**Структурная организация программ**

**Функции**

До настоящего момента вы не были готовы организовывать код своих программ в модульном стиле, поскольку могли конструировать программу как единственную функцию - main (); однако вы использовали библиотечные функции различного рода, а также функции, принадлежащие объектам.

Всякий раз, начиная писать программу на С++, вы должны продумывать ее модульную структуру с самого начала, и как вы увидите, понимание того, как должны быть реализованы функции, существенно для объектно-ориентированного программирования на С++.

Что такое функции

Для начала рассмотрим общие принципы работы функций.

Функция - это изолированный блок кода, имеющий определенное специфическое назначение.

Функция обладает именем, которое как идентифицирует ее, так и служит для вызова на выполнение внутри программы. Имя функции глобально, но не обязательно уникально в С++, как вы увидите это в следующей главе; однако функции, которые выполняют различные действия, обычно должны иметь разные имена.

Имена функций подчиняются тем же правилам, что и имена переменных. То есть имя функции - это последовательность букв и цифр, начинающаяся с буквы, причем знак подчеркивания тоже считается буквой. Имя функции должно обычно отражать то, что она делает, поэтому, например, вы можете назвать функцию, которая подсчитывает бобы, *count\_beans ()*.

Вы передаете информацию функции с помощью аргументов, специфицированных при ее вызове. Эти аргументы должны соответствовать параметрам, появляющимся в определении функции. Когда функция выполняется, то указанные вами аргументы заменяют параметры, использованные в ее

определении. Код функции выполняется так, как будто он был написан с применением значений ваших аргументов. На рисунке показаны отношения между аргументами при вызове функции и параметрами, заданными в ее определении.

В этом примере функция возвращает сумму двух переданных ей аргументов. В общем случае функция возвращает либо одно значение в точку программы, откуда она была вызвана, либо вообще ничего - в зависимости от того, как она определена. Можно подумать, что возврат единственного значения из функции ограничивает ее возможности, но это единственное значение может быть указателем, содержащим, например, адрес массива. Чуть позднее в этой главе вы узнаете подробнее о возврате данных из функции.



Зачем нужны функции

Одно из главных преимуществ, предоставляемых функцией, состоит в том, что она может быть выполнена столько раз, сколько необходимо, в разных точках программы. Без такой возможности упаковывать блоки кода в функции, программы были бы намного больше, поскольку тогда пришлось бы повторять один и тот же код везде, где он может понадобиться. Но реальная необходимость в функциях вызвана тем, чтобы можно было разбивать программу на легко управляемые фрагменты для независимой разработки и тестирования.

Представьте себе действительно большую программу - скажем, в миллион строк кода. Программу такого размера практически невозможно написать без функций. Функции позволяют сегментировать программу так, что ее можно писать по частям, и тестировать каждую часть независимо, прежде чем соединять ее с прочими частями. Это также дает возможность распределить работу между членами команды разработчиков, где каждый член команды отвечает за четко определенную часть программы, с хорошо определенным функциональным интерфейсом для остального кода. Например, одна функция может выводить массив на печать, другая, к примеру, сортировать этот же массив, третья еще что-либо. Ну а теперь, конечно же, все это нужно прочувствовать при работе с кодом, здесь то вы и поймете суть и разберетесь с функциями в программировании на С++.

Структура функции

Как вы уже видели, когда писали функции *main(),* функция состоит из заголовка функции, который идентифицирует ее, а за ним следует тело функции, заключенное в фигурные скобки, где и содержится ее исполняемый код.

Попробуем написать функцию, которая будет возводить значение в заданную степень, то есть вычислять результат умножения значения *х* на себя *n* раз, что в математике записывается как хп.

// Листинг 24

// Функция для вычисления х в степени n, где n больше или равно 0

double power(double х, int n) // Заголовок функции

{ // Тело функции начинается здесь...

double result = 1.0; // Здесь сохраняется результат

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

result \*= x;

return result;

} // ...и заканчивается здесь

Заголовок функции

Сначала рассмотрим в этом примере заголовок функции. Следующая строка - первая строка функции:

*double power(double х, int n) // Заголовок функции*

Он состоит из трех частей, которые описаны ниже.

* Тип возвращаемого значения (в данном случае - *double*).
* Имя функции (в данном случае - *power*).
* Параметры функции, заключенные в скобки (в данном случае- *х* и *n*, типа *double* и *int* соответственно).
* Возвращаемое значение возвращается вызывающей функции, поэтому, когда данная функция вызывается, то ее результат типа *double* подставляется в выражение, из которого она вызвана.

Наша функция имеет два параметра: х - значение типа *double*, которое нужно возвести в степень, и *n* - значение степени типа *int*. Эти переменные параметры участвуют в вычислениях, выполняемых функцией, вместе с другой переменной — *result*, объявленной в ее теле. Имена параметров и любые переменные, определен­ные в теле функции являются локальными по отношению к ней.

В конце заголовка функции и после закрывающей фигурной скобки ее тела точка с запятой не требуется.

Общая форма заголовка функции

Общая форма заголовка функции может быть записана следующим образом:

тип\_возврата имя\_функции (список\_параметров)

тип\_возврата может быть любым легальным типом. Если функция не возвращает значения, то тип возврата указывается ключевым словом *void*. Ключевое слово *void* также применяется для обозначения отсутствия параметров, поэтому функция, которая не имеет параметров и не возвращает значения, должна иметь следующую форму заголовка:

*void my\_function(void)*

Пустой список параметров также означает, что функция не имеет аргументов, поэтому вы можете пропустить ключевое слово *void* между скобками:

*void my\_function ()*

Функция с типом возврата *void* не должна использоваться в составе выражений в вызывающей программе.

Тело функции

Все необходимые вычисления функции выполняются операторами в ее теле, которое следует за заголовком. Тело функции из нашего последнего примера начинается с объявления переменной *result*, инициализированной значением 1.0. Переменная *result* локальна по отношению к функции, как и все автоматические переменные, объявленные в ее теле. Это значит, что переменная *result* прекращает существование после того, как функция завершит работу. Скорее всего, у вас немедленно возникнет вопрос: если переменная *result* прекращает существование по завершении работы функции, то как она может быть возвращена? Ответ в том, что при этом автоматически создается копия значения, подлежащего возврату, и эта копия возвращается в программу.

Вычисление производится в цикле *for*. Управляющая переменная цикла *i*объявлена в самом цикле, и предполагается, что она последовательно принимает значения от 1 до *n*. Переменная *result* умножается на *х* при каждой итерации цикла, поэтому это происходит *n* раз, чтобы сгенерировать необходимое значение. Если *n* равно 0,то оператор цикла не будет выполнен ни разу, поскольку условное выражение сразу возвратит *false*, и *result* останется равным 1.0.

Параметры и все переменные, объявленные в теле функции, локальны по отношению к ней. Ничто не мешает вам использовать те же имена переменных в других функциях, для других целей. В самом деле, иначе было бы чрезвычайно трудно обеспечить уникальность имен переменных внутри программы, состоящей из множества функций, особенно, если эти функции разрабатывает не один человек.

Область видимости переменных, объявленных внутри функции, определяется таким же образом, как уже упоминалось. Переменная создается в точке ее объявления и прекращает свое существование в конце блока, в котором была объявлена. Однако существует разновидность переменных, которая составляет исключение из этого правила - переменные, объявленные как *static*.

Будьте осторожны с маскированием глобальных переменных одноименными локальными.

Оператор *return*

Оператор *return* возвращает значение *result* в точку вызова функции. Общая форма оператора *return* такова:

return выражение;

где выражение должно вычисляться как значение типа, определенного в заго­ловке функции для возврата значения. Выражение может быть любым, какое хотите, до тех пор, пока оно в результате отдает значение требуемого типа. Выражение может включать вызовы функций и даже вызов той самой функции, в которой оно появляется.

Если тип возврата функции специфицирован как *void*, то за оператором *return* не должно следовать никакого выражения. Оно должна записываться очень просто:

*return;*

Использование функций

В точке, где в программе используется функция, компилятор должен знать кое-что о ней, чтобы скомпилировать ее вызов. Ему нужна достаточная информация, чтобы идентифицировать функцию, и убедиться, что вы применяете ее корректно. И если только функция, которую вы намерены использовать, не появилась где-то ранее в том же исходном файле, вы должны объявить функцию с помощью оператора, который называется прототипом функции.

**Прототипы функций**

Прототип функции предоставляет базовую информацию, которую компилятор должен проверить, чтобы убедиться, что функция используется корректно. Он определяет параметры, передаваемые функции, ее имя и тип возвращаемого значения. По сути, прототип содержит ту же информацию, что содержится в заголовке функции, с добавлением точки с запятой. Понятно, что количество параметров и их типы в прототипе функции должны быть такими же, как в ее заголовке.

Прототипы функций, которые вызываются из другой функции, должны появляться перед операторами, где эти функции вызываются, и потому обычно помещаются в начале исходного файла программы. Заголовочные файлы, которые вы включаете для использования стандартных библиотечных функций, помимо прочего, включают в себя прототипы этих библиотечных функций.

Для примера функции *power ()* вы можете написать следующий прототип:

*double power(double value, int index);*

He забывайте о точке с запятой в конце прототипа функции! Без этого вы получите от компилятора сообщение об ошибке.

Обратите внимание, что имена параметров в прототипе функции, отличающиеся от тех, что применялись в заголовке функции при ее определении. Это просто для того, чтобы показать, что такое возможно. Чаще в прототипах указываются те же имена, что и в заголовке определения функции, но это не обяза­тельно должно быть так.

Вы можете применять более длинные и выразительные имена параметров в прототипе функции, чтобы пояснить их назначение, а потом указать более короткие имена тех же параметров в определении функции, где длинные имена могли бы загромоздить код и сделать его менее читабельным.

При желании вы можете вообще пропустить имена параметров в прототипе функции и просто написать так:

*double power(double, int);*

Это предоставляет достаточно информации компилятору, чтобы он выполнил свою работу; однако лучше использовать некоторые осмысленные имена в прототи­пе, потому что это повышает читабельность и в некоторых случаях вообще отличает ясный код от запутанного. Если у вас есть функция с двумя параметрами одного и того же типа (предположим, к примеру, что у нас *index* в функции *power ()* также был бы типа *double*), то выбор для них осмысленных имен показывает, какой пара­метр идет первым, а какой вторым.

Использование функции

// Листинг 25

//Объявление, определение и применение функции

#include <iostream.h>

**double power(double x, int n);** //Прототип функции

void main ()

{

int index = 3;

double x = 3.0;

double y = 0.0;

**y = power(5.0,3) ;** //Передача констант в виде аргументов

cout << endl << “ 5.0 v kube = “ <<y ;

cout <<endl

<< “ 3.0 v kube = “

<<**power(3.0, index)** ; //Вывод возвращенного значения

**x = power (x, power(2.0, 2.0))** ;//применение функции в качестве //аргумента с автоматич. привед. 2-го параметра

cout << endl << “x =“ <<x;

cout <<endl;

}

//Функция для вычисления x в степени n, где n больше или равно 0

**duble power(double x, int n)** //Заголовок функции

**{** //Тело функции начинается здесь…

**double result = 1.0 ;**

**for(int i = 1 ; i<=n ; i++)**

**result \*=x;**

**return result;**

**}**//… и заканчивается здесь

После обычного оператора *#include* для поддержки ввода-вывода идет прототип функции *power ().* Если вы удалите его и попытаетесь перекомпилировать программу, то компилятор не сможет обработать вызовы функции в *main (),* а выдаст вместо этого серию сообщений об ошибках.

В этом примере использовано ключевое слово *void* в функции *main (),* где обычно появляется список параметров, чтобы показать, что параметры не применяются. Ключевое слово *void* также может применяться в качестве типа возврата функции, чтобы показать, что функция не возвращает значения. Если вы определяете тип возврата как *void*, то не должны помещать никакого значения рядом с оператором *return* внутри функции; в противном случае вы получите от компилятора сообщение об ошибке.

Чтобы использовать функцию power () для вычисления 5,03 и сохранить результат в переменной у в нашем примере используется следующий оператор:

*у = power (5.0, 3);*

Здесь значения 5.0 и 3 - аргументы функции. В данном случае они являются константами, но вы можете использовать любые выражения в качестве аргументов, если они дают результат требуемого типа.

Аргументы функции *power ()* подставляются вместо параметров *х* и *n*, которые используются в определении функции. Вычисление производится с применением этих значений, а копия результата, 125, возвращается вызывающей функции *main (),* где присваивается переменной *у*. Вы можете думать о функции как о значении в операторе или выражении, где она появляется.

Затем в примере значение переменной *у* выводится на экран:

*cout << endl<< “ 5.0 v kube = “ <<y ;*

Далее вызов функции применяется прямо в составе оператора вывода:

*cout <<endl*

*<< “ 3.0 v kube = “*

*<<power(3.0, index) ; //Вывод возвращенного значения*

Здесь значение, возвращаемое функцией, передается непосредственно в выходной поток. Поскольку вы нигде не сохраняете это значение, оно никаким другим способом вам недоступно. Первый аргумент в этом вызове функции - константа, а второй - переменная.

После этого функция *power ()* используется еще раз в операторе:

*х = power(х, power(2.0, 2.0)); // Использование функции, как аргумента*

В этом случае функция power () вызывается дважды. Первый вызов - правый в выражении, и его результат служит вторым аргументом для второго, левого вызова. Хотя оба аргумента в подвыражении power (2 . 0, 2.0) являются литералами типа *double*, на самом деле функция вызывается с первым аргументом 2 . 0 и вторым — целочисленным литералом 2. Компилятор преобразует значение *double*, приведенное в качестве второго аргумента, к типу *int,* поскольку знает на основании прототипа, что типом второго аргумента должен быть *int*:

*double power(double x, int n); // Прототип функции*

Результат 4.0 типа *double* возвращается первым вызовом функции *power()* после преобразования к типу *int* значение 4 передается в качестве второго аргумента следующему вызову функции, где первый аргумент - *х*. Поскольку *х* имеет значение 3.0, значение вычисляется 3,04 и результат, равный 81.0, присваивается *х*.

Этот оператор заключает в себе два неявных преобразования типа *double* в тип *int*, вставку которых обеспечивает компилятор. При таком преобразовании данных возможна потеря данных, поэтому компилятор издает предупреждающие сообщения, когда такое случается, хотя он сам и вставил это преобразование. Обычно полагаться на автоматическое преобразование типов, потенциально чреватое потерей данных - опасная практика в программировании, и совсем не очевидно вытекает из кода, что такое преобразование было намеренным. Гораздо лучше, когда необходимо, явно указывать в коде преобразование типа, используя

для этого операцию static\_cast. То есть последний оператор в этом примере лучше переписать так:

*х = power (х, static\_cast<int>(power(2.0, 2) ) ) ;*

Такое кодирование оператора позволяет избежать обоих предупреждений компилятора, которые вызывает исходная версия. Применение статического приведения не исключает возможности потери данных при преобразовании одного типа в другой. Но , поскольку оно указано явно, то компилятору ясно, что это входит в ваши намерения.

Передача аргументов в функцию

Очень важно понимать, как аргументы передаются в функцию, потому что это влияет на то, как вы их пишете, и то, как они в конечном итоге работают. Здесь есть множество ловушек, которых следует избегать, поэтому рассмотрим этот механизм поближе.

Аргументы, которые вы определяете при вызове функции, обычно должны соответствовать по типу и последовательности параметрам, заданным в ее опреде­лении. Как вы видели из последнего примера, если тип аргумента, указанного при вызове функции, не соответствует типу параметра, указанному в ее определении, то там, где возможно, происходит преобразование к требуемому типу в соответствии с правилами приведения операндов, описанными ранее. Если такое приведение невозможно, вы получаете от компилятора сообщение об ошибке. Однако даже если приведение возможно и код компилируется, это может привести к потере данных (например, когда тип long приводится к short), а потому его следует избегать.

Существуют два механизма, обычно используемые С++ для передачи аргументов в функции. Первый применяется, когда параметры в определении функции - обычные переменные (не ссылки). Это называется методом передачи по значению (pass-by-value) данных в функцию.

Механизм передачи по значению

Когда применяется такой механизм, то переменные или константы, которые вы специфицируете в качестве аргументов, вообще не передаются в функцию. Вместо этого создаются копии аргументов, и эти копии используются в качестве передаваемых значений. На рисунке показана диаграмма применительно к функции power ().

Каждый раз, когда вызывается функция *power (),* компилятор создает копии переданных аргументов и размещает их во временном участке памяти. Во время выполнения функции все ссылки на ее параметры отображаются на эти временные копии аргументов.



Одно из последствий механизма передачи по значению состоит в том, что функция не может напрямую модифицировать

// Листинг 25

// Бесполезная попытка модификации аргументов вызывающего кода

#include <iostream.h>

int incr10(int num); // Прототип функции

int main(void)

{

int num = 3;

cout << endl

<<” int incr10( num) = “<<int incr10( num)

<< endl

<< “num = “ << num;

}

// Функция для увеличения переменной на 10

int incr10(int num) // Применение такого имени может помочь...

{

num +=10; // Попытка изменить аргумент

return num; // Возврат измененного значения

}

Конечно, эта программа обречена на неудачу. Если вы запустите ее, то получите следующий вывод:

incr10(num) = 13

num = 3

Описание полученных результатов

Вывод подтверждает, что исходное значение num остается незатронутым. Изменению подвергается копия num, которая была создана и затем отброшена при выходе из функции.

Понятно, что механизм передачи по значению обеспечивает высокую степень защиты аргументов вызова от повреждения недобросовестной функцией, но все же может случиться, что вы действительно хотите, чтобы переданные аргументы были модифицированы функцией.

**Указатели как аргументы функций**

Когда вы используете указатели в качестве аргументов, механизм передачи по значению работает, как и прежде; однако указатель - это адрес другой переменной, и если вы возьмете копию этого адреса, то она будет по-прежнему указывать на ту же переменную. Таким образом, применение указателя в качестве параметра позволяет вашей функции получить сам аргумент, а не его копию.

Для демонстрации эффекта можно изменить последний пример, чтобы он использовал указатель:

// Листинг 26

//Успешная попытка модифицировать аргумент

#include <iostream.h>

int incr10 (int \*num); //Прототип функции

void main()

{

int num = 3;

int\*pnum = &num; //Указатель на num

cout <<endl

<< “adress has been received = ”<< pnum;

<< endl

<< “incr10(pnum) = ”<< incr10(pnum);

cout <<endl

<< “num = ”<< num;

}

// Функция для увеличения переменной на 10

int incr10(int\*num) //Функция с аргументом указателем

{

cout <<endl

<< “adress has been received = ”<< num;

\*num+=10; //Инкремент аргумента

return \*num; // Возврат измененного значения

}

Результат

adress has been received = 0012С6С

adress has been received = 0012С6С

incr10(pnum) = 13

num = 13

Точное значение адреса на вашем компьютере может отличаться от показанного

здесь, но два отображенных значения адреса должны быть идентичными.

Описание полученных результатов

Принципиальное отличие этого примера от предыдущей версии заключается в передачи указателя pnum вместо исходной переменной num. Прототип функции те­перь специфицирует тип параметра как указатель на *int*, а в функции *main ()* объяв­лен указатель *pnum*, инициализированный адресом *num*.

Функция *main ()* и функция *incr10 ()* выводят соответственно отправленный и принятый адрес, чтобы показать, что в обоих местах используется один и тот же адрес.

Результат программы показывает, что на этот раз значение *num* было изменено, и имеет значение, идентичное тому, что возвратила функция.

В измененной версии функции *incr10 ()* теперь и оператор, изменяющий переданное значения, и оператор *return* выполняют разыменование указателя, чтобы обратиться к значению, хранящемуся в нем.

В данном примере есть определенная доза лукавства. Мы узнали, как передавать в функцию указатель, как можно с помощью передачи указателя модифицировать переменную , но аргумент по-прежнему не был изменен. В данном случае аргументом был указатель, и он не изменился. Изменилась только переменная, на которую он указывал. Как действительно изменить аргумент в функции, мы узнает немного позже.

**Передача массивов в функцию**

В функцию можно передавать массивы, но в этом случае массив не копируется, даже несмотря на то, что при этом по-прежнему применяется передача аргумента по значению. Имя массива преобразуется в указатель, и копия указателя на начало массива передается в функцию по значению.

Это достаточно выгодно, потому что копирование больших массивов требует значительных затрат времени. Однако, как вы можете обнаружить в этом случае, элементы массива могут быть изменены внутри функции, и потому массив - единственный тип, который не может быть передан по значению

Плюсы и минусы этого можно проиллюстрировать на примере функции, вычисля­ющей среднюю величину значений, переданных в массиве.

// Листинг 27

// Передача массива в функцию

#include <iostream.h>

double average(double array[], int count); // Прототип функции

int main(void)

{

double values [] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 };

cout << endl

<< "AV= "

<< average(values, (sizeof values)/(sizeof values[0]));

cout << endl;

return 0;

}

// Функция для вычисления среднего

double average(double array[], int count)

{

double sum =0.0; // Здесь накапливается сумма

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

sum += array[i]; // Суммировать элементы массива

return sum/count; // Вернуть среднее

}

Результат:

Среднее =5.5

Описание полученных результатов

Функция *average ()* предназначена для работы с массивами любой длины. Как видно из ее прототипа, она принимает два аргумента: массив и счетчик количества элементов. Поскольку мы хотим, чтобы она работала с массивами произвольной длины, параметр массива указан без спецификации измерения. Функция вызывается в *main ()* следующим оператором:

cout << endl

<< "AV= "

<< average(values, (sizeof values)/(sizeof values[0]));

Первым аргументом передается имя массива, *values*, а вторым - выражение, которое вычисляется, как количество элементов массива.

Операция *sizeof* возвращает целое значение, которое означает количество байт, которое занимает ее операнд. Применяется к массиву и к элементу массива.

Внутри тела функции вычисление выполняется так, как можно было ожидать.

Вывод программы подтверждает, что все работает, как надо.

**Использование нотации указателей при передаче массивов**

Но этим мы не исчерпали все возможности. Как было сказано, имя массива пе­редается в виде указателя (точнее говоря, в виде копии указателя), поэтому внутри функции вы вообще не обязаны работать с данными как с массивом. Можно так изменить функцию в примере, чтобы повсюду использовать нотацию указателей, несмотря на тот факт, что речь идет о массиве.

// listing 28

// Сокрытие массива в функции указателем

#include <iostream.h>

double average(double\* array, int count); // Прототип функции

int main (void)

{

double values [] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};

cout << endl

<< "AV= "

<< average(values, (sizeof values)/(sizeof values[0]));

cout << endl;

return 0;

}

// Функция для вычисления среднего

double average(double\* array, int count)

{

double sum =0.0; // Здесь накапливается сумма

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

sum += \*array++; // Суммировать элементы массива

return sum/count; // Вернуть среднее

}

Результат:

Среднее =5.5

Вывод программы выглядит точно так же, как в предыдущем примере.

Описание полученных результатов

Как видите, для того, чтобы программа работала с массивом как с указателем, в нее требуется внести совсем немного изменений. Изменяется прототип и заголовок функции, хотя эти изменения не являются абсолютно необходимыми. Если вернуть их к исходному виду, где первый параметр специфицирован как массив *double*, и оставить тело функции в терминах указателя, она будет работать так же хорошо. Самый интересный аспект этой версии тела функции, кроется в операторе цикла for:

*sum += \*array++; // Суммировать элементы массива*

Можно подумать, что здесь нарушается правило о том, что нельзя модифицировать адрес, специфицированный как имя массива, поскольку увеличивается адрес, хранящийся в array. Но на самом деле это правило вовсе не нарушается. Механизм передачи по значению создает копию исходного адреса массива и передает его, а потому в теле функции модифицируется лишь эта копия (исходный адрес массива остается неизменным).

В результате всякий раз, когда вы передаете одномер­ный массив в функцию, то вольны трактовать переданное значение в любом смысле как указатель, а также произвольным образом изменять адрес, хранящийся в нем.

Передача в функцию многомерных массивов

Передача в функцию многомерного массива достаточно проста. Следующий оператор объявляет двумерный массив *beans:*

*double beans[2][4] ;*

Вы можете написать прототип гипотетической функции *field ()* примерно так:

*double field(double beans [2] [4]);*

Вас может удивить- как компилятор может знать, что это определение массива с размерностью, указанной в квадратных скобках, а не отдельного элемента массива ? Ответ прост - вы не можете написать отдельный элемент массива в качестве параметра в определении или прототипе функции, хотя можете передать его в качестве аргумента при ее вызове. Параметр, принимающий отдельный элемент массива в качестве аргумента, должен быть объявлен как отдельная переменная. Контекст массива здесь не применим.

При определении многомерного массива в качестве параметра вы можете также пропустить первое измерение массива. Конечно, функция должна как-то узнать о размере первого измерения. Например, вы можете написать так:

*double field(double beans[] [4], int index);*

Здесь второй параметр может передать необходимую информацию относительно величины первого измерения. В этом случае функция может оперировать двумерным массивом со значением первого измерения, указанным вторым аргументом функции, и фиксированным значением второго измерения, равным 4.

Использование подобной функции показано в следующем примере.

// listing 29

// Передача в функцию двумерного массива

#include <iostream.h>

double field(double array[] [4], int n) ;

int main(void)

{

double beans[3][4] = { { 1.0, 2.0, 3.0, 4.0 },

{ 5.0, 6.0, 7.0, 8.0 },

{ 9.0, 10.0, 11.0, 12.0 } };

cout « endl

« " harvest = " << field(beans, sizeof beans/sizeof beans[0]);

cout « endl;

return 0;

}

// Функция для вычисления всего объема урожая

double field(double beans [] [4], int count)

{

double sum = 0.0;

for (int i = 0; i < count; i++) // Цикл по строкам

for (int j = 0; j < 4; j++) // Цикл по элементам строки

sum += beans [ i ] [ j ] ;

return sum;

Результат:

harvest =78

Описание полученных результатов

Были использованы разные имена параметров в прототипе и заголовке функции, просто чтобы напомнить, что это возможно (правда, в данном случае это ничем не улучшает программу).

Первый параметр определен как массив с произвольным количеством строк, каждая из которых содержит четыре элемента. Функция вызывается с массивом beans, состоящим из трех строк.

Второй аргумент определен как частное от деления общего размера массива в байтах на размер его первой строки. В результате вычисления это дает количество строк в массиве.

Вся вычислительная работа функции выполняется во вложенных циклах for, где внутренний цикл суммирует элементы отдельной строки, а внешний цикл повторяет­ся для каждой строки.

Использование указателя вместо многомерного массива в качестве аргумента функции в этом примере не очень подходит. Когда передается массив, то на самом деле передается значение адреса, которое в данном случае указывает на первую строку- массив из четырех элементов. Это само по себе не обеспечивает простых операций с указателем внутри функции. В этом случае придется модифицировать оператор во вложенном цикле *for* следующим образом:

*sum += \* (\* (beans + i) + j) ;*

поэтому вычисления в нотации массива все-таки выглядят яснее.