинаковым именем и областью действия. Конкретная функция будет выбираться в зависимости от передаваемых аргументов. Функцию adder() можно вызывать с параметрами целого типа или типа float.

// Программа на C++ иллюстрирует перегрузку функций.

// Перегружаемая функция получает массив целых чисел или чисел с

// плавающей точкой и возвращает их сумму в виде числа int или float

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

int adder(int iarray[]);

float adder(float farray[]);

main()

{

int iarray[7]={5,1,6,20,15,0,12};

float farray[7]={3.3,5.2,0.05,1.49,3.12345,31.0,2.007};

int isum;

float fsum;

isum=adder(iarray);

fsum=adder(farray);

// Сумма целых чисел равна

cout << "The sum of the integer numbers is: "

<< isum << endl;

// Сумма чисел с плавающей точкой равна

cout << "The sum of the float numbers is: "

<< fsum << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int adder(int iarray[])

{

int i;

int ipartial;

ipartial=iarray[0];

for (i=1;i<7;i++)

ipartial+=iarray[i];

return (ipartial);

}

float adder(float farray[])

{

int i;

float fpartial;

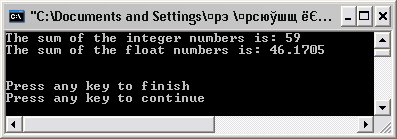
fpartial=farray[0];

for (i=1;i<7;i++)

fpartial+=farray[i];

return (fpartial);

}



При программировании перегружаемых функций имеется несколько препятствий, которые необходимо обойти. Например: если некоторые функции отличаются только типом, но не аргументами, то эти функции перегружать нельзя. Также запрещается следующий способ перегрузки:

int yourfunction(int number)

int yourfunction(int &value) //запрещается

Такой синтаксис не допускается, поскольку иначе каждый прототип получал бы аргументы одного и того же типа. Несмотря на эти ограничения, перегрузка является очень важной чертой в C++, и достаточно широко представлена в дальнейшем.

### Функции с переменным числом параметров. Многоточие (...).

Многоточие можно использовать тогда, когда по каким-то причинам предполагается создать функцию с неизвестным заранее числом идущих в функцию аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием, обязательно в конце списка формальных параметров.

Многоточие иногда используют, но в зависимости от компиляторов, один из старых способов может совсем не давать желаемого эффекта. До С++11 рекомендуется использовать только вариант с stdarg , а варианты, [показанные ниже](http://ci-plus-plus-snachala.ru/?p=63#x), рекомендуется не использовать вообще.

Для того, чтобы получить доступ ко всем параметрам, принимающих значения аргументов, заходящих в функцию, обязательно знать имя и тип хотя бы одного параметра.

В С++ можно создавать функции с переменным числом параметров. Параметры, их количество и типы, становятся известны только во время вызова функции. Формат описания функций с переменным числом параметров:

<тип> <имя> (<список явных параметров>, …)

{

<тело функции>

}

Здесь:

<**тип**> – тип возвращаемого значения,

<**список параметров**> – список параметров известных до вызова функции.

Переменный список параметров задается в заголовке функции многоточием:

int f(…)

Этот заголовок не вызывает у компилятора протестов. **Такая запись означает, что при определении функции компилятору неизвестны ни количество параметров, ни их типы, и он, естественно, не может ничего проверить. Количество параметров и их типы становятся известными только при вызове функции.**

**Однако у программиста с написанием таких функций сразу возникают проблемы. Ведь имена параметров отсутствуют! Поэтому доступ можно осуществить только одним способом – косвенным, используя указатель. Вспомним, что все параметры при вызове помещаются в стек. Если мы каким-то образом установим указатель на начало списка параметров в стеке, то, манипулируя с указателем, мы, в принципе, можем «достать» все параметры!**

Таким образом, список параметров совсем пустой быть не может, должен быть прописан хотя бы один явный параметр, адрес которого мы можем получить при выполнении программы. Заголовок такой функции может выглядеть так:

int f(int k...)

**Ни запятая, ни пробел после параметра не обязательны, хотя можно их и прописать.**

**Есть одно обстоятельство, которое ограничивает применение таких функций: при написании функции с переменным числом параметров помимо алгоритма обработки программист должен разрабатывать и алгоритм доступа к параметрам. Так что список необъявленных параметров не может быть совсем уж произвольным – в языке C++ не существует универсальных средств распознавания элементов этого списка. Это же означает, что передача аргумента не того типа, который задумывался, или не тем способом, который подразумевался при разработке, приведет к катастрофическим последствиям – компилятор-то ничего не проверяет.**

Попробуем написать функцию, вычисляющую среднее арифметическое своих аргументов. Для этого требуется решить несколько проблем

* как установиться на список параметров в стеке;
* как «перебирать» параметры;
* как закончить перебор.

**Для доступа к списку параметров нам потребуется указатель, значением которого будет адрес последнего явного параметра в списке. Ответ на второй вопрос очевиден – надо изменять значение этого указателя, чтобы переместиться на следующий параметр. Отсюда следует, что указатель должен быть типизированным, поскольку с бестиповым указателем нельзя выполнять арифметические операции. Это же означает, что программист при разработке функции с переменным числом параметров должен отчетливо себе представлять типы аргументов, которые будет обрабатывать функция. Кроме того, способ передачи параметров должен быть одинаковым для всех параметров: либо все – по значению, либо все – по ссылке, либо все – по указателю.**

Ответ на последний вопрос не вызывает затруднений. Это можно сделать одним из двух способов:

* явно передать среди обязательных параметров количество аргументов;
* добавить в конец списка аргумент с уникальным значением, по которому будет определяться конец списка параметров;

И тот, и другой способ имеют право на жизнь — все определяется потребностями задачи и вкусами программиста.

Если не один из этих способов не используется, тогда можно использовать *специальный набор макроопределений*.

Параметры функции помещаются в стек, при этом первый параметр оказывается в вершине стека. Переход от одного параметра к другому осуществляется с помощью указателей.

Задание числа дополнительных параметров с помощью первого параметра.

Функция вычисляет сумму значений дополнительных параметров. Список явных параметров состоит из одного параметра, который задает число дополнительных параметров.

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

void main()

{

int sum(int n,...); //прототип функции с переменным

//числом параметров

cout << "\n 4+6=" << sum(2, 4, 6); //10

cout << "\n 1+2+3+4+5+6=" << sum(6, 1, 2, 3, 4, 5, 6); //21

cout << "\n Parametrov net.Summa ravna:" << sum(0);

getch();

}

int sum(int n,...)//n-число суммируемых параметров

{

int s=0;

int \*p = &n; //получение адреса параметров в стеке

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

s+=\*++p; //суммируем числа

return s;

}



Для доступа к параметрам, которые один за другим попали в стек, используется адрес первого параметра **(\*p = &n;)**, таким образом, указатель устанавливается на начало списка параметров в стеке (в памяти). Затем в цикле при помощи указателя **p** перемещаемся по параметрам и суммируем их, извлекая **(\*p)** из памяти. Все параметры должны иметь **одинаковый тип**.

Проверка **соответствия** типов для дополнительных параметров **не выполняется**, поскольку компилятор не имеет информации, необходимой для проведения проверки. При вызове функции дополнительные параметры типа char и short передаются как **int**, а float — как **double**.

Реализация функции, которая в качестве первого параметра получает количество аргументов, на первый взгляд, не вызывает затруднений. Однако, если первый аргумент – целое число, то требуется преобразование указателя. И тут не все варианты проходят. Не будет работать такой вариант:

#include <iostream>

#include <stdarg.h>

using namespace std;

//Такой вариант неработоспособен

double f(int n, ...) //количество элементов

{

int \*p = &n; //указатель – «целый»

double sum = 0, count = n;

for (; n--; (double\*)p++) //преобразование int\* ->double\*

sum += (\*p);

return ((sum) ? sum / count : 0);

}

void main()

{

double y = f(4, 1.0, 2.0, 3.0, 4.0, 0.0); //NOT 2.5

cout << y << endl;

system("pause");

}



#include <iostream>

#include <stdarg.h>

using namespace std;

//Такой вариант тоже неработоспособен

double f(int n, ...) //количество элементов

{

double \*p = (double \*)&n; //преобразование адреса

double sum = 0, count = n;

for (; n--; p++) //изменение указателя

sum += (\*p);

return ((sum) ? sum / count : 0);

}

void main()

{

double y = f(4, 1.0, 2.0, 3.0, 4.0, 0.0); //NOT 2.5

cout << y << endl;

system("pause");

}



Причина кроется в том, что изменение указателя производится на столько байт, сколько в памяти занимает базовый тип. В обоих случаях мы установились не на начало списка double-параметров, а на sizeof(int) байтов «раньше» — на целую переменную. И от этого адреса происходит изменение указателя на 8 байт (sizeof(double)), что приводит к совершенно неверным результатам. Решение заключается в том, чтобы сначала изменить «целый» указатель, а потом уже его преобразовать в double \*. Так всегда необходимо делать, если тип первого параметра отличается от типов отсутствующих параметров (листинг 7.8).

Листинг 7.8. Вычисление среднего арифметического аргументов (количество)

#include <iostream>

#include <stdarg.h>

using namespace std;

double f(int n, ...) //количество элементов

{

int \*p = &n;

p++; //установка «целого» на double

double \*pp = (double \*)p; //преобразование типа указателя

double sum = 0, count = n;

for (; n--; pp++) //правильное увеличение на 8

sum += (\*pp);

return ((sum) ? sum / count : 0);

}

void main()

{

double m[] = { 1.0, 2.0, 3.0, 4.0, 0.0 };

cout << f(4, m[0], m[1], m[2], m[3], m[4]) << endl; //2.5

system("pause");

}



В строке

p++; //установка «целого» на double

операция p++ устанавливает указатель на первый элемент списка параметров типа double. Для дальнейшего изменения указателя на 8 мы использовали преобразование типа указателя:

double \*pp = (double \*)p; //преобразование типа указателя

После этой строки операция pp++ будет увеличивать указатель на sizeof(double)=8, что нам и требуется.

Мы использовали способ передачи параметров по значению. В этом случае в качестве фактических аргументов можно задавать произвольные выражения. Однако можно использовать и передачу ссылки – это несколько осложняет вызов, поскольку в этом случае в списке аргументов могут прописываться только переменные. Необходимо также помнить, что все фактические аргументы должны передаваться одинаковым способом. Тогда прототип первого варианта функции выглядит так (тело функции не изменяется):

double f(double &n, ...)

При вызове можно использовать элементы массива:

double m[] = {1.0,2.0,3.0,4.0,0.0};

cout <<f(m[0],m[1],m[2],m[3],m[4])<<endl;

Эта программа выведет на экран 2.5.

#include <iostream>

#include <stdarg.h>

using namespace std;

double f(double &n, ...) //количество элементов

{

double \*p = &n;

double sum = 0, count = 0;

while (\*p) //пока аргумент не равен нулю

{

sum += (\*p); //суммируем аргумент

p++; //перемещаемся на следующий аргумент

count++; //считаем количество аргументов

}

return ((sum) ? sum / count : 0);

}

void main()

{

double m[] = { 1.0, 2.0, 3.0, 4.0, 0.0 };

cout << f(m[0], m[1], m[2], m[3], m[4]) << endl; //2.5

system("pause");

}



Язык C++ в качестве элементов переменного списка аргументов разрешает прописывать указатели. Однако обработка такого варианта вызывает сложности – нам требуется двойной косвенный доступ, а указатель для доступа к стеку — «одноразовый». Таким образом, если передавать параметры-указатели, то в приведенной выше программе (с первым параметром-количеством) значение \*рр – это не число типа double, а адрес этого числа. Тут без «обмана» компилятора не обойтись, поскольку он просто так не пропускает преобразование «одноразовой» косвенности в двойную. Но мы помним, что все типизированные указатели, независимо от типа и косвенности всегда представляют собой адрес, размер которого в Intel – 4 байта. Тогда можно использовать union (листинг 7.9).

Листинг 7.9. Переменный список параметров-указателей (количество)

#include <iostream>

#include <stdarg.h>

using namespace std;

double f(int n, ...)

{

int \*p = &n; //«одноразовый» указатель

p++;//«достаем» список параметров-указателей

union Pointer

{

double \*\*pp; double \*kp;

}; //«подстава» указателей

Pointer A;

A.kp = (double \*)p; //«обманываем» компилятор

double sum = 0, count = n;

for (; n--; A.pp++) //изменяем двойной указатель!

sum += (\*\*A.pp); //двойной доступ!

return ((sum) ? sum / count : 0);

}

void main()

{

double y = f(4, 1.0, 2.0, 3.0, 4.0, 0.0); //NOT 2.5

cout << y << endl;

system("pause");

}

Хотя по стандарту такое использование union означает undefined behaviour (неопределенное поведение), и непереносимо, но на практике (например, на платформе Intel) работает хорошо. Однако при переносе на другую платформу надо будет проверять корректность работы такой функции.

Необходимо обратить внимание на то, что изменяем мы «двойной» указатель – «одноразовый» использовать нельзя, поскольку будет изменение на sizeof(double)=8 байт, а нам требуется изменение на sizeof(double\*)= 4 байта. И при суммировании используется двойной косвенный доступ. Обращение к такой функции выполняется так:

cout <<f(2,&a,&b)<<endl;

Аналогичная функция без счетчика аргументов может использовать в качестве признака окончания списка параметров нулевой указатель. Но в данном случае «обманывать» компилятор не требуется — в функции непосредственно используется «двойной» указатель (листинг 7.10).

Листинг 7.10. Переменный список параметров-указателей (ноль в конце)

#include <iostream>

#include <stdarg.h>

using namespace std;

double f(double \*a, ...)

{

double \*\*p = &a; //берем адрес-адреса

double sum = 0, count = 0;

while (\*p != 0) //NULL – прямо в списке параметров

{

sum += (\*\*p); //выбираем значения

count++; p++; //бежим по списку

}

return ((sum) ? sum / count : 0);

}

void main()

{

double a = 5, b = 8;

double y = f(&a, &b, 0); //6.5

cout << y << endl;

system("pause");

}



Вызов такой функции выглядит так:

double y = f(&a, &b, 0);

Особо обратите внимание на следующее: в списке параметров 0 – это значение, а не адрес. Поэтому в теле функции проверка окончания цикла делается с одной звездочкой, а не с двумя.

#include <iostream>

using namespace std;

// приметивная функция нахождения сумы

// первий параметр количество елементов

int sum(int count, ...){

int s = 0;

for(int i = 1; i <= count; ++i){

s += (&count)[i];

}

return s;

}

int main(){

cout << sum(3, 20, 30, 40) << endl; //90

system("pause");

return 0;

}



Определение конца списка параметров с помощью параметра индикатора.

#include <iostream>

#include <stdlib.h>

using namespace std;

void MyFunc(int a,...) //Так объявляется функция с заранее неизвестным числом параметров

{

int \*P=&a; //Взятие адреса у первого параметра

while (\*P) //Пока встречаются параметры и параметр не равен 0

{

cout << \*P << " "; //Вытаскиваем значение с адреса по которому живет параметр

P++; //Адресная арифметика. Смена текущего адреса на следующий

}

}

int main()

{

system("CLS");

MyFunc(100,200,300,400,0); //В функцию передается 4 параметра и пятый как идентификатор того, что параметры кончились

cout << "\n";

system("PAUSE");

}



Функция **product()** вычисляет произведение дополнительных параметров. Как только встречается нулевой параметр, вычисление произведения прекращается. Указатель **p** указывает на начало списка параметров **(double\*p=&a;)**.

#include <iostream>

using namespace std;

double product(double a, ...)

{

double b;

if (a)//проверяем на отсутствие дополнительных параметров

{ b = 1.0; //b - произведение

//цикл до тех пор, пока p не укажет на 0

for (double\* p=&a; \*p; p++)

b \*= \*p;

}

else

b = 0.0;

return b;

}

void main()

{ double product(double a, ...);//прототип функции с

// переменным числом параметров

cout << "\n product (6e0, 5e0, 3e0, 0e0) = "

<< product (6e0, 5e0, 3e0, 0e0); //90

cout << "\n product (1.5f, 2.0f, 0.0f, 3.0f, 0.0f) = "

<< product (1.5f, 2.0f, 0.0f, 3.0f, 0.0f); //3

cout << "\n product (0.0) = " << product (0.0); //0

cout << "\n product (1.0e-28,0.0) = "

<< product (1.0e-28,0.0) << "\n";//1.0e-28

system("PAUSE");

}



**Напомним, что при вызове функции дополнительные параметры типа float передаются как double, так что для параметров типа float надо использовать функцию с параметрами типа double.**

Использование специального набора макроопределений.

Для использования в программе функций с переменным числом параметров, при вызове которых можно использовать параметры разных типов, удобно использовать специальный набор макроопределений (заголовочный файл **stdarg.h**). Макрокоманды для доступа к списку фактических параметров переменной длины, имеют следующий формат:

//связывание переменной param с первым параметром

**void va\_start(va\_list param,<последний явный параметр>);**

//получение значения очередного параметра типа type

**type va\_arg(va\_list param, type);**

//организации корректного выхода из функции

**void va\_end(va\_list param);**

Они объявлены в загаловке <stdarg.h> и их синтаксиз следующий

type va\_arg(va\_list argptr, type);

void va\_copy(va\_list target, va\_list source);

void va\_end(va\_list argptr);

void va\_start(va\_list argptr, last\_parm);

Макросы va\_arg, va\_start и va\_end работают вместе, чтобы сделать возможной передачу функции переменного числа аргументов. Самым распространенным примером функции, которая принимает переменное число аргументов, является функция printf(). Тип va\_list определен в заголовке <stdarg.h>(а в среде языка C++ — в заголовке <cstdarg>).

Кроме перечисленных макросов, в файле **stdarg.h** определен специальный тип данных **va\_list**, который является указателем. Именно такого типа должны быть первые операнды, используемые при обращении к макрокомандам **va\_start, va\_arg, va\_end**.

Определим порядок использования макросов. В теле функции обязательно определяется переменная типа **va\_list**: **va\_list f**. С помощью макроса **va\_start** переменная **f** связывается с первым необязательным параметром, т.е. с началом списка неизвестной длины:

**va\_start(f,<последний явный параметр>)**

Теперь с помощью указателя **f** можно получить значение фактического параметра, задав его тип. Макрос

**va\_arg(f, type);**

позволяет получить значение очередного параметра типа **type**. Кроме того, макрос меняет значение указателя **f** на адрес следующего фактического параметра в списке.

Макрокоманда **va\_end** предназначена для организации корректного выхода из функции с переменным числом параметров: **va\_end(f)**.

#include <iostream>

#include <stdarg.h>

using namespace std;

// приметивная функция нахождения сумы

// первий параметр количество елементов

int sum(int count,...)

{

va\_list ap;

int s = 0;

va\_start(ap,count); //связывание переменной param с первым параметром

for(int i=0; i<count; i++)

s+=va\_arg(ap,int); //получение значения очередного параметра типа type

va\_end(ap); //организации корректного выхода из функции

return s;

}

int main()

{

cout << sum(3, 20, 30, 40) << endl; //90

system("pause");

return 0;

}



Общая процедура создания функции, которая может принимать переменное количество аргументов, такова. Функция должна иметь по крайней мере один известный параметр(может и больше), указываемый до переменного списка параметров. Крайний справа известный параметр называется last\_parm. Его имя используется в качестве второго параметра в обращении к макросу va\_start(). Прежде чем можно будет получить доступ к любому параметру из списка параметров переменной длины, необходимо инициализировать указатель на аргумент argptr, обратившись к макросу va\_start(). После этого параметры возвращаются посредством обращения к макросу va\_arg() с передачей значения параметра type, которое представляет собой тип следующего параметра. Наконец после прочтения всех параметров до возвращения из функции необходимо вызвать макрос va\_end(), чтобы гарантировать корректное восстановление стека. Если макрос va\_end() не вызывается, высока вероятность аварийного отказа программы.

**Пример**

В следующей программе применяется функция sum\_series(), которая возвращает сумму ряда чисел. Первый аргумент содержит количество аргументов, перечисленных за ним. В этом примере суммируются первые пять элементов следующего ряда чисел.

1 1 1 1 1

- + - + - + — ... + ---

2 4 8 16 2n

Результат равен 0.968750.

#include <stdio.h>

#include <stdarg.h>

#include <iostream>

using namespace std;

double sum\_series(int, ...);

int main(void)

{

double d;

d = sum\_series(5, 0.5, 0.25, 0.125, 0.0625, 0.03125);

printf("Sum is: %f\n",d);

system("PAUSE");

return 0;

}

double sum\_series(int num, ...)

{

double sum = 0.0, t;

va\_list argptr;

/\* Инициализируем указатель argptr. \*/

va\_start(argptr, num);

/\* Суммируем ряд. \*/

for(; num; num--) {

t = va\_arg(argptr, double);

sum += t;

}

/\* Реализуем корректное завершение. \*/

va\_end(argptr);

return sum;

}



Вернемся к уже привычным для нас примерам, для демонстрации использования макроопределений.

В качестве примера рассмотрим реализацию функции вычисления среднего арифметического (вариант с количеством аргументов) с использованием этих макросов (листинг 7.12).

Листинг 7.12. Вычисление среднего с использование стандартных средств (количество)

#include <iostream>

#include <stdarg.h>

#include <string.h>

using namespace std;

double f(int n, double a, ...)

{ va\_list p; //объявление указателя

double sum = 0, count = 0;

va\_start(p, n); //инициализация указателя

while(n--)

{ sum+=va\_arg(p,double); //перемещение указателя

count++;

}

va\_end(p); //«закрытие» указателя

return ((sum)?sum/count:0);

}

void main()

{

double y = f(4, 1.0, 2.0, 3.0, 4.0);

cout << y << endl;

system("pause");

}



Листинг 7.13. Вычисление среднего стандартными средствами (ноль в конце)

#include <iostream>

#include <stdarg.h>

#include <string.h>

using namespace std;

double f(double a, ...)

{ va\_list p; //объявление указателя

double sum = 0, count = 0;

va\_start(p, a); //инициализация указателя

double k = a; //промежуточная переменная

do { sum+=k; count++;

} while(k=va\_arg(p,double)); //пока не ноль -передвигаемся

va\_end(p); //«закрыли» указатель

return ((sum)? sum/count: 0);

}

void main()

{

double y = f(1.0, 2.0, 3.0, 4.0, 0.0);

cout << y << endl;

system("pause");

}



Многоточие используется тогда, когда количество аргументов неизвестно. Неопределенное количество аргументов может быть указано в операторе функции с формальными параметрами. Например:

void yourfunction(int t,float u, ...) ;

Такая запись указывает компилятору С, что за обязательными переменными t и u могут следовать, а могут и не следовать другие аргументы. Конечно же, многоточие приостанавливает проверку типов.

В следующей программе на С иллюстрируется использование многоточия. Однако, вы можете воздержаться от детального изучения данного алгоритма до тех пор, пока полностью не познакомитесь с типами указателей на строки.

/\*08ELIP.C

Программа на С иллюстрирует использование операции многоточие

(...) и поддерживающих ее макросов va\_arg, va\_start и va\_end\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

// многим напоминает printf

void vsmallest(char \*szmessage, ...);

main()

{

vsmallest("Print %d integers, %d %d %d",10,4,1);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vsmallest(char \*szmessage, ...)

{

int inumber\_of\_percent\_ds=0;

va\_list type\_for\_ellipsis;

int ipercent\_d\_format = 'd';

char \*pchar;

// возвращает местоположение ‘d’ в формате %d

pchar=strchr(szmessage,ipercent\_d\_format);

// подсчет неопределенных количеств форматов %d

while(\*++pchar != '\0') {

pchar++;

pchar=strchr(pchar,ipercent\_d\_format);

inumber\_of\_percent\_ds++;

}

printf("print %d integers,",inumber\_of\_percent\_ds);

//////////////////////////////////

// стандартные макросы С

// va\_start

// va\_arg

// va\_end

//////////////////////////////////

// масрос

// устанавливает type\_for\_ellipsis на начало

// переменного списка аргументов

va\_start(type\_for\_ellipsis,szmessage);

while(inumber\_of\_percent\_ds--)

// макрос

// выбор следующего аргумента из списка переменных

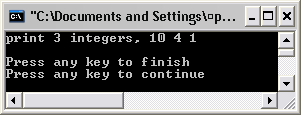
// int выбирает тип данных

printf(" %d",va\_arg(type\_for\_ellipsis, int));

// устанавливаент указатель в null

va\_end(type\_for\_ellipsis);

}



Прототип функции vsmallest() описан с двумя аргументами: указателем на строку и аргументом типа (...) — что составляет список аргументов переменной длины. Конечно, функции с переменным списком аргументов должны получать какую-то информацию о способе обработки этого списка. В программе 08ELIP.C эта информация поступает из строкового аргумента.

Строго говоря, функция vsmallest() похожа на функцию printf(). Эта подпрограмма просматривает строку форматирования szmessage и определяет, сколько форматов %d в ней содержится. Затем эта расчетная информация используется для выборки и печати информации из списка аргументов переменной длины. Хотя это выглядит очевидным, данный алгоритм требует достаточно сложных действий.

Функция strchr() возвращает местоположение символа "d" в формате %d. Первый формат %d можно проигнорировать, поскольку этого требует выходное сообщение. В цикле while продолжается обработка строки szmessage: просматривается и подсчитывается неопределенное количество форматов %d (inumber\_of\_percent\_ds). Когда этот процесс заканчивается, печатается начало выходного сообщения.

Макрос va\_start() устанавливает указатель type\_for\_ellipsis на начало переменного списка аргументов. Вспомогательный макрос va\_arg() выбирает следующий аргумент из списка переменных. Этот макрос использует свой второй параметр для определения выбираемого типа данных: в данном примере — это тип int. Функция vsmallest() заканчивается вызовом макроса va\_end(). Последний из трех стандартных макросов С для поддержки операции многоточие, va\_end(), устанавливает указатель в null.

Список указателей переменной длины на char (конкатинация строк)

**Напоследок осталось рассмотреть пример, в котором список указателей переменной длины составляют указатели на char. Мы выделяем этот случай по двум причинам:**

* размер данных (char) меньше, чем размер указателя (char\*) – в остальных случаях размер данных больше или равен размеру указателя;
* char\* — это единственный указатель, вместо которого при вызове можно задавать не адрес.

Типичной функцией, в которой можно применить переменный список параметров, является функция сцепления произвольного количества строк в одну. Заголовок такой функции может выглядеть так:

char \*f(char \*s1, ...)

Функция должна сначала вычислить количество памяти, необходимой для целевой строки, а потом уже помещать туда результат сцепления. Используем тот же прием, что и в предыдущем примере – последний параметр должен быть 0 (листинг 7.11).

Листинг 7.11. Сцепление строк (ноль в конце)

#include <iostream>

#include <stdarg.h>

#include <string.h>

using namespace std;

char \*f(char \*s1, ...)

{

char \*\*cp = &s1; //адрес первого указателя

int len = 0;

//цикл для определения общей длины сцепляемых строк

while (\*cp) { len += strlen(\*cp); cp++; }

char \*s = new char[len + 1]; //память для строки

s[0] = 0; //"очищаем" строку

//цикл для сцепления строк

cp = &s1; //опять установка на 1-й параметр

while (\*cp)

{

strcat(s, \*cp); //прицепляем первую (и следующие)

cp++; //перемещаемся на следующую

}

return s;

}

void main()

{

char \*sd = f("First ", "Two ", "Three ", 0);

cout << sd << endl;

system("pause");

}

Вызов такой функции может быть таким

char \*ss = f(s1, s2, s3, 0);

где s1, s2, s3 – это либо объявленные константы, либо переменные типа char \*. Ту же функцию можно вызывать и с явно прописанными константами:

char \*sd = f("First ", "Two ", "Three ", 0);



Очевидно, вместо параметра-указателя (ссылки) можно подставлять выражение, имеющее результатом указатель (ссылку). В частности, на месте указателя на char можно вызвать функцию, которая вводит строку с клавиатуры.

Приведем пример использования макросов в функции конкатенации любого числа строк. Строки передаются функции с помощью списка указателей. В конце списка помещается нулевой указатель **NULL**.

#include <stdio.h>

#include <conio.h>

#include <stdarg.h> //для макросов переменного списка параметров

#include <iostream>

using namespace std;

// функция конкатенации неопределенного числа строк

char \*concat(char \*s1, ...)

{

va\_list par; //указатель на параметры списка

char \*cp;

int len = strlen(s1); //длина 1-го параметра

va\_start(par,s1); //начало переменного списка

//цикл для определения общей длины параметров строк

while(cp = va\_arg(par, char \*))

len += strlen(cp);

//выделение памяти для результата

char \*stroka = new char[len + 1];

strcpy(stroka, s1);

va\_start(par, s1);//начало переменного списка

//цикл конкатенации параметров строк

while(cp = va\_arg(par, char \*))

strcat(stroka, cp);

va\_end(par);

return stroka;

}

void main()

{

setlocale(LC\_CTYPE, "Russian");

//прототип функции с переменным числом параметров

char \*concat(char \*s1, ...);

char \*s; //указатель для результата

s = concat("\nVerba"," volant, ","scripta ","manent\n",

"(Слова исчезают, написанное остается.)", NULL);

cout << s << "\n";

system("PAUSE");

}



В приведенной функции **concat()** тип параметров заранее известен и фиксирован (указатель на символы).

Изменение параметров по числу и по типу

В некоторых случаях параметры могут изменяться как по числу, так и по типу. В этих случаях необходимо, каким-то образом сообщать функции типы параметров для правильного их извлечения из стека. Например, в функциях **printf()** и **scanf()** в качестве первого операнда задается строка форматов, которая и содержит информацию о вводимых или выводимых данных.

Следующий пример, функция **miniprintf()**, показывает, как это можно сделать. В этой функции допускаются только два вида форматов **%d** и **%f**, которые позволяют макросу **va\_arg()** правильно извлекать значения. Окончание вывода определяется завершением перебора форматов.

#include <conio.h>

#include <stdarg.h> //для макросов переменного списка параметров

#include <iostream>

using namespace std;

void miniprintf(char \*format, ...)

{

va\_list ap; //указатель на необязательный параметр

char \*p; //для просмотра строки format

int ii; //целые параметры

double dd; //параметры типа double

va\_start(ap, format); //настроились на первый параметр

for(p = format; \*p; p++)

{ if(\*p != '%' ) //Если встретится символ %

{cout << \*p;

continue;

}

switch (\*++p) //То анализируем следующий за этим симолом символ

{ case 'd': ii = va\_arg(ap, int); //Если это символ d, то значит параметр int

cout << ii; //Выводим параметр типа int на экран

break;

case 'f': dd = va\_arg(ap, double); //Если это символ f значит параметр double

cout << dd; //Выводим параметр типа double на экран

break;

default: cout << \*p;

}

}//конец цикла просмотра строки форматов

va\_end(ap); //подготовка к завершению функции

}

void main()

{

void miniprintf(char \*format, ...);

int k = 123;

double d = 2.718282;

//Вывод двух чисел с разными типами

miniprintf("\n Integer k = %d \n Double d = %f", k, d);

cout << "\n";

system("PAUSE");

}



Дополнительные примеры функций с произвольным числом параметров

**НЕ РЕКОМЕНДУЕТСЯ. Может работать. А может работать неправильно**

#include <iostream>

#include <stdlib.h>

void MyFunc(int a, ...) //Так объявляется функция с заранее неизвестным числом параметров

{

}

int main()

{

system("CLS");

MyFunc(100, 200, 300, 400, 0); //В функцию передается 4 параметра и пятый будет использован как идентификатор того, что параметры кончились

system("PAUSE");

}

Вот такой код не вызывает возражений, а многоточие после первого параметра внутри параметров функции обозначает, что в функцию можно передать для обработки более одного параметра.

Должен возникнуть вопрос **как же получить доступ к каждому из этих параметров**. Для того, чтоб хорошо понять то что будет написано, требуется знание указателей. Чтобы иметь доступ к каждому из параметров, нужно знать адрес первого параметра, а чтобы взять этот адрес, как раз и требуется указатель

**НЕ РЕКОМЕНДУЕТСЯ. просто может работать. А может работать неправильно.**

#include <iostream>

#include <stdlib.h>

using namespace std;

void MyFunc(int a, ...) //Так объявляется функция с заранее неизвестным числом параметров

{

int \*P = &a; //Взятие адреса у первого параметра

while (\*P) //Пока встречаются параметры и параметр не равен 0

{

cout << \*P << " ";//Вытаскиваем значение с адреса по которому живет параметр

P++; //Адресная арифметика. Смена текущего адреса на следующий

}

cout << "\n";

}

int main()

{

system("CLS");

MyFunc(100, 200, 300, 400, 0); //В функцию передается 4 параметра и пятый как идентификатор того, что параметры кончились

system("PAUSE");

}



Что можно увидеть? Первым делом внутри функции происходят взятие адреса первого параметра, при этом тип указателя должен совпадать с типом этого параметра. Дальше идет простой перебор всех параметров с помощью цикла.

**Таким же образом можно привести классический пример суммирования элементов**

#include <iostream>

#include <stdlib.h>

using namespace std;

void MyFunc(double a, ...) //Так объявляется функция с заранее неизвестным числом параметров

{

double \*P = &a; //Взятие адреса у первого параметра

double sum = 0; //Инициализация значения суммы в ноль

while (\*P) //Пока встречаются параметры и параметр не равен 0

{

sum = sum + (\*P);//Прибавляем к сумме то что взяли по адресу P

P++; //Адресная арифметика. Смена текущего адреса на следующий

}

cout << sum << endl; //Вывод результата на экран

}

int main()

{

system("CLS");

MyFunc(100.0, 200.0, 300.0, 400.0, 0.0); //В функцию передается 4 параметра и пятый как идентификатор того, что параметры кончились

system("PAUSE");

}



Вообще весь этот механизм удобно использовать тогда, когда требуется обработка однотипных элементов и чем-то всё напоминает обычный массив элементов. Учитывая, что указатель должен быть того же типа, что и элемент, расположенный по адресу на который он ссылается, можно сказать, что переменное число параметров может быть корректно использовано как раз тогда, когда все параметры передаваемые в такую функцию принадлежат одному и тому же типу (имеются ввиду те параметры, которые идут туда где расположено многоточие).

Даже если типы передаваемых параметров будут различны, то компилятор не выдаст ошибки, но ошибки не выведется только потому что **если компилятор видит эти три точки на месте параметров, то он отключает проверку типов**. На самом деле легко увидеть эффект небезопасности работы с функцией, в которой указано, что будет более одного параметра, если первый параметр объявить например как **int**, (соответственно указатель на первый параметр как **int**), а во время передачи в функцию написать число с точкой (например **1.0** запустить и потом **11.0** и запустить). Т.е. если требуется передать параметры других типов, то их все нужно дописать перед списком предполагаемых неизвестных.

Например **void MyFunc(float x, char S[],int n,…)**, после чего с первыми работать в обычной манере, а с теми что после **n** с помощью указателя на **int** и, соответственно, передавать туда параметры с тем же типом, что и у этого **n** (Исходя из логики того, что тип указателя должен совпадать с типом объекта на адрес которого указатель ссылается).

## Аргументы функции main().

В обоих языках, С и C++, имеется возможность обработки аргументов командной строки, которые представляют собой параметры, вводимые вместе с именем программы при ее вызове из командной строки операционной системы. Эта возможность позволяет передавать аргументы непосредственно вашей программе без дополнительных запросов из этой программы. Например, пусть некоторая программа получает из командной строки четыре аргумента:

YOURPROGRAM Sneakers, Dumbdog, Shadow, Wonderdog

Здесь четыре значения передаются из командной строки программе YOURPROGRAM. На самом деле эта информация передается функции main(). Один аргумент, получаемый main(), argc, представляет собой целое число, определяющее количество элементов командной строки плюс 1.

Начиная с DOS 3.0, название программы считается первым элементом, передаваемым из командной строки. Второй аргумент — это указатель на массив указателей на строки, называемый argv. Так как все элементы являются символьными строками, argv имеет тип char \*[argc]. Поскольку все программы имеют название, argc всегда на единицу больше числа аргументов командной строки. В дальнейшем вы познакомитесь с различными способами извлечения разных типов данных из командной строки. Названия аргументов argc и argv являются общепринятыми именами переменных, используемых во всех программах на С и C++.

### Строки.

Аргументы командной строки передаются как символьные строки, что облегчает работу с ними. В следующем примере на С пользователь должен из командной строки задать программе несколько имен. Чтобы убедиться в том, что пользователь их ввел, анализируется аргумент argc и, если он не больше 2, происходит выход из программы с сообщением о повторной попытке.

/\*08SARGV.C

Программа на С иллюстрирует ввод в программу строковых данных

с аргументом командной строки

C:\> 08SARGV.EXE 07 Nik Helen\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

int main(int argc,char \*argv[])

{

int t;

if(argc<2)

{

/\* Введите несколько имен в командной строке \*/

/\* при запуске этой программы! \*/

printf("Enter several names on the commond line\n");

printf("when executing this program!\n");

/\* Попробуйте еще раз. \*/

printf("Please try again.\n");

exit(0);

}

for(t=1;t<argc;t++)

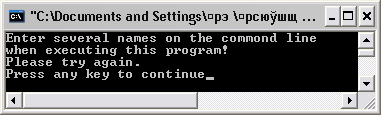
printf("Entry #%d is %s\n",t,argv[t]);

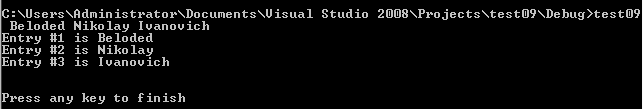
printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}





Эта программа целиком находится в функции main() и не требует дополнительных функций. Имена, введенные с командной строки, печатаются на экране в том же порядке. Если из командной строки вводятся цифровые значения, то они интерпретируются как ASCII-строки отдельных символов и должны печататься такими как есть.

### Целые числа.

Во многих программах желательно иметь возможность ввода из командной строки целых чисел; это может быть, к примеру, программа для вычисления средних оценок студентов. В таких случаях символьная информация в кодах ASCII должна быть преобразована в целые значения. В приведенном ниже примере на C++ из командной строки вводится одно целое число. Поскольку на самом деле это число является символьной строкой, оно преобразуется в целое при помощи библиотечной функции atoi(). Значение из командной строки ivalue передается в использованную в предыдущем примере функцию vbinary(), которая преобразует значение ivalue в строку двоичных цифр и печатает ее на экране. Когда управление возвращается функции main(), значение ivalue печатается в восьмеричном и шестнадцатеричном форматах.

// 08IARGV.CPP

// Программа на C++ иллюстрирует ввод в программу целых чисел

// из командной строки

// C:\> 08IARGV.EXE 97

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void vbinary(int idigits);

int main(int argc, char \*argv[])

{

int ivalue;

if(argc!=2)

{

// Введите десятичное число в командной строке.

// Оно будет преобразовано в двоичное, восьмеричное и

// шестнадцатеричное.

cout<<"Enter a decimal number on the command line.\n";

cout<<"It will be converted to binary, octal and hexadecimal.\n";

exit(1);

}

ivalue=atoi(argv[1]);

// Десятичное значение равно

cout << "The decimal value is: " << ivalue << endl;

vbinary(ivalue);

// Восьмеричное значение равно

cout << "The octal value is: " << oct << ivalue << endl;

// Шестнадцатеричное значение равно

cout << "The hexadecimal value is: " << hex << ivalue << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vbinary(int idigits)

{

int t=0;

int iyourarray[50];

while(idigits != 0)

{

iyourarray[t]=(idigits %2);

idigits/=2;

t++;

}

t--;

// Двоичное значение равно

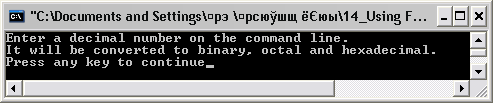
cout << "The binary value is: ";

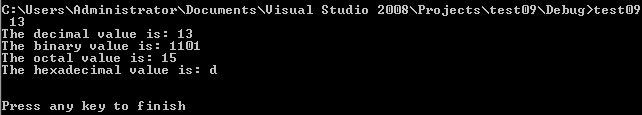
for(;t>=0;t--)

cout << dec << iyourarray[t];

cout << endl;

}





Особый интерес представляет форматирование различных чисел. Как вы помните, двоичные числа хранятся в массиве, а для вывода на экран содержимого массива iyourarray печатаются в обратном порядке по одной цифре за раз в десятичном формате:

cout << dec << myarray[i];

Для печати числа в восьмеричном формате используется следующий оператор:

cout << "The octal value is: "

<< oct << ivalue << endl;

Также можно распечатать шестнадцатеричный эквивалент, заменив oct на hex, что и показано ниже:

cout << "The hexadecimal value is: " << hex << ivalue << endl;

Если не применять дополнительного форматирования, то шестнадцатеричные цифры а, Ь, с, d, e и f печатаются строчными. Позднее вы познакомитесь со многими способами форматирования, в том числе узнаете, как печатать эти символы заглавными буквами.

### Числа с плавающей точкой.

Поскольку вы знаете, как вводить целые числа из командной строки, числа float не создают дополнительных проблем. Следующий пример на С позволяет ввести из командной строки значения нескольких углов. Вычисляются косинусы углов и печатаются на экране. Поскольку у значений углов тип float, они могут выглядеть по-разному, например: 12.0, 45.78, 0.12345 или 15.

/\*08FARGV.C

Программа на С иллюстрирует ввод в программу чисел

с плавающей точкой из командной строки

C:\> 08FARGV.EXE 90 45.78 0.12345 45\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

const double dPi=3.14159265359;

int main(int argc, char \*argv[])

{

int t;

double ddegree;

if(argc < 2)

{

/\* Введите несколько углов в командной строке. \*/

/\* Программа вычислит и напечатает \*/

/\* косинусы введенных углов. \*/

printf("Type several angels on the command line.\n");

printf("Programm will colculate and print\n");

printf("the cosine af the angles entered.\n");

exit(1);

}

for(t=1;t<argc;t++)

{

ddegree=(double) atof(argv[t]);

printf("The cosine of %f is %15.14lf\n",

ddegree, cos((dPi/180)\*ddegree));

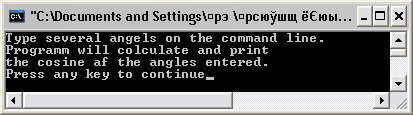
}

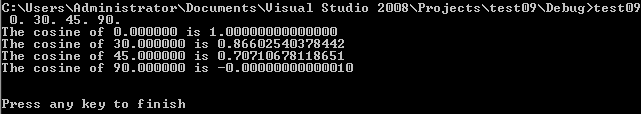
printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}





Функция atof() используется для преобразования строковых аргументов командной строки в тип float. В программе для вычисления косинуса в функции printf() используется функция cos().

## Области видимости. Локальные и глобальные переменные

Переменные могут быть объявлены как внутри тела какой-либо функции, так и за пределами любой из них.

Переменные, объявленные внутри тела функции, называются локальными. Такие переменные размещаются в стеке программы и действуют только внутри той функции, в которой объявлены. Как только управление возвращается вызывающей функции, память, отводимая под локальные переменные, освобождается.

Каждая переменная характеризуется областью действия, областью видимости и временем жизни.

Под областью действия переменной понимают область программы, в которой переменная доступна для использования.

С этим понятием тесно связано понятие области видимости переменной. Если переменная выходит из области действия, она становится невидимой. С другой стороны, переменная может находиться в области видимости, если к ней можно получить доступ (с помощью операции разрешения видимости, в том случае, если она непосредственно не видима).

Временем жизни переменной называется интервал выполнения программы, в течение которого она существует.

Локальные переменные имеют своей областью видимости функцию или блок, в течение которых она существует. В то же время область действия локальной переменной может исключать внутренний блок, если в нем объявлена переменная с тем же именем. Время жизни локальной переменной определяется временем выполнения блока или функции, в которой она объявлена.

Это означает, например, что в разных функциях могут использоваться переменные с одинаковыми именами совершенно независимо друг от друга.

В рассматриваемом ниже примере переменные с именем х определены сразу в двух функциях - в main () и в Sum (), что, однако, не мешает компилятору различать их между собой:

#include <iostream>

using namespace std;

// Прототип функции:

int Sum(int a, int b);

int main()

{

// Локальные переменные:

int x = 2;

int у = 4;

cout << Sum(x, у);

getchar(); getchar();

return 0;

}

int Sum(int a, int b)

{

// Локальная переменная х

// видна только в теле функции Sum()

int x = a + b;

return x;

}

В программе осуществляется вычисление суммы двух целочисленных переменных посредством вызова функции Sum().



**Глобальные переменные**, как указывалось ранее, объявляются вне тела какой-либо из функции и действуют на протяжении выполнения всей программы. **Такие переменные доступны в любой из функций программы, которая описана после объявления глобальной переменной. Отсюда следует вывод, что имена локальных и глобальных переменных не должны совпадать. Если глобальная переменная не проинициализирована явным образом, она инициализируется значением 0**.

Область действия глобальной переменной совпадает с областью видимости и простирается от точки ее описания до конца файла, в котором она объявлена. Время жизни глобальной переменной - постоянное, то есть совпадает с временем выполнения программы.

Вообще говоря, на практике программисты стараются избегать использования глобальных переменных и применяют их только в случае крайней необходимости, так как содержимое таких переменных может быть изменено внутри тела любой функции, что чревато серьезными ошибками при работе программы.

Рассмотрим пример, поясняющий вышесказанное:

#include <iostream>

using namespace std;

// Объявляем глобальную переменную Test:

int Test = 200;

void PrintTest(void);

int main()

{

setlocale(LC\_ALL, "Rus");

// Объявляем локальную переменную Test:

int Test = 10;

// Вызов функции печати

// глобальной переменной:

PrintTest ();

cout << "Локальная: "<< Test << '\n';

getchar(); getchar();

return 0;

}

void PrintTest(void)

{

cout << "Глобальная: " << Test << '\n';

}

Первоначально объявляется глобально переменная Test, которой присваивается значение 200. Далее объявляется локальная переменная с тем же именем Test, но со значением 10. Вызов переменной printTest() из main () фактически осуществляет временный выход и тела главной функции. При этом все локальные переменные становятся недоступны и PrintTest() выведет на печать глобальную переменную Test. После этого управление программой возвращается в функцию main(), где конструкцией cout<< выводится на печать локальная переменная Test. Результат работы программы выглядит следующим образом:

Глобальная: 200

Локальная: 10



**В C++ допускается объявлять локальную переменную не только в начале функции, а вообще в любом месте программы. Если объявление происходит внутри какого-либо блока, переменная с таким же именем, объявленная вне тела блока, "прячется".** Видоизменим предыдущий пример с тем, чтобы продемонстрировать процесс сокрытия локальной переменной:

#include <iostream>

using namespace std;

// Объявляем глобальную переменную Test:

int Test = 200;

void PrintTest(void);

int main()

{

setlocale(LC\_ALL, "Rus");

// Объявляем локальную переменную Test:

int Test = 10;

// Вызов функции печати

// глобальной переменной:

PrintTest();

cout << " Локальная: " << Test << '\n';

// Добавляем новый блок с еще одной

// локальной переменной Test:

{

int Test = 5;

cout << " Локальная: " << Test << '\n';

}

// Возвращаемся к локальной Test

// вне блока:

cout << " Локальная: " <<Test << '\n';

getchar(); getchar();

return 0;

}

void PrintTest(void)

{

cout << "Глобальная: " << Test << '\n';

}

Результат модифицированной программы будет выглядеть следующим образом:

Глобальная: 200

Локальная: 10

Локальная: 5

Локальная: 10



## Сложности в правилах области действия (scope rules).

**Если используются переменные с различной областью действия, то можно столкнуться с совершенно неожиданными результатами программирования, называемыми побочными эффектами.** Например, как вы уже знаете, может существовать переменная (вернее две переменные с одинаковым именем) как с файловой, так и с локальной областью действия. **Правила области действия констатируют, что переменная с локальной областью действия (называемая локальной переменной) имеет приоритет по сравнению с переменной с файловой областью действия (называемой глобальной переменной).** Хотя это выглядит достаточно просто, давайте рассмотрим несколько проблем, которые не так очевидны и с которыми вы можете столкнуться при программировании.

### Неопределенные символы в программе на С.

В следующем примере четыре переменные имеют локальную область действия внутри функции main(). Копии переменных il и im передаются в функцию iproduct(). Это не нарушает правила области действия. Однако, когда функция iproduct() пытается обратиться к переменной in, она ее не находит. Почему? Потому что область действия этой переменной локальна для функции main().

/\*08SCOPEP.C

Программа на С иллюстрирует проблемы, связанные с правилами

области действия. Предполагается, что функция вычисляет

произведение трех чисел. Компилятор выдает ошибку, поскольку

переменная in не известна функции, вычисляющей произведение\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

int iproduct(int iw,int ix);

/\* int in=10; \*/

main ()

{

int il=3;

int im=7;

int in=10;

int io;

io=iproduct(il,im);

/\* Произведение чисел равно \*/

printf("The product of the numbers is: %d\n", io);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

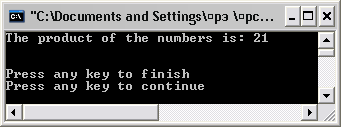
int iy;

iy=iw\*ix;

/\* iy=iw\*ix\*in; \*/

return (iy);

}



Компилятор С выдает предупреждение и сообщение об ошибке. Сначала он предупреждает о том, что переменная in, объявленная в функции main(), нигде там не используется, а затем выдает сообщение об ошибке, говорящее о том, что переменная in нигде не объявлена в функции iproduct(). Единственный способ избавиться от этой проблемы — дать переменной in глобальную область действия.

### Использование переменной с файловой областью действия.

В следующем примере переменной in определена файловая область действия. Глобальная видимость переменной in во всем файле позволяет ее использовать как функции main(), так и функции iproduct(). Также надо заметить, что обе функции, main() и iproduct(), могут изменять значение переменной. Существует хорошее программистское правило: если функции должны быть действительно переносимыми, не следует позволять им менять глобальные переменные.

/\*08FSCOPE.C

Программа на С иллюстрирует проблемы, связанные с правилами

области действия. Предполагается, что функция вычисляет

произведение трех чисел. Предыдущая проблема решена: третьей

переменной задана файловая область действия\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

int iproduct(int iw,int ix);

int in=10;

main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

printf("The product is: %d\n", io);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

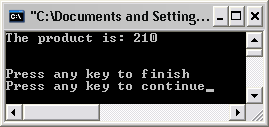
int iy;

iy=iw\*ix\*in;

return(iy);

}

Эта программа будет компилироваться без ошибок и напечатает на экране значение произведения 210.



### Приоритет переменных с файловой и локальной областями действия.

**Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение.** Это положение иллюстрирует следующая небольшая программа:

// 08SCOPEP.CPP

/\*Программа на С иллюстрирует проблемы, связанные с правилами

области действия. Предполагается, что функция вычисляет

произведение трех чисел, но каких чисел? Две переменные задаются

как параметры функции. Третья переменная имеет и файловую и

локальную область действия\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

int iproduct(int iw,int ix);

int in=10;

main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

printf("The product of the numbers is: %d\n", io);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

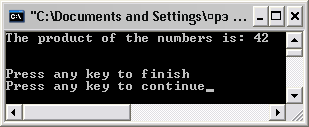
int iy;

int in=2;

iy=iw\*ix\*in;

return(iy);

}



В этом примере переменная in имеет одновременно файловую и локальную области действия. Когда переменная in используется в функции product(), локальная область действия имеет приоритет, и возвращаемый результат равен 3 \*7 \* 2 = 42.

### Проблемы области действия в C++.

Следующий пример на C++ работает нормально до момента вывода информации на экран. Оператор cout правильно распечатывает значения переменных il и im. Но при обращении к переменной in он выбирает глобальную переменную с файловой областью действия. Результат, выдаваемый программой, 3\*7\*10 = 42, является явной ошибкой. Как вы знаете, в подобной ситуации функция iproduct() использует локальное значение переменной in.

// 08SCOPEP.CPP

// Программа на C++ иллюстрирует проблемы, связанные с правилами

// области действия. Предполагается, что функция вычисляет

// произведение трех чисел. Переменная in имеет локальную область

// действия и используется в функции для вычисления произведения.

// Однако, в главной функции определено, что значение этой переменной

// равно 10. Что здесь неправильно?

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

int iproduct(int iw,int ix);

int in=10;

main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

cout << "The product of " << il <<" \* " << im << " \* " << in << " is: " << io << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

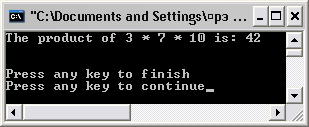
int iy;

int in=2;

iy=iw\*ix\*in;

return (iy);

}



Как решить эту проблему, если вы действительно хотите вычислить произведение и использовать глобальное значение переменной in? В C++ для этого можно использовать упомянутую выше операцию уточнения области действия и записать оператор следующим образом:

iy=iw\*ix\*::in;



### Операция уточнения области действия в C++.

Как было показано выше, объявление локальной переменной скрывает глобальную переменную с таким же именем. Таким образом, все обращения к имени глобальной переменной в пределах области действия локального объявления вызывают обращение к локальной переменной. Однако C++ позволяет обращаться к глобальной переменной из любого места программы с помощью использования операции разрешения области видимости. Для этого перед именем переменной ставится префикс в виде двойного двоеточия (::):

#include <iostream>

using namespace std;

// Объявление глобальной переменной

int Turn = 5;

int main ()

{

setlocale(LC\_ALL, "Rus");

// объявление локальной переменной

int Turn = 70;

// Вывод локального значения:

cout << Turn << '\n';

// Вывод глобального значения:

cout << ::Turn << '\n';

getchar(); getchar();

return 0;

}

В результате в две строки будет выведено два значения: 5 и 70.



Из рассмотренного примера видно, что были объявлены глобальная и локальная переменные с именем Turn, которые позже были выведены на печать.

В следующем примере для того, чтобы избежать конфликтов между переменными как с файловой, так и с локальной областями действия, используется операция уточнения области действия (scope resolution operator ::). Предыдущая программа выдавала неправильное значение произведения, так как в вычислениях использовалось локальное значение. Обратите внимание на то, как в приведенном листинге в функции iproduct() используется операция уточнения области действия.

// 08GSCOPE.CPP

// Программа на C++ иллюстрирует проблемы, связанные с правилами

// области действия, и использование операции уточнения области

// действия. При вычислении произведения с помощью этой операции,

// в функции используется переменная с файловой областью действия

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

int iproduct(int iw,int ix);

int in=10;

main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

cout << "The product of " << il <<" \* " << im

<< " \* " << in << " is: " << io;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

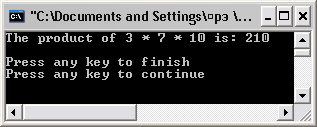
int iy;

int in=2;

iy=iw\*ix\*(::in);

return(iy);

}



Операцию уточнения области действия не обязательно заключать в круглые скобки — они использованы в этом примере только для выделения операции. Теперь при вычислениях используется глобальная переменная с файловой областью действия. При выводе результата на экран вы увидите значение 3 \* 7 \* 10 = 210.

Операция уточнения области действия очень важна в C++. Дополнительные примеры применения этой операции приводятся в последующем изложении материала.

## Математические функции

Прототипы стандартных математических функций определены в заголовочном файле math.h. Рассмотрим некоторые из них, наиболее часто употребляемые в повседневной работе.

Ранее уже упоминалась функция pow(), позволяющая возводить число в степень. Синтаксис данной функции выглядит следующим образом:

double pow(double x, double у);

Таким образом, компилятору сообщается, что необходимо число двойной точности х возвести в степень числа двойной точности у.

К данной категории также относятся логарифмические функции и функция извлечения корня числа:

double log(double); // натуральные

float logf(float);

long double logl(long double);

double log10(double); // десятичные

float log10(float);

long double log10 (long double);

double sqrt(double); // корень числа

float sqrtf(float);

long double sqrtl(long double);

Другая большая группа - функции получения абсолютной величины числа,

int abs(int); // целые

double f abs (double); // двойной точности

long labs (long); // длинные

float fabsf (float); // с плавающей точкой

long double fabsl (long double);// длинные

// двойной

// точности

воспринимают в качестве параметра аргумент некоторого типа (свой для каждой из функций) и возвращают его беззнаковую форму.

Для вычисления остатка от деления числа х на у используется функция fmod(), которая имеет следующий синтаксис:

double fmod(double x, double y);

Стандартная библиотека располагает широким набором тригонометрических функций и их модификаций для различных типов аргументов:

// Арккосинус:

double acos(double);

float acosf(float);

// Арксинус:

double asin(double);

float asinf(float);

// Арктангенс:

double atan(double);

float atanf(float);

// Арктангенс отношения у/х:

double atan2(double x, double y);

float atan2f(float, float);

// Косинус:

double cos(double);

float cosf(float);

// Гиперболический косинус:

double cosh(double);

float coshf(float);

// Синус:

double sin(double);

float sinf(float);

// Гиперболический синус:

double sinh(double);

float sinhf(float);

// Тангенс:

double tan(double);

float tanf(float);

// Гиперболический тангенс:

double tanh(double);

float tanhf(float);

Следует отметить, что углы тригонометрических функций указываются в радианах. Ниже приводится пример, осуществляющий перевод градусов в радианы и вывод значения синуса для введенного в градусах числа.

#include <math.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

double Angle;

double PI = 3.14159;

cout << "Введите угол в градусах: ";

cin >> Angle;

cout << "Значение синуса: ";

cout << sin(Angle \* PI / 180) << '\n';

getchar(); getchar();

return 0;

}



К сожалению, в C++ нет готовой реализации функции возведения аргумента в квадрат. Эту функцию можно реализовать, например, следующим образом:

inline double sqr(double x)

{return pow(x, 2);}

Заметим, что для работы данной функции необходимо задействовать заголовочный файл math.h, содержащий прототип функции pow().

## Указатель на функцию

### Указатели на функции

**Синтаксис**

Указатели на функции задаются следующим образом:

void f() { }

void (\*pf)() = &f;

pf();

В этом коде f - некоторая функция.

Во второй строчке определяется переменная pf, которая является *указателем на функцию, которая ничего не возвращает и не принимает ни одного аргумента*. В определении pf ей присваивается адрес функции f.

В третьей строке мы вызываем функцию по указателю pf(в данном случае будет вызвана функция f).

**Пример 1**

void swap(int & a, int & b) { }

void (\*pswap)(int &, int &) = &swap;

**Пример 2**

void f(int a) { }

void g(int b) { }

void (\*pf)(int) = &f;

pf(10); // Вызывается f(10)

pf = &g;

pf(20); // Вызывается g(20)

***Примечание***

Во время присваивания указателю на функцию адреса функции амперсанд(&) можно опустить. В этом случае будет выполнено автоматическое привидение имени функции к адресу этой функции.

**Пример 3**

void f() { }

void g() { }

void (\*pf) = &f; // Верно, &f - указатель на функцию f

pf = g; // Тоже верно, имя функции(g) автоматически приводится к указателю на функцию.

**Пример 4**

Пусть мы хотим реализовать таймер, который через заданные промежутки времени выполняет произвольный код. Пользователи нашего таймера должны иметь возможность указать какой именно код должен исполняться таймером. ООП-подход к решению данной задачи на C++ может выглядеть следующим образом:

struct TimerClient {

virtual void OnTimer() = 0;

};

Пользователи нашего таймера могут наследоваться от TimerClient и передавать нашему таймеру указатели на наследников. Наша реализация таймера будет вызывать OnTimer() и таким образом исполнять клиентский код.

Альтернативой ООП являются указатели на функции. Мы могли бы передать таймеру указатель на функцию, которую будет вызывать наша реализация.

### Указатели на методы

**Синтаксис**

Рассмотрим следующий класс:

struct Array2D

{

int height() const;

int width() const;

};

Методы height и width имеют одинаковую сигнатуру. Если у нас есть экземпляр класса Array2D, то мы можем использовать указатели на методы следующим образом:

int (Array2D::\*f)() = &Array2D::height;

// Определение указателя на метод объекта Array2D, который возвращает int

// и не принимает аргументов. (f)

Array2D a;

Array2D \* p = &a;

(a.\*f)(); // Вызывается a.height()

(p->\*f)();

f = &Array2D::width;

(a.\*f)(); //Вызывается a.width()

(p->\*f)();

***const функции в структуре (классе)***

Вначале о **const**, применяемом к функции структуры или класса.

Функция класса, объявленная **const**, трактует **this** как указатель на константу. Вообще тип **this** в методе класса **X** будет **X\***. Но если метод класса объявлена как **const**, то тип **this** будет **const X\***. В таких методах не может быть ничего присвоено переменным класса, которые не объявлены как **static** или как **mutable** . Также **const**-функции не могут возвращать не **const** ссылки и указатели на данные класса и не могут вызывать не **const-функции** класса. **const-функции** иногда называют инспекторами (**inspector**), а остальные мутаторами (**mutator).**

class **C**

{

public:

int **inspect()** const; // Эта функция обещает не менять \*this

int **mutate();** // Эта функция может менять \*this

};

В классе могут присутствовать две функции отличающиеся только **const**.

class **C**

{

**...**

public:

int **func ()** const;

int **func ();**

};

Не всякая функция может быть объявлена константной. **Конструкторы и деструкторы не могут быть объявлены как const. Также не бывает static const функций.**

class **C**

{

int **i;**

public:

staticint **func ()** const; //ошибка

};

**Замечание.**

Модификатор **const** после имени функции означает, что функция не может изменять члены-переменные того объекта, к которому она принадлежит. Более того, из этой функции можно вызывать только такие же константные функции. Зачем все это нужно - используя это, можно защитить самого себя от случайного изменения информации, которая не должна меняться

А теперь пример для указателя на метод класса (структуры)

struct MyClass

{

int method() const

{

return 10;

}

int method()

{

return 5;

}

}

int (MyClass::\*f)() const = &MyClass::method;

int (MyClass::\*g)() = &MyClass::method;

MyClass obj;

(obj.\*f)(); // Вернёт 10. Будет вызван константный метод

(obj.\*g)(); // Вернёт 5. Будет вызван неконстантный метод.

### Примеры

Пример 1. Простой указатель на функцию

#include <iostream>

#include <math.h>

using namespace std;

// площадь круга радиуса R

double area(double R)

{

const double PI = 3.1415;

return PI \* R \* R;

}

int main()

{

setlocale(LC\_ALL, "rus");

double r = 1.0;

cout << "для круга радиуса " << r << " площадь = " << area(r) << endl;

double(\*pfunc)(double); // переменная указатель на функцию

pfunc = area;

cout << "для круга радиуса " << r << " площадь = " << (\*pfunc)(r) << endl;

cin >> r;

return 0;

}



Пример 2. Массив указателей на функции

Предположим, для некоторого крупного проекта мы готовим последовательность тестов, выполняемых в процесс развития и роста самого тестируемого проекта (это так называемая технология разработки тестирования, и очень продуктивная). Число пошаговых тестов в такой ситуации будет постоянно нарастать по мере продвижения готовности базового проекта.

#include <iostream>

using namespace std;

void test1(void)

{

cout << "выполняется тест №1" << endl;

// ...

}

void test2(void)

{

cout << "выполняется тест №2" << endl;

// ...

}

void test3(void)

{

cout << "выполняется тест №3" << endl;

// ...

}

void(\*tests[])(void) = // массив указателей на функции

{

test1, test2, test3 // последовательность тестов

// ... и много других тестов

};

int main()

{

setlocale(LC\_ALL, "rus");

for (int i = 0; i < sizeof(tests) / sizeof(tests[0]); i++)

{

tests[i]();

}

int x;

cin >> x;

return 0;

}



В этом листинге **tests[ ]** – это [массив](http://purecodecpp.com/archives/1) **указателей на функции** без параметров и без возвращаемых значений. Красивое решение, не правда ли? Мы можем не задумываясь добавлять новые функции массива вызываемых тестов, и они все последовательно будут вызываться при выполнении.

Пример 3. Указатель на функцию – параметр функции

Мы можем пойти ещё дальше: если указатель на функцию представляет функцию в выражениях, можно передать указатель на функцию в качестве параметра другой, охватывающей функции, и последняя будет выполнять в своём теле переданную параметром функцию, не зная какое действие при этом выполняется.

Для иллюстрации всего сказанного создадим программу самого примитивного калькулятора, выполняющего арифметические действия над целочисленными операндами

#include <iostream>

#include <stdlib.h>

using namespace std;

int calculate(int op1, int op2, int(\*func)(int, int))

{

return func(op1, op2);

}

/\* сложение, вычитание, умножение, деление,

остаток, возведение в степень: \*/

int summ(int op1, int op2) { return op1 + op2; }

int diff(int op1, int op2) { return op1 - op2; }

int mult(int op1, int op2) { return op1 \* op2; }

int divd(int op1, int op2) { return op1 / op2; }

int bals(int op1, int op2) { return op1 % op2; }

int powr(int op1, int op2)

{

int ret = 1;

while (op2--) ret \*= op1;

return ret;

}

typedef int(\*fint\_t)(int, int); // fint\_t - указатель на функцию

// функции 6-ти арифметических операций

fint\_t foper[] = // массив указателей на функции

{

summ, diff, mult, divd, bals, powr

};

int main()

{

setlocale(LC\_ALL, "rus");

char coper[] = { '+', '-', '\*', '/', '%', '^' };

int noper = sizeof(coper) / sizeof(coper[0]);

do

{

char buf[120];

char \*str = buf;

char \*endptr;

char oper;

//flush сбрасывает буфер

**cout <<** "выражение для вычисления (<op1><знак><op2>): " **<< flush;**

cin >> buf;

int op1, op2;

//strtod - преобразование строки в

//значение с плавающей запятой двойной точности

op1 = strtod(str, &endptr);

oper = \*endptr++;

op2 = strtod(str = endptr, &endptr);

int i;

for (i = 0; i < noper; i++)

{

if (oper == coper[i]) {

cout << op1 << ' ' << oper << ' ' << op2 << " = "

<< calculate(op1, op2, foper[i]) << endl;

break;

}

}

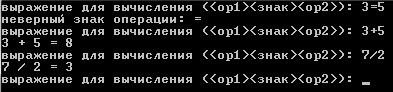
if (i == noper)

cout << "неверный знак операции: " << oper << endl;

} while (true);

return 0;

}



Здесь собственно вычисление выполняет функция **calculate()**. Но она ничего не «знает» о выполняемых действиях и о арифметических операциях вообще: она применяет к 2-м первым своим параметрам одну из 6-ти функций, которую ей передали 3-м параметром для выполнения.

В этом коде функция **strtod()** – стандартная функция библиотеки языка C (**ANSI C**, стандарта **POSIX**), которая извлекает десятичное число из строки, полученной со стандартного потока ввода. В контексте наших обсуждений это интересно тем, что:

* программа на C++ может использовать всё множество библиотечных вызовов языка C;
* программа C++ использует на этапе выполнения разделяемые библиотеки языка C (**.so** или **.dll**), и при отсутствии стандартной библиотеки C, программы на C++ становятся неработоспособными.

Наблюдательный пользователь мог бы заметить, что функция **calculate()** при всём своём желании и не могла бы выполнить ни одно из требуемых арифметических действий, так как выполняющие эти действия функции **sum()**, **dif()**, **mul()** и **div()** описаны **позже** функции **calculate()** и не видимы в коде функции **calculate()**.

Таким образом, мы рассмотрели несколько различных случаев, когда функции в C++ используются, как экземпляры данных. Они могут объединяться в массивы, встраиваться в качестве полей [структур](http://purecodecpp.com/archives/1439), или передаваться другим функциям в качестве параметров. Все эти и подробные возможности реализуются за счёт указателей на функции (хотя по синтаксису записей они могут и не выглядеть как указатели, за счёт умного компилятора C++, который по контексту понимает, что должны использоваться адреса функций).

Пример 4. Сортировка массива

Создадим массив, который в зависимости от выбора пользователя сортируется по убыванию или же возрастанию.

Чтобы сортировать какой-то массив, нужно этот массив создать и заполнить значениями, с этого и начинается написание программы по сортировке массива.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL,""); //Для русских символов

const int N=10; //Размер массива

int my\_choose=0; //В переводе звучит как Мой Выбор

int A[N]={9,8,7,6,1,2,3,5,4,9}; //Объявление массива и его инициализация значениями

cout << "1. Сортировать по возрастанию\n";

cout << "2. Сортировать по убыванию\n";

cin >> my\_choose; //Ожидание ввода пользователя

cout<<"Исходные данные: ";

return 0;

}

После написания такого кода возникает необходимость отображения массива на экране. Если подумать, то мы будем показывать исходный массив, будем показывать модифицированный. Одна операция будет выполнена как минимум дважды, а это плохо при написании всего в одну кучу, потому что увеличивает количество строк кода программы, следовательно увеличивает время написания программы и следовательно повышает вероятность ошибок и усложняет поиск ошибок. На таком маленьком коде — это едва заметно, но постепенно код будет расти и чем больше будет, тем это будет заметнее. У хорошей программы как и у хорошего здания должен быть крепкий фундамент, поэтому выносим операцию отображения массива на экране в отдельную функцию.

void show\_array(const int Arr[],const int N) //Функция покажет массив на экране

{

for (int i=0; i<N; i++)

cout<<Arr[i] << " "; //Поэлементно выводим значения массива на экран

cout << "\n";

}

И еще одно пояснение к простому примеру

#include <iostream>

using namespace std;

bool from\_min(int a,int b)

{

return a>b;

}

int main()

{

cout<<from\_min(14,5);//return => 1

cout<<"\n";

cout<<from\_min(5,14);//return => 0

cin.get();

cin.get();

return 0;

}

В этом коде мы не возвращаем явно указанного значения, но несмотря на это в разных случаях функция будет возвращать различные данные. Если условие после return для параметров функции справедливо, функция- вернет 1, иначе 0.

Для решения задачи используем алгоритм пузырьковой сортировки. Добавляем сразу три функции в основной код. Две из них — это функции, указатель для которых будет использован, а третья — пузырьковая сортировка.

#include <iostream>

using namespace std;

bool from\_min(const int a,const int b) //Проверка одного условия

{

return a>b;

}

bool from\_max(const int a,const int b) //Проверка другого условия

{

return a<b;

}

void bubble\_sort(int Arr[],const int N, bool (\*compare)(int a,int b)) //Функция пузырьковой сортировки

{

for (int i=1;i<N;i++){

for (int j=0;j<N-1;j++){

if ((\*compare)(Arr[j],Arr[j+1]))

swap(Arr[j],Arr[j+1]); //Сравниваем элемент со сразу за ним идущим

}

}

}

void show\_array(const int Arr[],const int N)

{

for (int i=0;i<N;i++)

cout<<Arr[i] << " ";

cout << "\n";

}

int main()

{

setlocale(LC\_ALL,"");

const int N=10;

int my\_choose=0;

int A[N]={9,8,7,6,1,2,3,5,4,9};

cout<<"1. Сортировать по возрастанию\n";

cout<<"2. Сортировать по убыванию\n";

cin>>my\_choose;

cout<<"Исходные данные: ";

show\_array(A,N);

cout<<"Сортированный массив: ";

switch (my\_choose)

{

case 1: bubble\_sort(A,N,from\_min); break;

case 2: bubble\_sort(A,N,from\_max); break;

default: cout<<"\rНеизвестная операция ";

}

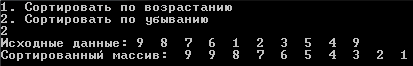
show\_array(A,N);

cin.get();

cin.get();

return 0;

}



**Замечание.**

* имя функции — начальный адрес её кода имеет приоритет ниже чем круглые скобки, поэтому указатель на функцию нужно заключать в скобки
* int \*func(int a,int b) — функция принимает два параметра на целое и возвращает указатель на целое
* int (\*func)(int a,int b) — указатель на функцию с двумя параметрами, каждый параметр — целое

http://ermak.cs.nstu.ru/cprog/html/093.htm

### Указатель на функцию и динамическое связывание

**Указатель на функцию** - переменная, которая содержит адрес некоторой функции. Соответственно, косвенное обращение по этому указателю представляет собой вызов функции.

Определение указателя на функцию имеет вид:

int (\*pf)(); // без контроля параметров вызова

int (\*pf)(void); // без параметров, с контролем по прототипу

int (\*pf)(int, char\*); // с контролем по прототипу

В соответствии с принципом контекстного определения типа данных эту конструкцию следует понимать так: pf - переменная, при косвенном обращении к которой получается функция с соответствующим прототипом, например int\_F(int, char\*), то есть pf содержит адрес функции или указатель на функцию. Следует обратить внимание на то, что в определении указателя присутствует прототип - указатель ссылается не на произвольную функцию, а только на одну из функций с заданной схемой формальных параметров и результата.

Перед началом работы с указателем его необходимо назначить на соответствующий объект, в данном случае - на функцию. В синтаксисе Си выражение вида **&имя\_функции** имеет смысл - начальный адрес функции или указатель на функцию. Кроме того, по аналогии с именем массива использование имени функции без скобок также интерпретируется как указатель на эту функцию. Указатель может быть инициализирован и при определении.

int INC(int а) { return a+1; }

extern int DEC(int);

int (\*pf)(int);

pf = &INC;

pf = INC; // присваивание указателя

int (\*pp)(int) = &DEC; // инициализация указателя

Естественно, что функция, на которую формируется указатель, должна быть известна транслятору - определена или объявлена как внешняя. Синтаксис вызова функции по указателю совпадает с синтаксисом ее определения.

n = (\*pf)(1) + (\*pp)(n); // эквивалентно

n = INC(1) + DEC(n);

Как и обычный указатель, указатель на функцию имеет размерность адреса, т.е. на уровне архитектуры является машинным словом, содержащим адрес функции в соответствии с системой формирования адресов процессором (системой адресации памяти). Это позволяет, в свою очередь, опускаться до таких машинно-зависимых действий как вызов функции по заданному адресу памяти, преобразуя целую константу к указателю на функцию.

void (\*pf)(void) = (void(\*)(void))0x1000; // Константа 1000 – шестнадцатеричный адрес

void main() { (\*pf)(); } // функции, которую вызывает main

// void(\*)(void) –абстрактный тип данных –

// указатель на функцию

Указатель на функцию прежде всего имеет отношение к системному программированию. Достаточно сказать, что кроме Си в явном виде он встречается только в Паскале (процедурный тип), но во многих случаях входит в скрытые механизмы управления программной средой, известные как **динамическое связывание функций**. Динамическое и статическое связывание относя и к переменным. По отношению к функции термин динамическое связывание следует понимать как установление соответствия между именем функции и ее адресом. Если рассматривать этот вопрос по отношению обычным функциям на Си, то можно выделить три этапа связывания:

* если определение функции и ее вызов находятся в одном модуле (файле), то транслятору ничего не мешает вычислить относительный адрес функций внутри объектного модуля во время трансляции – **статическое связывание**;
* при вызове внешней функции, определение которой находится в другом файле (а программный код – в другом объектном модуле), адрес функции становится известен при компоновке программного файла из объектных модулей. Хотя процедура компоновки выполняется после трансляции, данное связывание также называется **статическим**;
* набор внешних функций (объектный модуль целиком) может быть оформлен в виде динамически связываемой библиотеки – DLL (dynamic linking library). Она загружается в память одновременно с программным модулем и связывание имени внешней функции с ее адресом в адресном пространстве DLL происходит при загрузке. Это связывание уже считается **динамическим**. При наличии в каждом приложении отображаемого адресного пространства (виртуальной памяти) программный код DLL может одновременно использоваться (разделяться) несколькими приложениями.

Если на уровне среды программирования или непосредственно в Си-программах имеет место элемент динамического связывания, то с уверенностью можно сказать, что в том или ином виде (явно или неявно) используется указатель на функцию:

* уже упомянутые нами DLL, а также все возможные виды динамической загрузки внутреннего (исполняемого) программного кода;
* виртуальные функции в Си++ – при конструировании объекта производного класса в базовый класс помещается указатель на таблицу виртуальных функций для производного класса (массив указателей на функции);
* функции, работающие с произвольными типами данных – для настройки на конкретный тип данных в качестве «довеска» получают указатель на функцию, которая «знает», как с ним работать. В результате основной алгоритм изолируется от типа данных, с которым работает.

### Таблицы функций, вызов по имени

Тип данных вида **void (\*pp[])()** расшифровывается в соответствии с контекстным определением типа данных как массив указателей на функции с общим прототипом (схемой передачи параметров и результата) – последовательность операций в контексте – массив – указатель – вызов функции. Образно, хотя и не совсем точно этот тип можно назвать таблицей функций, вызов которых может производиться по номеру (индексу).

В самом простом примере наличие таблицы функций позволяет сделать вызов функции по заданному символьному имени регулярным (циклическим). В принципе, то же самое можно сделать с помощью обычного переключателя (switch), но для этого придется еа каждую новую функцию добавлять фрагмент программного кода. Здесь же достаточно внести в массивы символьное имя и ее адрес.

extern double sin(double);

extern double cos(double);

extern double tan(double);

char \*names[] = { "sin","cos","tan",NULL}; // Массив имен (указатели на строки)

double (\*pf[])(double) = { sin, cos, tan }; // Таблица функций (адреса функций)

//---------------------------------------------------------------------------------

//---- Вызов функции по имени из заданного списка

double call\_by\_name(char \*pn, double arg) {

for ( int i=0; names[i]!=NULL; i++)

if (strcmp(names[i],pn) == 0) { // Имя найдено -

return ((\*pf[i])(arg)); // вызов функции по i-му

} // указателю в массиве pf

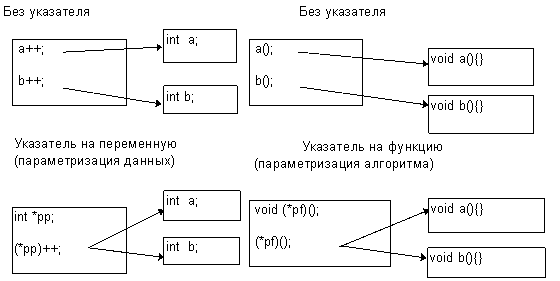
return 0.;

}

В принципе, используя технику работы с двоичными файлами и динамические массивы, можно было бы выполнить динамическую загрузку функций из программных файлов. Однако эта задача является архитектурно зависимой и требует учета особенностей перемещения программного кода и защиты памяти программы в операционной системе.

### Указатель на функцию как средство параметризации алгоритма

Оригинальность и обособленность указателя на функцию заключается в том, что указуемым объектом является не переменная (компонента данных программы), а функция (компонента алгоритма). Но сущность указателя при этом не меняется: если обычный указатель позволяет параметризовать алгоритм обработки данных, то указатель на функцию позволяет параметризовать сам алгоритм. Это значит, что некоторая его часть может быть заранее неизвестна (не определена, произвольна) и будет подключаться к основному алгоритму только в момент его выполнения (**динамическое связывание**).



Параметризация алгоритма

Для реализации указанного принципа основная функция должна получать необходимый для ее работы «довесок» в виде **формального параметра – указателя на функцию**. В качестве примера приведем функцию вычисления определенного интеграла, в которой подынтегральная функция (в математическом смысле) передается в виде вычисляющей ее функции (в «программистском» смысле), которая передается формальным параметром - указателем на функцию.

//------Численное интегрирование произвольной функции

double INTEG(double a, double b, int n, double(\*pf)(double))

// a,b - границы интегрирования, n - число точек

// pf - подынтегральная функция

{

double s,h,x;

for (s=0., x=a, h = (b-a)/n; x <=b; x+=h)

s += (\*pf)(x) \* h;

return s;

}

extern double sin(double);

void main()

{

printf("sin(0..pi/2)=%lf\n",INTEG(0.,M\_PI/2,40,sin));

}

### Сортировка одной и той же структуры данных по разным критериям

Другой частный пример подобного рода – сортировка одной и той же структуры данных по разным критериям. В массиве структур типа user для задания способа сравнения используется указатель на внешнюю функцию сравнения, которая в качестве параметров получает две ссылки на сравниваемые структуры.

//------Сортировка массива структур по разным критериям

struct user{

char name[20];

int account;

double time;

} S[]={…};

//-------- Вставка погружением

// cmp - указатель на функцию сравнения двух struct user

void sort(user A[], int (\*cmp)(user&,user&)){

for (int i=1;A[i].name[0]!=0;i++)

for(int j=i; j>0 && (\*cmp)(A[j],A[j-1])<0; j--) {

user c=A[j];

A[j]=A[j-1]; // Обмен элементов массива -

A[j-1]=c; // структурированных переменных

}

}

//--------- функция сравнения по имени

int cmp\_name(user &u1, user &u2){

return strcmp(u1.name,u2.name);

}

//--------- функция сравнения по account

int cmp\_account(user &u1, user &u2){

return u1.account - u2.account;

}

//--------- функция сравнения по time

int cmp\_time(user &u1, user &u2){

return u1.time - u2.time;

}

void main(){

sort(S,cmp\_name);

sort(S,cmp\_account);

sort(S,cmp\_time);

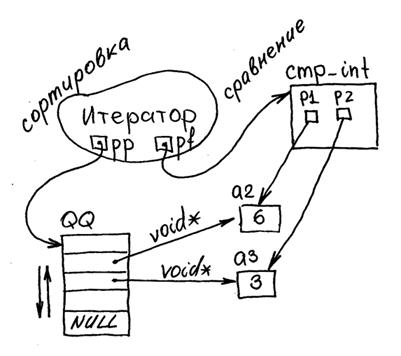
}

Этот пример можно обобщить на случай использования данных любого типа. В Си отсутствует понятие произвольный (неопределенный) тип, но понятие указатель на неопределенный (произвольный) тип существует – это void\*. Однако прямое использование этого указателя невозможно. Какой же здесь выход?

Если существуют несколько идентичных структур данных, отличающихся типом хранимых данных, то один и тот же алгоритм будет отличаться только отдельными операциями, касающимися переменных этого типа. Например, при сортировке массива указателей на строки сравнение реализуется выражением вида **strcmp(p[****i],****p[i-1])**, а в массиве указателей на целые используется непосредственное сравнение \*p[i]<\*p[i-1]. Если эту операцию вынести за пределы алгоритма, реализовать отдельной функцией, а указатель на нее передавать в качестве параметра, то мы получим универсальную функцию сортировки массивов указателей на переменные любого типа данных. Можно назвать ее условно (хотя и не совсем точно) **итератором**.

Типичными итераторами являются:

* итератор обхода (foreach), выполняющий для каждой переменной в структуре данных указанную функцию;
* итераторы проверки и поиска по условию (firstthat), возвращающие указатель на первую (последнюю, очередную) переменную, которая удовлетворяет условию, проверяемому в функции;
* итераторы, работающие на упорядоченных данных, использующие сравнение: сортировка, поиск минимального, двоичный поиск, включение с сохранением порядка.



Структура итератора

**Итератор** являет собой пример универсализации алгоритма, основанный на разделении «компетенций» между различенными его частями:

* структура данных, обрабатываемая итератором, содержит в своих элементах указатели на переменные произвольного (неизвестного для итератора) типа **void\***, но одинакового в каждом экземпляре структуры данных. Структура данных «знает», где хранятся переменные, но «не знает», как их обработать;
* итератор получает в качестве параметров указатель на структуру данных и указатель на функцию обработки. Эта функция «знает», как обработать переменные, но «не знает», где их взять;
* итератор выполняет алгоритм обработки структуры данных в соответствии со своим назначением: **foreach** - обходит все переменные, **firstthat** - обходит и проверяет все переменные, итератор сортировки -сортирует указатели на хранимые объекты (или соответствующие элементы структуры данных, например, элементы списка);
* действие, которое надлежит выполнить над хранимыми объектами произвольного типа, реализуется вызовом внешней функции через указатель. Итератор извлекает из структуры данных указатели void\* на хранимые данные и передает их внешней функции.
* прототип (схема формальных параметров и результата) внешней функции различна для разных типов итераторов. Для **foreach** – указатель имеет вид **void (\*pf)(void\*)**, для **firstthat** – **int (\*pf)(void\*)**, для итераторов, использующих сравнение – **int (\*cmp)(void\*,****void\*)**.

//------------------------------------------------------

//----- Итераторы foreach, firstthat и поиска минимального для списка

struct list {

list \*next; // Указатель на следующий

void \*pdata; }; // Указатель на данные

//----- Итератор: для каждого элемента списка

void ForEach(list \*pv, void (\*pf)(void\*) ) {

for (; pv !=NULL; pv = pv->next)

(\*pf)(pv->pdata);

}

//----- Итератор: поиск первого в списке по условию

void \*FirstThat(list \*pv, int (\*pf)(void\*)) {

for (; pv !=NULL; pv = pv->next)

if ((\*pf)(pv->pdata)) return pv ->pdata;

return NULL;

}

//----- Итератор: поиск минимального в списке

void \*FindMin(list \*pv, int (\*pf)(void\* ,void\*))

{

list \*pmin;

for ( pmin=pv; pv !=NULL; pv = pv->next)

if ((\*pf)(pv->pdata ,pmin->pdata) <0) pmin=pv;

return pmin;

}

//----- Примеры использования итератора ------------------

//----- Функция вывода строки

void print(void \*p) { puts((char\*)p); }

//----- Функция проверки : длины строки >5

int bigstr(void \*p) { return strlen((char\*)p ) > 5; }

//----- Функция сравнения строк по длине

int scmp(void \*p1, void \*p2)

{

return strlen((char\*)p1)- strlen((char\*)p2);

}

//----- Вызов итераторов для статического списка,

// содержащего указатели на строки

list a1={NULL,"aaaa"}, a2={&a1,"bbbbbb"}, a3={&a2,"ccccc"}, \*PH=&a3;

//----- Итератор сортировки для массива указателей

void Sort(void \*\*pp, int (\*pf)(void\*,void\*))

{

int i,k;

do{

for (k=0,i=1; pp[i] !=NULL; i++)

if ( (\*pf)(pp[i-1],pp[i])>=0) // вызов функции сравнения

{

void \*q; // перестановка указателей

k++;

q = pp[i-1];

pp[i-1] = pp[i];

pp[i] = q;

}

}while(k);

}

// Пример вызова итератора сортировки для массива

// указателей на целые переменные

int cmp\_int(void \*p1, void \*p2)

{ return \*(int\*)p1-\*(int\*)p2; }

int b1=5, b2=6, b3=3, b4=2;

void \*PP[] = {&b1, &b2, &b3, &b4, NULL};

void main()

{

char \*pp;

ForEach(PH,print);

pp = (char\*) FirstThat(PH,bigstr);

if (pp !=NULL) puts(pp);

pp = (char\*) FindMin(PH,scmp);

if (pp !=NULL) puts(pp);

Sort(PP,cmp\_int);

for (int i=0; PP[i]!=NULL;i++)

printf("%d ",\*(int\*)PP[i]);

puts("");

}

## Массив указателей на функции

Трудно понимать указатели, но указатели понимать нужно. Сам по себе указатель вызывает массу непониманий и массу вопросов, а тут еще массив зачем-то с функциями к нему.

Массив указателей на функции можно использовать в разных целях. Один из вариантов такого использования — это если, к примеру, у вас в коде есть одна куча разных функций, но схожих по смыслу и другая куча разных функций, не похожих на первые функции, но похожих друг на друга. Вот можно похожие функции собрать в массив и потом уже вызывать из массива. Не возьмусь утверждать, но скорее всего так будет проще не запутаться.

#include<iostream>

usingnamespace **std;**

void **a() {cout<<**"This is A";} //функция а

void **b() {cout<<**"This is B";} //функция b

void **c() {cout<<**"This is C";} //функция с

int **main()**

{

void **(\*P[3])()={a,b,c};** //Массив из 3-х указателей на функции

**(\*P[2])();** //Обращение к элементу массива вызывает функцию

//cin.get();

**cout<<**"\n";

**system(**"PAUSE");

return **0;**

}



Чтобы этот материал лучше окреп в сознании, возьмем еще 1 пример. В этом примере программа будет вести себя в зависимости от того, что выбрал пользователь. Принципиально этот пример вообще ничем не отличается от вышеприведенного, но может помочь понять то, что не получилось понять с первого раза.

#include <iostream>

using namespace std;

void a(int a) {cout<<"Функция a";}

void b(int b) {cout<<"Функция b";}

void c(int c) {cout<<"Функция с";}

int main()

{

setlocale(LC\_ALL,"");

int my\_choose=0;

void (\*P[3])(int)={a,b,c};

cout<<"Какую из 3 функций выбрать? ";

cin>>my\_choose;

my\_choose--; //После считывания уменьшаем число на 1. Чтобы отсчет был как от единицы

(\*P[my\_choose])(my\_choose); //Вызываем функцию, обращаясь к индексу массива

cin.get();

cin.get();

return 0;

}



**Замечание.**

Если **(\*P[my\_choose])(my\_choose);**

**P[my\_choose]** — выделяет указатель, расположенный в элементе массива с индексом **my\_choose**

**\*** — разыменовывает указатель, чтобы вызвать функцию

**(my\_choose)** — my\_choose передается в функцию как аргумент

## Шаблоны функций в С++. Основные понятия

http://ci-plus-plus-snachala.ru/?p=55

#include <iostream>

#include <stdlib.h>

using namespace std;

template <class T> //Шаблон функции с одним параметром T

T MyFunc(T a) //Тип функции MyFunc определяется шаблоном. В функцию принимается один параметр a, тип которого также определяется шаблоном

{

return a; //Что приняли, то и вернули

}

void main()

{

system("CLS");

cout<<MyFunc(10)<<endl; //MyFunc возвращает тип int

cout<<MyFunc(10.222)<<endl; //MyFunc возвращает тип double

cout<<MyFunc("HELLO")<<endl; //возвращает тип массив char

system("PAUSE");

}



**Ключевое слово template обозначает Шаблон.**

**template <class** **T> обозначает Шаблон функции с одним параметром** **T**

Несложно заметить большое сходство с написанием прототипа обычной функции.

После написания **template** и указания в угловых скобках всех параметров (В приведенном примере один параметр **T**) в примере написана функция **MyFunc**. Вместо явного указания типа, тип для функции и тип для параметра этой функции был указан как тип по шаблону. Для такого указания типа используются имена параметров, указанных в угловых скобках шаблона.

Проще говоря — этот описанный тип **T** можно изменить на привычный тип (**int**, **double** или другой).

Сейчас тот этап когда нужно все четко расставить на свои места и понять, что вместо указания явного типа для описываемой функции были использованы возможности шаблонов С++. Можно сказать, что шаблон функции вытеснил описание типов и встал на их место.

Дальше проще. Функция **MyFunc** возвращает некоторое значение. Для того, чтоб было максимально просто понять материал,- что функция принимает, то и возвращает, используется только один параметр. Так как функция возвращает один параметр, то и в шаблоне для этой функции должен быть указан один параметр. Если в шаблоне указан один параметр — то и внутри функции по этому шаблону нужен один, если два — то два и т.д.

Теперь остается функция **main**. Внутри функции **main** три раза происходит вызов функции **MyFunc** и каждый раз в нее передается и в ней принимается один параметр. При этом каждый раз этот параметр воспринимается компилятором как параметр некоторого типа, причем типы различны. (Это указано в комментариях кода). Тут как раз немного отображается смысл использования шаблонов. Без шаблонов нужно писать отдельные функции, которым нужно явно задать тип, который они возвращают и в нашем случае — это три отдельные функции с уникальными именами, выполняющие одно и то же с единственным отличием — разница результирующего типа.

**Пример.**

*Используя шаблоны функций сравнить два числа и вернуть большее значение*

**!!! Этот пример в С++ приводит к ошибке.**

#include <iostream>

#include <stdlib.h>

using namespace std;

template <class T>

T max(T a, T b)

{

if (a>b)

return a;

else

return b;

}

void main()

{

system("CLS");

cout << max(12,100)<<endl;

cout << max(12.33,3.0)<<endl;

//cout<<max(12.33,3)<<endl; //ОШИБКА

system("PAUSE");

}

Небольшой код, в функцию **max** передается два параметра, тип которых при приеме внутри функции определяется шаблоном. В самом шаблоне определяемый тип указан как один параметр. В коде указан момент, который может привести к ошибке. Просто если в самом шаблоне указан один параметр, то такая запись интерпретируется компилятором как *«Сколько бы функция не принимала параметров вовнутрь себя, возложив на меня обязанности указания типов, все эти параметры я расценю как однотипные»* и, соответственно работайте с ними как с однотипными и не выпендривайтесь.

Решение проблемы в примере. Если нужно, чтоб шаблон обрабатывал параметры как параметры с разными типами, значит нужно внутри шаблона прописать более двух параметров. Это делается через запятую

#include <iostream>

#include <stdlib.h>

using namespace std;

template <class T1, class T2> //Шаблон функции с двумя параметрами

T1 max(T1 a, T2 b) //Функция max, тип функции равен типу первого принимаемого параметра

{

if (a > b) return a;

else return b;

}

void main()

{

system("CLS");

cout << max(12, 100)<< endl; //Возвращет тип int и равно 100

cout << max(120.222, 100)<< endl; //Возвращает тип double и равно120.222

cout << max(10, 55.5555)<< endl; //Возвращает тип int и равно 55

system("PAUSE");

}

**Обратите внимание, что теперь можно выполнить сравнение двух разных типов данных.**



В этом коде у невнимательного читателя может возникнуть вопрос: «А почему функция возвращает не **55.5555**, а 55» — Ответ прост. Шаблоны функций очень похожи на прототипы функций. В качестве первого параметра вовнутрь функции передается целочисленное значение **10**. Это **10** обрабатывается компилятором с помощью шаблона. В шаблоне для типа первого параметра функции определен параметр **Т1** (Он первый внутри угловых скобок). В добавок к этому сама функция **max** имеет тип, равный типу **Т1**, поэтому функция возвращает тот тип, который компилятор определяет из первого входного параметра вовнутрь функции.

Изъясняясь короче

* + Из функции **main** вызывается функция **max**. В функцию передается **2** параметра
  + Функция **max** принимает два переданных в нее параметра
  + В функции **max** для обработки данных происходит некоторый анализ при помощи шаблона
  + В шаблоне указано, что функция имеет два различных параметра, компилятор говорит понял, разберусь
  + Компилятор обработал данные с помощью шаблона и выбрал подходящий тип для каждого из параметров
  + Код написан таким образом, что тип функции **max** равен типу первого принимаемого параметра

Ко всему этому написанному материалу можно добавить, что можно использовать различного рода локальные переменные, тип которых будет определяться компилятором при помощи шаблона. (по аналогии с примером типа функции)

#include<iostream>

#include<stdlib.h>

usingnamespace **std;**

template **<**class **T>**

void **Show(T a)**

{

**T n;** //Переменная n. Тип переменной равен типу параметра а и определяется шаблоном

**n = a;** //Присвоение в n значения a

**cout << n << endl;** //Что приняли, то и вывели на экран

}

void **main()**

{

**system(**"CLS");

**Show(10);**

**Show(20.999);**

**Show(**"HELLO");

**system(**"PAUSE");

}



По факту это то же, что и первый приведенный код. Просто кому-то может понадобиться использовать шаблон таким образом где внутри функции определяется какая-то локальная переменная, тип которой определяется с помощью шаблона или какие-то другие причины.

**Замечание.**

* Шаблоны функции позволяют вам объявлять типонезависимые или общие функции
* Если вы используете функцией только один какой-то тип, нет необходимости применять шаблон
* Если параметрам функции требуется несколько различных типов, то шаблон назначает каждому типу уникальный идентификатор, который компилятор в процессе компиляции обработает как переменную нужного типа

## Параметры шаблонов функций.

http://ci-plus-plus-snachala.ru/?p=3965

Проблемным моментом в шаблонной функции предыдущего раздела был тип возвращаемого значения. Это просто логическое продолжение того материала.

Для начала листинг кода такой вот несложной функции

#include <iostream>

using namespace std;

template <typename T1, typename T2> //Шаблон с двумя аргументами

inline T1 max(T1 const& a, T2 const &b){ //Функция с шаблонными аргументами

if (a>b) return a;

else return b;

}

int main()

{

cout << max(10, 66.999); //Вызов функции

cout << endl;

system("PAUSE");

}



Здесь, в принципе, ничего сложного нет. Если знакомы с функциями и передачей в них параметров, то все должно быть понятно. Если еще плохо знаете, то, наверное, рановато сюда попали. В **main** происходит вызов функции **max**, в которую передаем 2 значения, причем типы этих значений различны (**int** и **double**). Функция принимает эти параметры, как **а** и **b**. Принимает она их по ссылке, благодаря чему не создается нутренних локальных копий этих переменных. **А указанием const, мы обозначаем, что изменять мы эти значения не намерены.** Типы принимаемых параметров, мы оставляем на потом. Т.е. когда нам надо будет вызвать эту функцию, тогда мы и скажем, что подставить в **T1** и что подставить в **T2**.

Этот код имеет 2 недостатка. Первый, наиболее очевидный, в том, что возвращаемое значение из функции **max**, может принимать оба типа. Какой она тип вернет зависит от порядка передаваемых в нее аргументов.

#include <iostream>

using namespace std;

template <typename T1, typename T2>

inline T1 max(T1 const& a, T2 const &b){

if ( a > b ) return a;

else return b;

}

int main(){

const double N1 = 66.999;

const int N2 = 54;

std::cout << (max)(N1,N2) << "\t| " << sizeof(max)(N1,N2) << "\n"; //Вернулся double

std::cout << (max)(N2,N1) << "\t| " << sizeof(max)(N2,N1) << "\n"; //Вернулся int, после точки самоубилось

system("PAUSE");

}



Думать о том, в каком порядке передавать аргументе — дело не благодарное. А в ином случае вот такие вот фокусы происходят.

Другая проблема не столь очевидная в том, что **Т2** может неявно приводиться к **Т1**. Если немного подумать, можно легко понять, что функция возвращает тип **Т1**. А тип **Т2** может не совпадать с типом **Т1**. Что если переменная типа **Т2** больше чем переменная типа **Т1** ? Возвращать надо **Т2**, но так типы не совпадают, происходит приведение типа из **Т2** в **Т1**. Для этого приведения внутри функции создается внутренняя локальная переменная. Живет она внутри функции и умирает вместе с завершением работы функции. В свою очередь из-за этого нельзя вернуть результат по ссылке.

#include <iostream>

using namespace std;

template <typename T1, typename T2>

inline T1 const& max(T1 const& a, T2 const &b){

if (a > b) return a;

else return b;

}

int main(){

const double N1 = 66.999; //

const int N2 = 54;

std::cout << (max)(N2,N1) << "\n"; //T1 == int (54), T2 == double(66.999), T2>T1 --> Приведение типов --> Сломалось.

std::cout << (max)(N1,N2) << "\n"; //T1 == double(66.999), T2 == int(54), T1>T2 --> T1 приводить к T1 бессмысленно, нет приведения, усё ОК.

system("PAUSE");

}



Такие вот недостатки у такого очевидного способа.

Один из вариантов обхода такого неудобства — это задавать дополнительный параметр вывода в шаблон.

#include <iostream>

using namespace std;

template <typename T1, typename T2, typename RT> //Добавили третий параметр

inline RT const& max(T1 const& a, T2 const &b){ //Указали, что тип возвращаемого значения - тип третьего параметра

if (a > b) return a;

else return b;

}

int main(){

const double N1 = 66.999; //

const int N2 = 54;

//Явно указали типы для шаблонной функции

std::cout << max<int,double,double>(N2,N1) << "\n";

std::cout << max<double,int,double>(N1,N2) << "\n";

system("PAUSE");

}



Такой вот способ решения в лоб. Но здесь опять же имеет значение порядок передаваемых типов. Шаг влево, шаг вправо и програма ломается. Немного невнимательности и ищи, свищи проблему.

Просто поменяйте местами типы в угловых скобках.

В общем так делать можно, но это требует особой внимательности и делать так достаточно утомительно.

Такая проблема тоже решаема. Решается она достаточно просто. Достаточно изменить порядок параметров, принятых шаблоном. И все будет намного удобнее.

#include <iostream>

using namespace std;

template <typename RT, typename T1, typename T2> //RT в начало

inline RT const& max(T1 const& a, T2 const &b){ //Возвращаем тип RT

if (a>b) return a;

else return b;

}

int main(){

const double N1 = 66.999; //

const int N2 = 54;

std::cout << max<double>(N2, N1) << "\n"; //В угловых скобках указываем тип возвращаемого значения

std::cout << max<double>(N1, N2) << "\n";

system("PAUSE");

}



Теперь все заработало. В угловых скобках мы явно указали тип данных, который ожидаем от функции, этот тип данных пришел в **RT**. Остальные типы пришли в **T1** и в **T2**. Они туда ушли молча. Ушли в том порядке, в котором мы отдаем их вовнутрь функции. Так в первом случае (первой строчке) в **T1** ушел тип из переменной **N2** (**int**), в **T2** тип **N1** (**double**), а во втором случае **T1** из **N1**(**double**),**T2** из **N2**(**int**).

Согласитесь, что это намного удобнее чем думать о порядке типов и изменять весь этот порядок при необходимости.

**Замечание.**

* Следует явно задавать все типы, которые нельзя определить неявно
* В угловых скобках перед именем шаблонной функции указываются параметры шаблона, в круглых скобках после имени шаблонной функции указываются параметры вызова

Это все касается шаблонов функций. В шаблонах классов, например, можно использовать аргументы шаблона по умолчанию.

## Шаблоны функций. Аргументы по умолчанию

http://ci-plus-plus-snachala.ru/?p=4019

Этот материал показывает маленький фокус с обработкой матриц. До С++11 обработка двумерных и более мерных матриц внутри функций требовала того, чтобы программисты С++ явно указывали саму матрицу и ее размеры. Либо использования двумерного массива как одномерного, либо другого извращенного варианта. Язык С++ развивается и дополняется различными возможностями. С одной стороны это плохо, с другой удобства — это удобства. Поэтому простой прием обработки двумерного статически создаваемого массива внутри функции.

Очень простая задача. Показать двумерный массив на экране при помощи функции.

Вот наш массив

int m[3][3] {{ 1, 2, 3}, {4, 5, 6}, { 8, 9, 10}};

Прошу заметить, что **это не массив \*\*, а массив [][],** т.е. статически создаваемый. Вот этот массив мы и будем выводить на экран. Начнем с самого простого способа, используя возможности С++11.

Благодаря тому, что количество строк и столбцов в этом массиве одинаковое, можно обработать его с помощью функции очень даже простым способом.

#include <iostream>

using namespace std;

template <const unsigned size = 0> //Аргумент шаблона по умолчанию, можно int size=0, но const unsigned посолиднее и правильнее.

void foo(const int m[][size]) // Массив сам определит размер в пустых скобках, это как при объявлении массива int Arr[]={1,2,3}; --> Arr знает, что у него 3 элемента

{

for (int i = 0; i<size; i++){

for (int j = 0; j<size; j++){

cout << m[i][j] << "\t";

}

cout << "\n";

}

}

int main() {

int m[3][3] {{ 1, 2, 3}, { 4, 5, 6 }, { 8, 9, 10 }}; //Инициализация массива значениями

foo(m); //Вызов функции, передаем в функцию массив

system("PAUSE");

}



Из-за того, что размеры строк и колонок одинаковые, нам не нужно дополнительно узнавать сколько там у массива получилось в пустых скобках. Мы ведь итак знаем, что у него число строк и колонок одинаковое. Удивительно, но **этот код прекрасно работает благодаря тому, что мы использовали шаблон, в котором указали аргумент по умолчанию**.

Но что делать, **если массив не такой "ровный", а прямоугольный**, где число строк и колонок разнится. То, что массив способен распознать часть своего размера не обозначает, что это сможем сделать мы, да и этот показанный прием даст сбой. **Вот пример такого сбоя**.

#include <iostream>

using namespace std;

template <const unsigned size1 = 0, const unsigned size2 = 0>

void foo(const int m[][size1])

{

for (int i = 0; i<size1; i++){

for (int j = 0; j<size2; j++){

cout << m[i][j] << "\t";

} cout << "\n";

}

}

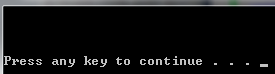
int main() {

int m[3][4] {{ 1, 2, 3, 4}, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }};

foo(m);

system("PAUSE");

}



И на экране нет массива! Сбой, прием не работает. **Это не сбой, это наша (в текущей ситуации моя) ошибка**. Достаточно посмотреть, что получается в **size1** и в **size2** с помощью **cout**. (Думаю, это Вы осилите и без моей помощи). Получим массив **[0][4]**. Когда функция foo инстанирует шаблон, то этот шаблон создаст массив m[0][4], а это совсем не наш **[3][4]**. Вы же можете создать, например, нулевой массив **int Arr[0][0];** А заполнить? Но то только пол беды. А если бы и не ноль строк, узнать число элементов сам массив может и может, а мы вот по идее не можем. **Внутри функции получается указатель, а размер указателя — это не размер массива**. В первом успешном варианте нас спасло то, что мы знали, что число строк и колонок одинаковое, а сейчас-то нам число строк неизвестно совсем. Но, решение есть и оно, как оказывается, очень даже не сложное. Для наглядности **size1** и **size2** поменяю на **Row** и **Col** соответственно, это не должно Вас смутить.

#include <iostream>

using namespace std;

template <const unsigned Row = 0, const unsigned Col = 0> //Два параметра по умолчанию

void foo(int m[][Col])

{

for (unsigned i = 0; i<Row; i++){

for (unsigned j = 0; j<Col; j++){

cout << m[i][j] << "\t";

} cout << "\n";

}

}

int main() {

int m[3][4] {{ 1, 2, 3, 4}, { 4, 5, 6, 7 }, { 8, 9, 10, 11 }};

foo<3>(m);//Заметьте, что мы не указываем число колонок, но можно указать через запятую. Так короче

system("PAUSE");

}



Несложно же. И это работает. До C++11 компиляторы не захотят компилировать эти коды, но время не стоит на месте. В завершении приведу еще один пример, не знаю насколько он полезен или же бесполезен, но он может оказаться весьма интересным для изучения возможностей языка C++. **Вернем ссылку на двумерный массив**.

#include <iostream>

//#include <mem.h>

#include <memory.h>

using namespace std;

template <const unsigned Row = 0, const unsigned Col = 0>

const int(&foo(const int m[][Col]))[Row][Col]

{

static int Arr[Row][Col]; //обязательно static

for (int i = 0; i<Row; i++){

for (int j = 0; j<Col; j++){

Arr[i][j] = m[i][j] \* 2; //Увеличиваем ячейку вдвое

}

}

return Arr;

}

int main() {

const int Row = 3, Col = 4;

int K[Row][Col] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } };

memcpy(K, foo<Row>(K), Col \* Row \* sizeof(K) / sizeof(\*K)); //Копируем в К массив, получившийся в функции

/\*выводим массив на экран\*/

for (const auto &i : K){

for (const auto &j : i){

cout << j << "\t";

}

cout << "\n";

}

system("PAUSE");

}



**static** нужен для того, чтобы по завершению работы функций, массив не самоубился. Вспоминаем о том, что все локальные переменные, созданные внутри функции по завершению функции умирают. А массив — это переменная. Поэтому в этом варианте ключевое слово **static** очень нам пригодилось.

**Дополнительно смотри тему 07\_Шаблоны в ОАП**

## Функции округления

Зачастую требуется воспользоваться округленным значением той или иной переменной. C++ предлагает набор функций для решения этой задачи. В зависимости от конкретной ситуации может понадобиться функция, округляющая значение аргумента в большую или меньшую сторону. Рассмотрим наиболее часто используемые варианты вызовов.

**Для округления числа в меньшую сторону используется функция floor()** и ее разновидности для различных типов аргументов и возвращаемых параметров. Данная функция имеет следующий синтаксис:

double floor(double x);

long double floorl(long double x);

**Округление в большую сторону производится с помощью функции ceil()**:

double ceil(double x);

long double ceil(long double x);

Однако в реальности проблема выбора, в какую же сторону производить округление, возлагается на разрабатываемую программу. Ниже предлагается два варианта решения этой задачи.

Вариант I:

inline double Round(double x)

{return floor(x +.5);}

Вариант II:

double round(double num)

{

double frac;

double val;

frac = modf(num, val);

if(frac < 0.5) num = val;

else num = val + 1.0;

return num;

}

Для работы обоих вариантов функции необходимо задействовать заголовочный файл math.h, содержащий прототип функций floor() и modf().

## Компиляция программ, состоящих из двух или более функций.

Простейший способ использования нескольких функций в одной программе заключается в том, чтобы поместить их в один файл, после чего осуществить компиляцию программы, содержащейся в этом файле так, как будто она состояла из одной функции

Второй способ заключается в применении директивы #include Если одна функция содержится в файле с именем filel.c, а вторая в файле file2.c, поместите эту директиву в файл filel.c

#include "file2.с"

Дополнительная информация о директиве #include будет приведена позднее. Другие возможные способы являются в большей степени системнозависимыми Вот некоторые из них

**ОС UNIX.** Предположим, filel.c и file2.c — два файла, содержащие программные тексты, соответствующие функциям языка Си. В результате выполнения команды

cc filel.c file2.c

будет осуществлена компиляция функций, содержащихся в обоих файлах, и получен файл выполняемого кода с именем a.out. Кроме того, будут созданы два файла с «объектным» кодом — file 1.0 и file2.0. Если позже вы измените текст, содержащийся в файле с именем file1.c, а второй файл оставите без изменений, то сможете осуществить компиляцию первого файла, а затем объединить полученный объектный код с объектным кодом, соответствующим второму файлу, при помощи команды

cc file.c file.0

**Компиляторы Lattice С и MICROSOFT С**. Выполните раздельную компиляцию функций, содержащихся в файлах filel.c и file2.c; в результате будут получены два файла с объектным кодом — filel.obj и file2.obj. Используйте системный редактор связей для объединения их друг с другом и со стандартным объектным модулем С. obj:

link с file1 file2

**Системы, построенные на основе трансляции в ассемблерный код.** Некоторые из таких систем позволяют компилировать функции, содержащиеся в нескольких файлах, сразу так же, как в ОС UNIX с помощью команды:

cc fue1.с file2.c

или какого-то ее эквивалента. В некоторых случаях вы можете получить отдельные модули с кодом ассемблера, а затем объединить их, используя процесс ассемблирования.

**Резюме.** Для создания больших программ вы должны использовать функции в качестве «строительных блоков». Каждая функция должна выполнять одну вполне определенную задачу. Используйте аргументы для передачи значений функции и ключевое слово return для передачи результирующего значения в вызывающую программу. Если возвращаемое функцией значение не принадлежит типу int, вы должны указать тип функции в ее определении и в разделе описаний вызывающей программы. Типичное определение функции имеет следующий вид:

имя (список аргументов)

описание аргументов

тело функции

***Пример***

diff(x, у) /\* имя функции и список аргументов \*/

int x, у; /\* описание аргументов \*/

{ /\* начало тела функции \*/

int z; /\* описание локальной переменной \*/

z = x - у;

return(z);

} /\* конец тела функции \*/

Наличие списка аргументов и описаний не является обязательным. Переменные, отличные от аргументов, описываются внутри тела, которое заключается в фигурные скобки.

Если вы хотите, чтобы при выполнении функции происходило изменение значений переменных в вызывающей программе, вы должны пользоваться адресами и указателями.

**Что вы должны были узнать.**

* Как определять функцию.
* Как передавать функции информацию: при помощи аргументов.
* Различие между формальным и фактическим аргументами: первый является переменной, используемой функцией, а второй — значением, поступающим из вызывающей функции.
* Где необходимо описывать аргументы: после имени функции и перед первой фигурной скобкой.
* Где необходимо описывать остальные локальные переменные: после первой фигурной скобки.
* Когда и как использовать оператор return.
* Когда и как использовать адреса и указатели для доступа к объектам.

**Вопросы и ответы.**

**Вопросы.**

1. Напишите функцию, возвращающую сумму двух целых чисел.
2. Какие изменения должны были бы произойти с функцией из вопроса 1, если вместо целых складывались бы два числа типа float?
3. Напишите функцию alter(), которая берет две переменные х и у типа int и заменяет соответственно на их сумму и разность.
4. Проверьте, все ли правильно в определении функции, приведенной ниже?

salami(num)

{

int num, count;

for(count = 1; count <= num; num++ )

printf("O салями! \n" );

}

**Ответы.**

sum(j,k)

{

return (j + k);

}

float sum(j.k)

float j,k;

Необходимо также привести описание функции float sum() в вызывающей программе.

1. Поскольку мы хотим изменить две переменные в вызывающей программе, можно воспользоваться адресами и указателями. Обращение к функции будет выглядеть так: alter(&x,&y) Возможное решение имеет следующий вид:

alter(px, py)

int \*px, \*ру; /\* указатели на х и у\*/

{

int sum, diff;

sura = \*px+ \*py; /\* складывает содержимое двух переменных, определяемых адресами \*/

diff = \*рх - \*ру;

\*рх = sum;

\*ру = diff;

}

1. Нет; переменная num должна быть описана перед первой фигурной скобкой, а не после нее. Кроме того, выражение num++ необходимо заменить на count++

**Упражнения.**

1. Напишите функцию max(х,у), возвращающую большее из двух значений.
2. Напишите функцию chline(ch.i,j), печатающую запрошенный символ с i-й по j-ю позиции. Смотри программу художник-график, приведенную ранее.