Функции в языке высокого уровня

Учебные и воспитательные цели

1. Дать основные сведения о функциях в языке C++.
2. Изучить механизм передачи данных в функцию.

Учебные вопросы

1. Метод структурного программирования. Обращение к функциям.
2. Классы памяти и области действия.
3. Рекурсия в языке C++.

Литература

1. Освой самостоятельно Borland C++ 5 / Крейг Арнуш. – М.: Бином, 1997. – 720 с.
2. Освой самостоятельно Borland C++ Builder 3 / Кент Рейсдорф: Пер с англ. – М.: ЗАО Издательство БИНОМ, 1999. – 736 с.

1. Метод структурного программирования. Обращение к функциям

Жизненно важная составляющая языка C++ – функция. Идея функции состоит в том, чтобы разместить логически различные части программы в изолированные, легко обозримые фрагменты кода.

Большинство языков программирования используют понятия функции и процедуры. C++ формально не поддерживает понятие процедуры. Все подпрограммы C++ являются функциями, а процедура вводится как частный случай функции с возвращаемым значением типа void.

Функции являются первичными кирпичиками языка C++, которые логически расширяют его для решения специальных задач. Для простоты можно представить функцию как маленькую подпрограмму, выполняющую очень специфическую часть общей задачи.

Объявление и определение функций

Общая форма стандарта ANSI C объявления и определения функций (поддерживаемого и C++) такова:

возвращаемыйТип имяФункции(<список типизованных параметров>)
{
    <тело функции>
}

Примеры

```
double sqr(double y) { return y * y }
char prevChar(char c)
{
    return --c;
}
```

Запомните следующие правила объявления и определения функций в C++:

1. При объявлении функции перед именем функции всегда указывается тип возвращаемого ею значения.
2. Если список параметров пуст, круглые скобки после имени функции обязательно должны быть указаны. C++ позволяет вам использовать ключевое слово void для явного указания того, что список параметров пуст, хотя это будет возвратом к стандарту языка C.
3. Список параметров с указанием типов должен иметь следующий вид: [const] тип1 параметр1, [const] тип2 параметр2, ... Параметры задаются подобно тому, как определяются переменные: сначала вы задаете тип параметра, а потом идентификатор параметра. В C++ параметры в списке разделяются запятыми. Вы не можете сгруппировать параметры одного типа, указав их тип единожды, для каждого параметра вы должны указать его тип. Ключевое слово const объявляет компилятору, что значение параметра не должно изменяться функцией.

4. Тело функции C++ заключается в фигурные скобки ({}). За закрывающей скобкой не должно стоять точки с запятой.
5. C++ поддерживает передачу параметров, как по ссылке, так и по значению. По умолчанию параметры передаются по значению. В этом случае функция работает с копией передаваемых данных, оставляя оригинал данных неизменным. Для определения параметра, передающегося по ссылке, необходимо после указания типа ввести символ &. Имя ссылочного параметра является псевдонимом фактического параметра. Любое изменение ссылочного параметра влечет за собой изменение переменной, переданной в качестве фактического параметра. Общая форма задания ссылочных параметров: [const] тип1& параметр1, [const] тип2& параметр2, ... Ключевое слово const объявляет компилятору, что значение параметра не должно изменяться функцией.
6. C++ поддерживает локальные константы и локальные данные различных типов. Все эти элементы могут быть объявлены во вложенных блоках операторов, однако C++ не поддерживает вложенных функций.
7. Ключевое слово return передает в вызывающую функцию возвращаемое значение, если оно существует.
8. Если тип возвращаемого функцией значения – void, вы не обязаны использовать оператор return, кроме того случая, когда вам нужно произвести выход из функции прежде, чем она выполнится целиком. В этом случае оператор return не имеет параметра, ничего возвращать не нужно.

C++ требует, чтобы либо объявление, либо определение функции предшествовало ее вызову. Объявление функции, называемое обычно *прототипом,* содержит имя функции, тип возвращаемого функцией значения и количество, и тип ее параметров. Указание имен параметров не является обязательным. За списком параметров, после закрывающейся скобки, должна следовать точка с запятой, а не операторный блок. C++ требует, чтобы вы объявили функцию, если вызываете ее до того, как она будет полностью определена.

Ниже приведен простой пример объявления прототипа:

```
// Объявление прототипа функции
sqr double sqr(double);
int main()
{
        cout << “5^2 = ” << sqr(5) << “\n”;
        return 0;
}
// Определение функции
double sqr(double z) { return z * z; }
```

Обратите внимание на то, что объявление функции sqr содержит только тип ее единственного параметра. Это является обычной практикой, однако я не советую вам так поступать. Задание имени параметра в прототипе функции

делает более ясным смысл функции, особенно если вы написали ее давно и забыли о ее назначении.

Обычно объявленная функция является глобальной, т. е. она доступна всем другим функциям. Однако вы можете объявить прототип функции в той функции, из которой она вызывается; в этом случае прототип будет скрыт от других функций.

При вызове функции вы должны поставить в соответствие параметрам передаваемые аргументы. Соответствие аргументов параметрам устанавливается в том порядке, в котором они были объявлены. Тип аргументов должен совпадать или быть совместимым с типом параметров. Рассмотрим, в качестве примера, функцию volume, определенную следующим образом:

```
double volume(double length, double width, double height)
{
        return length * width * height;
}
```

При вызове функции volume вы должны передать ей три параметра типа double либо совместимого типа (в данном случае любого численного типа). Ниже приведены несколько простых примеров вызовов функции volume:

```
double len = 34, width = 55, ht = 100;
int i = 3;
long j = 44;
unsigned k = 33;
cout « volume(len, width, ht) « endl;
cout « volume(1, 2, 3) « endl;
cout « volume(i, j, k) « endl;
cout « volume(len, j, 22.3) « endl;
```

C++ дает вам возможность игнорировать возвращаемое функцией значение. Такой способ вызова функции используется, когда вас интересуют только выполняемые функцией действия, а не возвращаемое значение. В этом случае вы используете функцию наподобие процедуры языка Pascal и других процедурных языков программирования.

Что такое параметр?

Вы можете облегчить себе жизнь, если лучше разберетесь в том, каким образом параметры передаются в функцию. Когда вы определяете переменные в списке параметров функции, вы объявляете новые переменные внутри функции, которым при вызове функции будут передаваться значения. Внутри функции создается копия передаваемых величин, и именно она используется в вычислениях. Ввиду того, что параметры – это лишь копия, их изменение не влечет за собой изменение тех величин, что были переданы в функцию из вызывающей функции. Например:

```
void foo(int parm)
{
        ++parm;
```

```
}
int main()
{
        int value = 5;
        foo(value);
        cout « value « endl;
        return 0;
}
```

Результатом работы этой программы будет число *5*. Хотя функция foo изменяет свой параметр, передаваемая в качестве параметра переменная не изменяется. Если вы хотите изменить переменную-оригинал, вы можете воспользоваться способом передачи параметра по ссылке. Ссылочный параметр работает так же, как и ссылочные переменные, которые мы изучали ранее. Перепишем последний пример:

```
void foo( int &parm) // используется ссылка на переменную целого типа
{
        ++parm;
}
int main()
{
        int value = 5;
        foo(value);
        cout « value « endl;
        return 0;
}
```

Результатом работы новой программы будет число 6. Здесь переменная value инициализируется числом 5, передается в функцию через ссылочную переменную parm, которая затем увеличивается. Так как эта переменная – ссылочная, она хранит не число 5, а ссылку на переменную-оригинал value, где и находится число 5. Поэтому при изменении parm изменяется и переменная value.

В старом C было принято передавать в качестве параметра указатель, чтобы получить тот же результат, который в C++ достигается применением ссылочного параметра. Вы можете найти много примеров такого рода в различных программах. Но и в программах на C++ этот способ применяется, например, в программировании для Windows при работе с динамическими библиотеками независимых фирм или при обращении к функциям Windows напрямую. Небольшая переделка нашей простой программы позволит нам использовать в качестве параметра указатель:

```
void foo(int *parm) // используется указатель на переменную целого типа
{
        ++*parm;            //не забудьте символ *
}
int main( )
{
```

```
    int value = 5;
    foo(&value);        // передается адрес переменной
    cout « value « endl;
    return 0;
}
```

Заметьте, что такой способ кажется несколько громоздким по сравнению с использованием ссылочной переменной, вы должны следить за тем, что делаете, правильно ли обрабатываете данные и т. д.

Другой веский аргумент в пользу использования ссылочных параметров – это передача в функцию больших объектов. Когда вы копируете что-то вроде переменной целого типа или указателя, проблемы нет, поскольку эти объекты занимают всего несколько байт. Но когда нужно будет передать такой объект, как структура, вам придется копировать огромное количество данных. Ведь некоторые структуры могут иметь размер в килобайт и более. А поскольку копирование будет происходить при каждом вызове функции, работа вашей программы может замедлиться настолько, что вы заметите это невооруженным глазом.

В этом случае наилучшее решение – это передача структуры по ссылке. Теперь вы копируете всего лишь несколько байт, чтобы создать ссылку на структуру, а не все содержимое структуры. К сожалению, при таком подходе вы создаете себе и новую проблему – теперь ваша функция может изменить содержимое структуры.

Использование модификатора const в списке аргументов

Здесь нам на помощь приходит модификатор const. До сих пор мы имели с ним дело при объявлении именованных констант. Он очень полезен при объявлении данных типа ссылки и указателя. Применив этот модификатор к ссылочному параметру функции, вы не только избавите себя от неоправданного копирования данных, но и не позволите функции испортить эти данные. Функция сможет использовать содержимое структуры, но не сможет изменить ее ни в какой из ее частей.

Общая форма использования модификатора const в списке параметров приведена ниже:

```
возвращаемыйТип имяФункции(const типАргумента &аргумент);
Пример:
struct userlnfo
{
        int age;
        char name[150]; // Большой размер, чтобы вместить любое имя
}
void processUserlnfo(const userlnfo &ui)
{
        if(ui.age < 18)
        {
                cout « "Вы слишком малы." « endl;
```

```
            return;
        }
        if(ui.age < 21)
            ui.age = 21; //  !!! ОШИБКА !!!
}
```

Обратите внимание на строку с комментарием, сообщающем об ошибке. У компилятора не возникнет проблем с той частью функции processUserlnfo, которая проверяет, достаточно ли зрел пользователь, но он сразу же остановиться с сообщением об ошибке, как только перейдет к той части этой функции, где она меняет возраст пользователя.

А теперь, после всех этих аргументов в пользу передачи структур по ссылке, а не по значению, позвольте заметить, что нет ничего страшного в том, чтобы передавать небольшие структуры по значению. Когда структура состоит только из нескольких переменных и занимает немного памяти, нет аргументов против того, чтобы передавать ее по значению. Но, вероятно, было бы лучше взять себе за правило: всегда передавать структуры по ссылке.

В C++ функции могут не только принимать, но и возвращать ссылку на переменную. Важно помнить следующее:

- Если возвращаемое значение – указатель, нельзя в операторе return функции указывать адрес локальной переменной.
- Если возвращаемое значение ссылка, нельзя оператором return возвращать локальную переменную.

После выхода из функции локальная переменная не существует, и в результате получим повисшую ссылку, т.е. возвращаемое значение будет ссылаться на область стека (памяти), которая не будет содержать локальную переменную.

Функции и локальные переменные

Одно из требований структурного программирования – это максимальная независимость и универсальность функций. Очевидно, чтобы обеспечить такую гибкость, функции должны иметь свои типы данных, константы и переменные.

Локальная переменная функции существует только во время вызова этой функции. Как только происходит возврат из функции, исполняющая система удаляет локальные переменные и освобождает память. Следовательно, между вызовами функции содержимое локальных переменных теряется. Инициализация локальных переменных производится каждый раз при вызове функции.

Используйте локальные переменные для хранения и изменения тех параметров, которые объявлены с модификатором const. Не следует объявлять локальную переменную с тем же именем, что и у глобальной переменной, используемой в данной функции. Локальная переменная скрывает глобальную переменную с тем же именем, в результате затрудняется использование глобальной переменной и создается путаница.

В листинге 1 представлен код простейшей программы использующей локальную переменную.

```
// Программа знакомит с понятием локальной переменной
#include <iostream.h>
void increment(int start)
{
        int ix;
        for(ix = start; ix < start + 5; ++ix)
                cout « ix « endl;
}
int main( )
{
        increment(23);
        return 0;
}
```

Листинг 1.

Функция increment использует параметр start в качестве начала отсчета в коротком цикле for, который увеличивает локальную переменную ix и выводит ее значение. Функция main просто вызывает increment со значением start равным 23.

Статические переменные в функциях

В программе, приведенной в листинге 1, локальная переменная теряет свое значение после завершения программы. C++ позволяет вам определить локальную переменную, сохраняющую свое значение между вызовами функции, простым применением ключевого слова static слева от описателя типа. Статическая переменная обычно инициализируется, причем инициализация выполняется один раз, при первом вызове функции.

В практике программирования встречаются случаи, когда желательно сохранение значения локальной переменной между вызовами функции. Этот тип локальной переменной называется *статической переменной.* В своем роде, это вариант глобальной переменной, поскольку в памяти компьютера эти переменные располагаются рядом, но доступна эта переменная только из той функции, в которой она объявлена, поскольку является локальной переменной данной функции.

Когда выполнение функции завершается, значение статической переменной сохраняется. Компилятор поддерживает это свойство языка, помещая статические переменные в область памяти, отводимую при начале работы программы и поддерживаемую на протяжении всего времени ее выполнения. Вы можете использовать одно имя для статических переменных в разных функциях. Это не смутит компилятор, поскольку он отслеживает функциональную принадлежность каждой переменной. В листинге 2 представлен исходный код программы использующей статическую переменную.

```
// Использование статической локальной переменной
#include <iostream.h>
double average(double x)
{
	static double count = 0;
	static double sum = 0;
	++count;
	sum += x;
	return sum / count;
}
int main()
{
	cout << “average = ” << average(1) << endl;
	cout << “average = ” << average(2) << endl;
	cout << “average = ” << average(4) << endl;
	cout << “average = ” << average(10) << endl;
	cout << “average = ” << average(11) << endl;
	return 0;
}
```

Листинг 2.

Программа выдаст следующий результат:
average = 1
average = 1.5
average = 2.33333
average = 4.25
average = 5.6

Программа в листинге 2 определяет функцию average, которая имеет статические локальные переменные. Функция инициализирует эти переменные нулями. Заметим, что инициализация происходит один раз, еще до первого вызова функции. При следующих вызовах average строки, где происходит инициализация, будут обходиться, и управление будет передаваться оператору, который увеличивает значение переменной count. Следующий оператор увеличивает значение переменной sum на величину параметра x. Функция возвращает вычисленное значение среднего, полученное делением sum на count.

В функции main производится серия вызовов функции average. Операторы стандартного потокового вывода выводят обновленное значение average. Это стало возможным благодаря статическим локальным переменным count и sum функции average. Если бы C++ не поддерживал статических переменных, вам бы пришлось обращаться за помощью к глобальным переменным, что было бы довольно сомнительным вариантом.

Досрочный выход из функции

Обычно бывает нужным прервать выполнение функции при наступлении условий, не позволяющих продолжать вычисления. Такой выход в C++ выполняет оператор return. Для функции типа void используется return без возвращаемого выражения. Во всех других функциях оператор return должен возвращать значение, которое может объяснять причину возврата из функции.

Аргументы по умолчанию

В C++ вы можете назначить параметрам функции значения по умолчанию.

Аргументы по умолчанию – это простое и одновременно мощное языковое средство, которое означает следующее: если вы опустили значение аргумента для параметра, который имеет аргумент по умолчанию, для этого аргумента автоматически будет использовано значение по умолчанию.

При задании аргументов по умолчанию нужно следовать следующим правилам:

- Если вы присваиваете значение по умолчанию какому-то параметру, вы должны задать значения по умолчанию и для всех остальных параметров в списке, следующих за этим параметром. Нельзя присваивать значения по умолчанию в случайном порядке. Таким образом, список параметров делится на две части: параметры, которые не имеют значений по умолчанию, и параметры, которые имеют такие значения.
- Вызов функции должен содержать аргумент для каждого параметра, не имеющего значений по умолчанию.
- При вызове функции можно опустить аргументы для параметров, имеющих значения по умолчанию.
- Если вы опустили аргумент для параметра, имеющего используемый по умолчанию аргумент, вы должны опустить аргумент и для всех оставшихся в списке параметров.

Наилучший вариант перечисления параметров в списке – это их упорядочение по частоте использования значения по умолчанию. Располагайте первыми параметры, значения по умолчанию которых используются редко, а последним ставьте параметр, который принимает значение по умолчанию чаще всех предыдущих.

В листинге 3 представлен простой пример программы использования аргументов по умолчанию.

```
// Программа, иллюстрирующая применение аргументов по умолчанию
#include <iostream.h>

void MyMessage(const char *msg, const char *name = "Сергей")
{
        cout << name << " сказал: \"" << msg << "\"" << endl;
}
```

```
int main()
{
	MyMessage(“Здравствуй!”);
	MyMessage(“Привет!”, “Игорь”);
	return 0;
}
```

Листинг 3.

По окончании работы программа выдаст следующие строки:

Сергей сказал “Здравствуй!”

Игорь сказал “Привет!”

Программа маленькая, но интересная. Она начинается с определения функции MyMessage с двумя параметрами: msg и name. Обратите внимание на второй параметр, которому присваивается значение "Сергей". Это и есть то значение по умолчанию, которое примет параметр name, если не получит аргумента при вызове функции. Функция выводит на экран строку текста, используя значения параметров функции.

Теперь сравним строки в функции main. Первый вызов функции MyMessage не передает аргумент для второго параметра, но если вы посмотрите на результат этого вызова, то увидите, что слово «Здравствуй!» произнес Сергей. Во втором вызове MyMessage аргумент передается, и теперь Игорь отвечает на приветствие Сергея.

Аргументы по умолчанию невероятно полезная вещь! Очень часто параметры функции меняются не при каждом вызове и в большинстве случаев оказываются равными одним и тем же величинам. В этом случае имеет смысл сделать эти значения параметров параметрами по умолчанию, что упростит вам запись вызовов функции.

2. Классы памяти и области действия

Область действия (scope rules) переменной – это правила, которые устанавливают, какие данные доступны из данного места программы.

В языке С каждая функция – это отдельный блок программы. Попасть в тело функции нельзя иначе, как через вызов данной функции. В частности, нельзя оператором локального перехода goto перейти в середину другой функции.

С точки зрения области действия переменных различают три типа переменных: глобальные, локальные и формальные параметры. Правила области действия определяют, где каждая из них может применяться.

Локальные переменные – это переменные, объявленные внутри блока, в частности внутри функции. Язык С поддерживает простое правило: переменная может быть объявлена внутри любого блока программы. Локальна переменная доступна внутри блока, в котором она объявлена. Область действия локальной переменной – блок.

Локальная переменная существует пока выполняется блок, в котором эта переменная объявлена. При выходе из блока эта переменная (и ее значение) теряется.

Формальные параметры – это переменные, объявленные при описании функций как ее аргументы. Функции могут иметь некоторое количество параметров, которые используются при вызове функций для передачи значений в тело функции. Формальные параметры могут использоваться в теле функции так же, как локальные переменные, которыми они по сути дела и являются. Область действия формальных параметров – блок, являющийся телом функции.

Глобальные переменные – это переменные, объявленные вне какой-либо функции. В отличие от локальных переменных глобальные переменные могут быть использованы в любом месте программы, но перед их первым использованием они должны быть объявлены. Область действия глобальной переменной – вся программа.

Использование глобальных переменных имеет свои недостатки:

- они занимают память в течение всего времени работы программы;
- использование глобальных переменных делает функции менее общими и затрудняет их использование в других программах;
- использование внешних переменных делает возможным появление ошибок из-за побочных явлений. Эти ошибки, как правило, трудно отыскать.

В языке С есть инструмент, позволяющий управлять ключевыми механизмами использования памяти и создавать мощные и гибкие программы. Этот инструмент – классы памяти. Каждая переменная принадлежит к одному из четырех классов памяти. Эти 4 класса памяти описываются следующими ключевыми словами:

- auto – автоматическая,
- extern – внешняя,
- static – статическая,

- register – регистровая.

Тип памяти указывается модификатором – ключевым словом, стоящим перед спецификацией типа переменной. Например,

static int sum;
register int plus;

Если ключевого слова перед спецификацией типа локальной переменной при ее объявлении нет, то по умолчанию она принадлежит классу auto. Поэтому практически никогда это ключевое слово не используется. В этом просто нет необходимости.

Автоматические переменные (auto) имеют локальную область действия. Они известны только внутри блока, в котором они определены. Другие функции могут использовать то же имя, но это должны быть переменные, относящиеся к разным блокам. Автоматическая переменная создается (т.е. ей отводится место в памяти программы) при входе в блок функции. При выходе из блока автоматическая переменная уничтожается, а область памяти, в которой находилась эта переменная, считается свободной и может использоваться для других целей.

Автоматические переменные хранятся в оперативной памяти машины, точнее, в стеке. Регистровые (register) переменные хранятся в регистрах процессора. Доступ к переменным, хранящимся в регистровой памяти, гораздо быстрее, чем к тем, которые хранятся в оперативной памяти компьютера. В остальном регистровые переменные аналогичны автоматическим переменным. Регистровая память процессора невелика, и если доступных регистров нет, то переменная становится простой автоматической переменной.

Синтаксис регистровой переменной имеет вид:

register int quick;

В большинстве случаев создатели компиляторов предусматривают оптимизацию программ, в частности используя регистровую память для размещения в ней переменных, которые активно используются в программе. Авторы компиляторов фирмы Borland утверждают, что оптимизация компиляторов по использованию регистровых переменных сделана так хорошо, что указание использовать переменную как регистровую может только ухудшить эффективность создаваемого машинного кода. В то же время в опциях интегрированной среды есть возможность задавать способы использования регистровых переменных:

- не использовать вообще;
- использовать только тогда, когда есть ключевое слово register;
- по усмотрению компилятора.

Внешняя переменная (extern) относится к глобальным переменным. Она может быть объявлена как вне, так и внутри тела функции.

Появление ключевого слова extern связано с модульностью языка С, т.е. возможностью составлять многофайловую программу с возможностью раздельной компиляции каждого файла. Когда мы в одном из файлов опишем

вне тела функции глобальную переменную float global то для нее выделится место в памяти в разделе глобальных переменных и констант. Если мы используем эту глобальную переменную в другом файле, то при раздельной компиляции без дополнительного объявления переменной компилятор не будет знать, что это за переменная. Использование объявления extern float global не приводит к выделению памяти, а сообщает компилятору, что такая переменная будет описана в другом файле. И тогда при компоновке программы, состоящей из нескольких файлов, компоновщик будет искать описание этой переменной и связывать ее с использованием в других файлах. Объявление внешней переменной может быть как вне функции, так и внутри функции. Если это же имя без ключевого слова extern будет объявлено внутри функции, то под этим именем будет создана уже другая автоматическая переменная. Можно к объявлению этой переменной добавить ключевое слово auto, чтобы показать, что вы не ошиблись, а намеренно продублировали имя. Объявлений переменной как внешней может быть несколько, в том числе и в одной функции или в одном файле. Описание же переменной должно быть только одно.

При описании статических переменных перед описанием типа ставится ключевое слово static. Область действия локальной статической переменной – вся программа. Место в памяти под локальные статические переменные выделяется в начале работы программы в разделе глобальных и статических переменных. Однако область видимости локальных статических переменных такая же, как и у автоматических. Значение статических переменных сохраняется от одного вызова функции до другого. Локальные статические переменные инициализируются нулем, если не указан другой инициализатор. При этом описание с инициализацией static int count = 10 локальной статической переменной count вызывает однократную инициализацию переменной count при выделении под нее места. При последующих вызовах функции, в которой описана эта переменная, инициализации не происходит. Это позволяет использовать такую переменную например, для счетчика количества вызовов функции.

Можно описать также глобальную (внешнюю) статическую переменную, т.е. описать переменную типа static вне любой функции. Отличие внешней переменной от внешней статической переменной состоит в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная – только функциями того файла, где она описана, причем после ее определения. Все глобальные переменные – и статические, и нестатические – инициализируются нулем, если не предусмотрено другой инициализации.

В таблице 1 приведены область действия и продолжительность существования переменных разных классов памяти.

Таблица 1.

Класс памяти	Ключевое слово	Время существования	Область действия
Автоматический	auto	временно	блок
Регистровый	register	временно	блок
Статический	static	постоянно	блок
локальный	static	постоянно	файл
Статический	extern	постоянно	программа
глобальный			
Внешний			

В программе может быть описано несколько переменных с одним и тем же именем, конечно, в разных блоках.

Перегрузка функций

Перегрузка функций – свойство языка C++, аналогов которому не имеют ни C, ни Pascal, ни BASIC. Эта особенность языка позволяет вам определять функции с одним и тем же именем, но разными списками параметров. Тип возвращаемого функцией значения не является частью того, что называется сигнатурой функции, поскольку C++ позволяет вам игнорировать возвращаемое значение. В этом случае компилятор не сможет различить вызовы одноименных функций с одинаковыми списками аргументов, но разными типами возвращаемых значений, которые игнорируются в этих вызовах. Поэтому такая функция не может быть перегруженной.

Список параметров часто называют *сигнатурой функции.*

Использование аргументов по умолчанию в перегруженных функциях может привести к дублированию сигнатур для некоторых функций, когда используются аргументы по умолчанию. Компилятор C++ в таких случаях выдает сообщение об ошибке.

Не определяйте слишком много перегруженных функций, старайтесь использовать аргументы по умолчанию. Не делайте перегруженными функции, различающиеся по выполняемым ими действиям.

Основное достоинство перегруженных функций – это возможность определять несколько функций с одним и тем же именем, но с разными типами параметров. В листинге 4 представлен исходный код программы, использующей перегруженные функции. Программа выполняет следующие действия:

1. Объявляет переменные типов char, int и double и инициализирует их.
2. Выводит начальные значения этих величин.
3. Вызывает перегруженные функции, которые увеличивают значения этих величин.
4. Выводит вычисленные значения.

```
// Программа C++, иллюстрирующая перегрузку функции
#include <iostream.h>
// версия функции inc для аргумента целого типа
void inc(int &i)
{
	i = i + 1;
}
// версия функции inc для вещественного аргумента двойной точности
void inc(double &x)
{
	x = x + 1;
}
// версия функции inc для символьного аргумента
void inc(char &c)
{
	c = c + 1;
}
int main()
{
	char c = 'A';
	int i = 10;
	double x = 10.2;
// вывод начальных значений
cout << “c = “ << c << endl << “i = “ << i << endl << “x = “ << x << endl;
// вызовы функции inc
	inc(c);
	inc(i);
	inc(x);
// вывод вычисленных значений
cout << “После использования перегруженной функции:” << endl;
cout << “c = “ << c << endl << “i = “ << i << endl << “x = “ << x << endl;
	return 0;
}
```

Листинг 4.

Сеанс работы программы:

c = A

i = 10

x = 10.2

После использования перегруженной функции:

c = B

i = 11

x = 11.2

Программа из листинга 4 объявляет три варианта перегруженной функции inc. Первая версия этой функции имеет ссылочный параметр i целого

типа. Функция увеличивает параметр i на 1. Поскольку параметр i является ссылочным параметром, то в результате вызова функции inc(int&) переданный функции аргумент будет изменен, хотя он и находится за пределами области действия функции. Во второй версии функции inc параметр-ссылка x имеет тип double. Функция увеличивает значение параметра x на 1. Поскольку параметр x является ссылочным параметром, то в результате вызова функции inc(double &) переданный функции аргумент будет изменен, хотя :он и находится за пределами области действия функции. Третья версия функции inc, имеющая параметром ссылку на переменную типа char, работает аналогичным образом.

В функции main объявляются переменные c, i и x типов char, int и double соответственно. Затем функция присваивает переменным c, i и x значения 'A', 10 и 10.2. Операторы выводят начальные значения переменных c, i и x. Далее производится вызов перегруженной функции. Вначале вызывается функция inc(char &), поскольку передаваемый аргумент имеет тип char. Затем вызывается функция inc(int i), так как передаваемый аргумент имеет тип int. Далее вызывается функция inc(double x), поскольку передаваемый аргумент имеет тип double. Дальнейшие операторы выводят результаты работы программы.

3. Рекурсия в языке C++

Многие задачи могут быть решены, если их разбить на простые и подобные друг другу задачи. Рекурсия – один из способов такого решения.

Рекурсивные функции – это функции, вызывающие сами себя. Стоит помнить, что на каждый вызов функции расходуются ресурсы системы, которые ограничены, и чтобы у вас не возникла проблема нехватки памяти, вам следует ограничивать количество рекурсивных вызовов. Поэтому каждая рекурсивная функция должна содержать в себе проверку условия окончания рекурсии.

Вероятно, самая простая и известная рекурсивная функция – это *функция факториала.* Факториал целого числа N есть произведение всех целых от 1 до N. Для обозначения этой функции в математике применяется восклицательный знак (!).

Математически эта функция записывается как:

N!=1*2*3* ... * (N-2) * (N-1) * N

Рекурсивный вариант получения того же результата:

N! = N * (N-1)!

(N-1)! = (N-1) * (N-2) !

(N-2)! = (N-2) * (N-3) !

…

2! = 2 * 1

1! = 1

Многие задачи, кроме рекурсивного решения, могут иметь и какое-то другое. Функция факториала может быть вычислена как рекурсивно, так и нерекурсивно. Однако в некоторых случаях рекурсивное решение выглядит очень элегантно.

Обычный повод применения рекурсии – сэкономить на размере программы: исходный текст программы получается короче, легче читается, и исполняемый модуль получается меньшего размера. Довод против рекурсии – это замедление работы программы. Рекурсивные функции обычно, но не всегда, работают медленнее, чем их нерекурсивные варианты. И когда вы можете решить задачу двумя способами, вам следует взвесить все за и против, а еще лучше, написать два варианта функции, прежде чем выбрать один из способов.

Рекурсивная функция должна содержать в себе проверку окончания рекурсии. Не применяйте рекурсию, если есть альтернативный, более эффективный метод вычислений.

Рассмотрим пример вычисления факториала с использованием рекурсии, представленный в листинге 5. Программа предлагает ввести целое число и затем выводит значение факториала числа.

```
// C++ пример программы, использующей рекурсивную функцию.
#include <iostream.h>
const int MIN = 1;
const int MAX = 30;
double factorial(double f)
{
        if(f > 1)
                return f * factorial(f -1);
        else
        return f;
}
int main()
{
        int x;
        do
        {
        cout << “Введите число между ” << MIN << “ и ” << MAX << “ :  ”;
        cin >> x;
        } while(x < MIN || x > MAX);
        cout << x << “! = ” << factorial(x) << endl;
        return 0;
}
```

Листинг 5.

Программа из листинга 5 начинает свою работу с определения констант MIN и MAX. Первое ограничение необходимо, потому что для отрицательных чисел функция факториала не определена. Второе ограничение вводится для того, чтобы не задать уж слишком большое число. Функция factorial имеет один аргумент типа double и возвращает значение того же типа. Вся рекурсия умещается в одной строке. Здесь вызывается функция factorial с аргументом f - 1. В предыдущей строке находится оператор контроля, останавливающий рекурсию. Как только вы дойдете до значения аргумента, равного 1, вы прекращаете рекурсивные вызовы, поскольку рекурсия закончена. Если бы этой проверки не было, вы бы вызывали функцию снова и снова, пока не подвесили бы программу, исчерпав в конце концов количество памяти потраченной на слишком большое число вызовов функции. В операторе return возвращается результат вычислений.

Функция main содержит простой цикл, в котором вводится число и проверяется, попало ли оно в нужный интервал. После этого производится вызов функции factorial, с введенным числом в качестве аргумента. Обратите внимание на то, что пользователь вводит число целого типа, в то время как параметр функции – вещественная переменная двойной точности. Это еще один пример автоматического преобразования целой величины в вещественную. Обратный вариант автоматического преобразования типов (вещественного в целый) невозможен.

Заключение

Сегодня вам были представлены функции языка C++ и их использование при работе с данными.

Общая форма определения функции:

возвращаемыйТип имяФункции(<список параметров>)

{

< объявление данных >

< тело функции>

return возвращаемоеЗначение;

}

Если функция вызывается до своего определения, обязательно должен быть задан прототип функции. Общая форма объявления функции:

возврТип имяФункции(<список параметров>);

При объявлении функции имена параметров могут быть опущены.

Передача аргумента по ссылке позволяет функции изменять значение переданного аргумента и экономит память, так как при этом не создается локальная копия аргумента.

Ключевое слово const предохраняет передаваемые по ссылке аргументы от случайного изменения.

Локальные переменные превращают функцию в мощный независимый инструмент языка. Объявление локальных переменных подобно объявлению глобальных переменных.

Ключевое слово static позволяет объявить переменную как статическую. Статическая переменная является локальной переменной, но сохраняет свое значение между вызовами функции. Обычно статические переменные инициализируются. Начальные значения присваиваются перед первым вызовом функции, в которой определена статическая переменная.

Выход из функции осуществляется по оператору return. Void-функции могут не возвращать значения.

Используя аргументы по умолчанию для некоторых параметров, при вызове функции вы можете не задавать аргументы для этих параметров, тогда им автоматически будут присваиваться значения по умолчанию.

Рекурсивными называются функции, которые вызывают сами себя. Количество рекурсивных вызовов должно быть ограничено, чтобы не столкнуться с проблемой нехватки памяти. По этой причине каждая рекурсивная функция должна выполнять проверку условия на окончание рекурсии.

Перегрузка функций позволяет вам иметь несколько функций с одним именем, но с разными списками аргументов (список аргументов еще называется сигнатурой функции). Тип возвращаемого функцией значения не является частью сигнатуры.