4

Функции

- Создание функций
- Расширение возможностей функций
- Создание заголовочных файлов
- Понятие области видимости

На этом уроке с помощью нашего неутомимого гимнаста Сэла мы научимся создавать простейшие функции. Функцией можно сделать любое упражнение, которое умеет делать Сэл. Например, если один раз научить Сэла выполнять подскок, то учить его снова уже не придется. Все, что вам нужно будет сделать, — это создать функцию, объясняющую, как нужно выполнять подскок.

В дальнейшем вы убедитесь, насколько полезны функции. Вы также научитесь сохранять нужные функции в заголовочных файлах, чтобы их можно было использовать снова в любой создаваемой вами программе.

В конце урока будет введено понятие области видимости. Поскольку компилятор трактует каждую функцию как отдельную программную сущность, то имена, присвоенные всем ее переменным и объектам, действительны только внутри области видимости данной функции. На самом деле это очень хорошо, так как позволяет не запоминать все идентификаторы программы. Тем не менее, чтобы избежать конфликтов имен, вы должны знать несколько правил области видимости.

Понятие функции

Иногда определенное действие состоит из группы отдельных команд. В этом случае гораздо удобнее обратиться сразу ко всей группе, а не выписывать все необходимые команды по отдельности.

Рассмотрим следующую группу команд:

```
Sal.up();
Sal.left();
Sal.up();
Sal.right();
Sal.up();
Sal.ready();
```

Очевидно, что все эти действия определяют одно упражнение. Если мы заставим компьютер отождествить эту группу действий с одним именем, например *влево-вправо*, то, дав команду *влево-вправо,* мы на самом деле попросим Сэла выполнить некоторую последовательность действий. Другими словами, всю описанную выше последовательность инструкций мы определяем как единую функцию, которая описывает, как выполняется эта последовательность. Аналогичный элемент в других языках программирования называют процедурой, или подпрограммой. Хотя в терминах C++ такие части программы называют *функцией* (function), по-прежнему вполне применим и исходный термин — *процедура* (procedure).

> Функция представляет собой группу инструкций, которой назначено некоторое имя. Все эти инструкции могут быть выполнены посредством вызова функции по ее имени.

Благодаря функциям компьютер становится сообразительнее. Однажды определив функцию, вам больше не понадобится повторять все входящие в функцию инструкции. Компьютер уже будет знать, как они выполняются! Кроме того, программа становится понятнее.

> Как и люди, компьютер не может запомнить функцию раз и навсегда. Поэтому вам необходимо включать код вашей функции в каждую программу, где вы собираетесь ее использовать.

Вы, конечно, заметили, что последовательность движений Сэла составляет единое упражнение. Инструктор по гимнастике может дать отдельное имя и, например, такой последовательности движений:

Позиция готовности

Прыжок

Позиция готовности

Инструкторы называют эту последовательность действий *подскоком*. Таким образом, когда инструктор хочет заставить группу выполнить подскок, то уже не объясняет всю последовательность действий.

Например, однажды показав, как выполняются упражнения *подскок* и *влево-вправо*, инструктор будет отдавать свои команды следующим образом:

Подскок

Влево-вправо

Подскок

Обратите внимание, насколько упростились команды. Вместо того чтобы давать подробное описание всей последовательности движений, инструктор теперь просто называет имя нужного упражнения.

Ту же стратегию можно применить и на вашем компьютере — можно сгруппировать несколько инструкций и присвоить им единое имя. Теперь, чтобы выполнить функцию, вам достаточно просто указать ее имя, а компьютер выполнит каждую входящую в нее инструкцию.

Робот тоже может научиться работать с функциями. Например, можно создать функцию рисования линии заданного размера.

Создание функции

Попробуем создать функцию. Поскольку прежде, чем писать любую программу (в том числе функцию), нужно точно понять, что мы хотим от нее получить, для начала сделаем описание программы.

Функция подскока:

Команда Сэлу подпрыгнуть

Команда Сэлу занять исходное положение

Окончание функции подскока.

На языке C++ это будет выглядеть следующим образом:

```
void JumpJack()
{
    Sal.up();
    Sal.ready();
}
```

Имя функции

Компьютеру нужно объяснить, во-первых, действие функции, а во-вторых, ее имя. По-русски это звучало бы так: *это функция выполнения подскока*. Эта фраза объясняет, что функция выполнения подскока состоит из некоторой последовательности инструкций.

Границы функции

Где первая и где последняя инструкция в принадлежащей функции последовательности инструкций?

Компьютеру нужен точный ответ на этот вопрос. Чтобы выделить первую и последнюю инструкции, вокруг интересующей нас последовательности действий

ставятся открывающая и закрывающая фигурные скобки {}. Функция начинается с открывающей фигурной скобки и заканчивается закрывающей. Внутри этих скобок инструкции обычно сдвигают вправо. Для компьютера это не важно, но для нас это очень полезно и является хорошим стилем программирования.

С технической точки зрения вся функция рассматривается как единая составная инструкция (что это такое, вы узнаете позже). Но если в конце строки с именем функции вы ставите точку с запятой, то тем самым даете понять компилятору, что ваша инструкция на этом заканчивается (еще до начала тела функции).

В программе перед именем функции ставится слово void, о назначении которого будет рассказано ниже, а за именем функции следуют круглые скобки (). В конце строки с именем функции точка с запятой не ставится.

Рассмотрим программу c2jmpjck.cpp:

```
//              c2jmpjck.cpp
// Составлена Paulo Franca
// Последняя доработка 05.01.96
// В этой программе для выполнения упражнений Сэл использует
// функцию JumpJack
#include "franca.h"  // Вставка программ
                     // из программного обеспечения книги
athlete Sal;         // Создание объекта Sal класса athlete
void JumpJack()      // Начало функции JumpJack
{
    Sal.up();        // Прыжок
    Sal.ready();     // Исходное положение
}                    // Конец функции JumpJack
void mainprog()      // Начало основной программы
{
    Sal.ready();     // Исходное положение
    JumpJack();      // Подскок!
}                    // Конец основной программы
```

В программу входят коды функций JumpJack() и mainprog(). Заметим, что mainprog — это имя функции, в которой вы объясняете компьютеру, что на самом деле вы от него хотите. Эта функция в принципе аналогична всем остальным, за исключением того, что должна присутствовать в каждой вашей программе. В функции mainprog() можно вызывать другие функции. В нашем примере это была функция JumpJack().

Функция должна быть описана в программе до ее первого использования.

Если мы хотим, чтобы Сэл выполнил еще несколько подскоков, нужно просто включить в программу две дополнительные инструкции. Тогда функция mainprog() будет выглядеть следующим образом:

```
void mainprog()    // Начало основной программы действий
{
    Sal.ready();   // Исходное положение
    JumpJack();    // Подскок!
    JumpJack();    // Еще подскок!
    JumpJack();    // И еще подскок!
}                  // Конец основной программы действий
```

Запомните, функция — это, по сути, набор действий, которому вы присвоили собственное имя. Благодаря функциям ваши программы становятся понятнее, поскольку отпадает необходимость снова и снова описывать всю последовательность шагов. Кроме того, перечисленные в функции действия могут выполнять самые разные объекты.

Управление несколькими объектами

Предположим, что у Сэла есть подружка Салли, тоже занимающаяся гимнастикой. Включив в программу следующее объявление, мы легко получаем сразу двух гимнастов:

```
athlete Sal, Sally;
```

Это объявление сообщает компьютеру, что теперь имеются два объекта типа athlete, и, следовательно, мы можем вывести на экран оба этих объекта:

```
Sal.ready();
Sally.ready();
```

Как заставить Сэла выполнить подскок, мы уже знаем. А что, если нам захочется, чтобы то же самое сделала Салли? В этом случае бесполезно вызывать функцию JumpJack(), поскольку все перечисленные в ней упражнения предназначены для Сэла. Что же делать? Есть два простых решения:

- Можно специально для Салли создать еще одну функцию JumpJack().
- Можно просто перечислить для Салли все действия, входящие в подскок.

Оба этих решения некорректны. Мы ведь уже описывали, как выполняется подскок, и это описание одинаково для Сэла, Салли и для любого другого гимнаста. (В конце концов, опытные инструкторы не объясняют *каждому* члену группы, как выполняется то или иное упражнение, не правда ли?)

Аргументы функций

Было бы здорово переделать функцию так, чтобы она описывала выполнение подскока *любому* гимнасту. Это должно быть описание *способа выполнения упражнения*, который не зависит от того, *кто* именно его выполняет.

В каком-то смысле это напоминает театральную постановку. Автор пишет пьесу, в которую входят те или иные персонажи. При этом он, как правило, не знает, кто конкретно будет играть их роли. Одну и ту же роль могут играть разные актеры, но актер и роль не привязаны друг к другу раз и навсегда, и каждый актер играет за свою жизнь множество разных ролей.

Аргументы и параметры

Когда мы создаем функцию, то обычно не знаем заранее, какие объекты будут ее аргументами. Следовательно, мы говорим о фиктивных или подразумеваемых объектах, которые называют *параметрами* (parameters). При вызове функции параметр заменяется реальным объектом. Реальные объекты, которые при выполнении функции используются вместо параметров, называют *аргументами* (arguments). Все прямо как в театре.

Итак, мы хотим, чтобы функция JumpJack() программы c2jmpjck.cpp могла управлять действиями как Сэла, так и Салли (или любого другого гимнаста, который может появиться в дальнейшем). Этого можно добиться, заставив функцию работать не с конкретным, а с любым объектом типа athlete.

1. Объясните компьютеру, как заставить фиктивного гимнаста выполнить упражнение. Вы можете дать этому гимнасту какое-то имя, например *некто* (параметр somebody). Для этого *некто* включите в функцию все знакомые нам инструкции выполнения упражнения. Но у нас нет гимнаста с именем *некто*, а есть только Сэл и Салли. Следовательно, *некто* — это и есть параметр.
2. Научив гимнаста *некто* выполнять подскок, заставьте Салли (или Сэла) сыграть роль этого *некто* в вашей функции. Салли — это объект, реально существующий в вашей программе (так же, как актеры существуют в реальной жизни), и он может занять место фиктивного *некто*.

Посмотрим, как это сработает в программе c2jmpbdy.cpp:

```
//                  c2jmpbdy.cpp
//
#include "franca.h"
athlete Sal, Sally;
void JmpJack(athlete somebody)
{
    somebody.up();
    somebody.ready();
}
void mainprog()
{
    Sal.ready();
    JmpJack(Sal);
    Sally ready();
    JmpJack(Sally);
}
```

В функции JmpJack() мы использовали объект sombody, которого, как мы знаем, на самом деле не существует. Тем не менее мы объяснили ему, какие действия надо выполнить, чтобы совершить подскок. Если вы посмотрите на основную часть программы, то заметите, что функция JumpJack() упоминалась в ней дважды:

```
JmpJack(Sal);
JmpJack(Sally);
```

В первом случае внутри фигурных скобок находится Sal. Что это означает?

Мы выполняем функцию JumpJack(), но вместо фиктивного параметра somebody в качестве аргумента указываем реальный объект Sal. Все движения будут выполнены реальным гимнастом Сэлом, а не каким-то гипотетическим *некто*. Во втором случае в качестве аргумента указан объект Sally. После выполнения программы экран вашего компьютера будет выглядеть так, как показано на рис. 4.1.

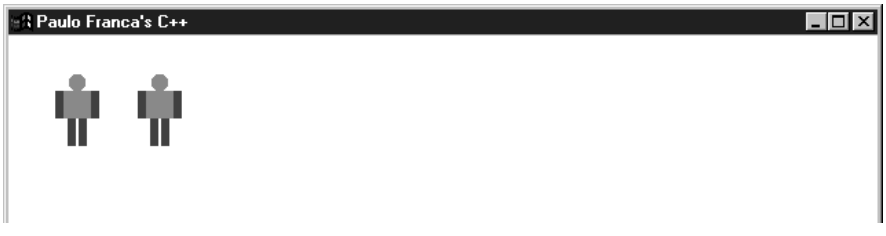

Рис. 4.1. Результат выполнения программы c2jmpbdy.cpp

Используя новый метод, попытаемся поменять местами аргументы, поочередно объявляя Сэла и Салли:

- Пусть гимнасты займут исходные позиции и объявят свои имена. Затем вызовем функцию JmpJack() с каждым гимнастом в качестве аргумента. Посмотрите на результат.
  - Для этого нужно, чтобы перед вызовом функции JmpJack() Сэл сказал «Сэл», а Салли сказала «Салли».
- Теперь вызовем функцию JmpJack() в обратном порядке: сначала с Салли, а затем с Сэлом. Посмотрите на результат.
  - Другими словами, сначала записываем инструкцию JmpJack(Sally);, а затем — инструкцию JmpJack(Sal);.

Аргументы по умолчанию

Иногда бывает удобно определить для параметров значения по умолчанию. Особенно это полезно, когда в качестве аргумента чаще других приходится использовать один и тот же объект (аргумент по умолчанию).

Пусть, например, мы хотим, чтобы в большинстве случаев упражнение выполнял Сэл и только иногда Салли. В определении функции мы можем задать аргумент

по умолчанию, присвоив параметру нужное значение. Например, если мы следующим образом изменим заголовок функции JmpJack(), то, если при вызове функции никого другого не задано, Сэл будет выступать в роли гимнаста по умолчанию («штатного» гимнаста):

```
void JmpJack(athlete sombody=Sal)
```

Таким образом, инструкция

```
JmpJack();
```

приведет к тому же результату, что и инструкция

```
JmpJack(Sal);
```

Это дает нам возможность при вызове функции просто опускать наиболее часто встречающееся значение ее аргумента. В случае необходимости мы можем включить туда нужный нам аргумент. Например, если мы хотим использовать Салли, то можем сделать это, вызвав функцию следующим образом:

```
JmpJack(Sally);
```

> Когда вы определяете аргумент по умолчанию, то последний должен быть предварительно определен и известен функции (см. раздел об области видимости).

Функции нескольких аргументов

Можно создавать функции с несколькими аргументами. Например, мы можем описать функцию выполнения подскока, в качестве аргументов которой указаны оба гимнаста. В C++ в одной программе возможно сосуществование нескольких функций с одним именем, но с различным числом или типом (классом) аргументов.

Давайте создадим новую функцию JmpJack(), описывающую выполнение упражнения обоими гимнастами. Очевидно, что для этого можно просто вызвать нашу прежнюю функцию JmpJack() для каждого из гимнастов:

```
void JmpJack(athlete first, athlete second)
{
    JmpJack(fist);
    JmpJack(second);
}
```

Гимнасты будут упражняться по очереди. Можно, конечно, и так, но как заставить их работать одновременно? Фактически полностью одновременное выполнение упражнений невозможно, поскольку компьютер выполняет инструкции последовательно. Мы попытаемся контролировать время, в течение которого гимнаст находится в каждой позиции.

Чтобы создать иллюзию одновременного выполнения упражнений, мы можем заставить первого гимнаста занять исходное положение за 0 секунд, а затем заставить второго сделать то же самое за 1 секунду. Аналогично поступим с другими движениями. Сейчас большинство компьютеров выполняет операции так быстро, что вы вряд ли их заметите. Но картинка, которая остается на экране достаточно продолжительное время (1 секунду), четко отпечатается в вашем сознании.

Ниже представлен алгоритм работы функции.

Функция JmpJack() для выполнения одновременного подскока двух гимнастов:

Первый гимнаст остается в *исходном положении* 0 секунд.

Второй гимнаст остается в *исходном положении* 1 секунду.

Первый гимнаст находится в *прыжке* 0 секунд.

Второй гимнаст находится в *прыжке* 1 секунду.

Первый гимнаст остается в *исходном положении* 0 секунд.

Второй гимнаст остается в *исходном положении* 1 секунду.

Сможете ли вы самостоятельно написать функцию и оценить результат? Начните со следующей инструкции:

```
void JmpJack(athlete fist, athlete second)
```

Значения и ссылки

Обычно когда вы вызываете функцию с аргументом, то функция использует копию этого аргумента. Другими словами, когда выполняется следующая инструкция, компьютер создает копию объекта Sally и затем использует ее в функции JmpJack():

```
JmpJack(Sally);
```

С самим же объектом Sally ничего не происходит, вне зависимости от того, какие операции выполняются в функции. В C++ при вызове функции это обычное явление. Вы можете быть уверены, что функция никак не повлияет на исходный объект.

В рассмотренном нами примере разница между оригиналом и копией не заметна. Вместо исходного гимнаста создается другой, который действует согласно записанным в функции инструкциям. Глядя на экран, вы не сможете сказать, кто выполняет упражнения — истинный гимнаст или его копия.

Увидеть, к чему приводит использование копий, можно с помощью объекта типа Clock (часы). В следующем примере мы сначала создадим объект типа Clock, который назовем timer (таймер), а затем вызовем функцию перезапуска таймера. Вместо того чтобы перезапустить оригинальные часы, функция перезапустит их копию.

Наблюдать это можно с помощью информационных рамок, показывающих время до начала, в процессе и после выполнения функции. Поэтому создадим три рамки

и пометим их соответственно Before:, During: и After:. Проблему иллюстрирует программа c2clkcpy.cpp:

```
#include "franca.h"             // c2clkcpy.cpp
// Эта программа демонстрирует использование в функциях копий аргументов
Box before ("Before:"), during ("During:"), after ("After:");
void zero(Clock clone)
{
    clone.reset();              // Перезапуск часов
    during.say(clone.time());   // Вывод времени
}
void mainprog()
{
    Clock timer;
    timer.wait(1);              // Пауза в 1 секунду
    before.say(timer.time());   // Вывод времени
    zero(timer);                // Вызов функции перезапуска
    after.say(timer.time());    // Вывод времени
}
```

Если вы запустите эту программу, то увидите на экране три числа. Первое число показывает, сколько секунд прошло с момента запуска программы. Из-за сообщения wait(1) оно должно равняться примерно 1.

Второе число показывает, сколько секунд прошло после того, как часы были перезапущены функцией zero(). Это число должно равняться 0.

Третье число показывает, сколько времени прошло с того момента, как часы были перезапущены последний раз. Вы, наверное, полагаете, что это число должно быть примерно равно нулю, поскольку, будучи аргументом функции, объект timer был обнулен. Но это не так! На самом деле число, которое вы увидите, будет равно 1 или чуточку больше. Это говорит о том, что был обнулен не объект timer, а его копия. Убедитесь в этом на рис. 4.2.

Рис. 4.2. Результат выполнения программы c2clkcpy.cpp

Защита ваших объектов от ваших же функций на первый взгляд кажется прекрасной идеей, но как быть, если вам нужно работать с исходным объектом?

Передача по ссылке или по значению

C++ позволяет показать, что функция предназначена для работы с оригиналом объекта, а не с его копией. В этом случае говорят, что параметр передается не по *значению* (value), а по *ссылке* (reference). Передача по ссылке выполняется очень просто. Отметьте, какие параметры вы хотите передать по ссылке (передача по значению происходит по умолчанию). В C++ для этого перед именем параметра ставится символ амперсанда &. В нашем примере измененная функция будет иметь следующий вид:

```
void zero(Clock &clone)
{
    clone.reset();              // Перезапуск часов
    during.say(clone.time());  // Вывод времени
}
```

Вот и все изменения! Все инструкции в функции остались прежними. Благодаря символу ссылки (&) можно быть уверенным, что все операции будут выполнены над исходным объектом. Теперь можете изменить программу и оценить результат.

> Если у вас имеется более одного параметра, то символ & нужно указать перед всеми параметрами, которые вы хотите передавать по ссылке.

Применение ссылок

Очевидно, что ссылки применяются, когда нужно изменить исходный аргумент функции. Бывают и другие ситуации, в которых удобно использовать ссылки.

На создание копии объекта компьютер затрачивает определенное время. В большинстве случаев это несущественно. Однако чем сложнее объект, тем больше времени требуется на его копирование. Например, объекты типа athlete являются достаточно сложными. Если вы уверены, что не нуждаетесь в защите вашего оригинала от модификации, а сложность вашего объекта достаточно велика, вы также можете воспользоваться передачей по ссылке.

Обучение объекта

Вы уже, наверное, обратили внимание, что вызов функции похож на передачу сообщения объекту. В обоих случаях выполняется некоторая последовательность действий. Когда вы посылаете сообщение объекту, вы вызываете функцию, связанную с объектами данного класса. Такие функции называются функциями-членами. Более подробно о функциях-членах вы узнаете на уроках 16 и 17.

Когда создавался класс athletes, он предназначался для обработки сообщений ready, up, left и right. Это достигалось созданием функции-члена для каждого из этих действий. Я мог бы включить сюда и функцию для подскока, но не захотел лишить вас удовольствия создать ее самостоятельно.

Функции-члены и функции-не-члены

У объекта имеются функции обработки всех предназначенных для него сообщений. Например, объекты класса athlets имеют следующие функции:

```
ready()
up()
left()
right()
say()
```

Эти функции являются частями объекта и не могут выполняться независимо от него. Говоря формальным языком, они представляют собой *функции-члены* класса athlets. Функции-члены могут быть определены только после определения соответствующего класса. Мы научимся это делать на следующем уроке. С другой стороны, функция JmpJack() не относится ни к одному из классов, и именно поэтому мы смогли ее создать. Функции, которые не являются частями какого бы то ни было класса, называются *функциями-не-членами*.

Заголовочные файлы

Если одну из функций вы хотели бы использовать сразу в нескольких программах, то вам придется копировать ее в каждую новую программу. Это, согласитесь, не очень удобно. Было бы лучше, если бы можно было научить компьютер самостоятельно выбирать и копировать необходимую вам функцию.

Функцию (как и любую другую программу) можно сохранить в файле. Затем, чтобы сделать копию функции, можно использовать предназначенную для этого директиву #include. Я подготовил для вас несколько функций, но если вы решите копировать их по одной, то быстро откажетесь от этого занятия. Поэтому все свои функции я разместил в заголовочном файле franca.h. Если вы используете следующую инструкцию, то просите компьютер найти файл franca.h и скопировать его в вашу программу:

```
#include "franca.h"
```

То же самое вы можете делать со своими функциями и программами.

Если функцию JumpJack() сохранить на диске, например в файле jumpjack.cpp, то вы можете скопировать эту функцию в программу с помощью следующей строки:

```
#include "jumpjack.cpp"
```

Теперь в своей программе вы не увидите функции JumpJack() (как вы не можете видеть функции из файла franca.h). Тем не менее непосредственно перед тем, как компилятор начнет транслировать вашу программу на машинный

язык, копия запрашиваемой программы появится там, где находится директива #include.

Чтобы компилятор мог найти файл, который необходимо включить в программу, этот файл должен находиться в том же каталоге, что и ваша программа (например, каталог C:\franca). Если это не так, нужно указать полный путь к файлу:

```
#include "c:\franca\jumpjack.cpp"
```

Когда вы хотите включить в программу файлы, находящиеся в ваших каталогах, то имя файла заключается в двойные кавычки. В других случаях вам может понадобиться включить в программу файлы, находящиеся в каталогах компилятора (зачем это делать, вы узнаете позже). Тогда вместо двойных кавычек имя файла за-ключается в угловые скобки. Например:

```
#include <math.h>;
#include <iostream.h>;
```

Директивы

Директива (directive) — это инструкция компилятору. Она объясняет, что вы хотите сделать, перед тем как ваша программа будет откомпилирована. Директивы не относятся к языку C++, скорее их можно отнести к компилятору.

Компилятор распознает директиву по знаку фунта (#) перед первым символом строки. Имеется несколько директив компилятора, но нас будет интересовать только директива #include.

Создание заголовочных файлов

Файлы, содержащие части программ, предназначенные для копирования в другие программы, называются *заголовочными файлами* (header file) и обозначаются расширением .h (вместо .cpp). В программу можно включать и файлы с расширением .cpp, но если хотите работать профессионально, то должны сохранять файлы с расширением .h. Это делается с помощью команды File ▸ Store As или File ▸ Save As в среде разработки приложений вашего компилятора. Наберите имя файла и после него символы .h (см. также дополнительный материал в конце этого урока).

Самостоятельная практика

- Сделайте заголовочный файл из функции JmpJack().
- Напишите программу, в которой используется функция из заголовочного файла jumpjack.h.

Область видимости

Если я скажу вам: «Мне нравится Наполеон», то вы, возможно, причислите меня к поклонникам великого императора Франции. Однако если в данный момент вы думаете о покупке торта, то, скорее всего, решите, что я имею в виду одноименный торт.

Что-то подобное может произойти, если не определить нашу *область видимости* (scope). Вам не понадобится каждый раз объяснять, имеете ли вы в виду императора или торт, если уверены, что употребляете слово в правильном контексте — области видимости, — продиктованном темой беседы.

Область видимости используется в C++ и других языках программирования и позволяет вводить различные объекты с одинаковыми именами — при условии, что они находятся в разных областях видимости.

Если объект определен внутри функции, то он недоступен вне ее, то есть объект *локален* по отношению к функции. Если же объект определен вне всех функций, то он доступен для всех функций, определенных после него. Такой объект называется *глобальным*.

В программе c2jmpjck.cpp имеется глобальный объект Sal. Функция JmpJack() просто использовала этот глобальный объект для выполнения упражнения.

В последней версии программы c2jmpbdy.cpp имеются два глобальных объекта Sal и Sally. Функция JmpJack() имеет параметр somebody и, таким образом, в качестве аргументов ей доступны оба объекта. Объекты Sal и Sally по-прежнему остаются глобальными, но внутри функции mainprog() теперь можно определить локальные объекты Sal и Sally.

Хотя в глобальных объектах нет ничего плохого, все же лучше их избегать. Ниже приведена программа c2jmbd1.cpp, представляющая собой улучшенный вариант программы c2jmpbdy.cpp.

```
//                  c2jmbd1.cpp
//
// В этой программе для выполнения упражнений используется
// функция JumpJack() с объектом класса athlete в качестве параметра
#include "franca.h"
void JmpJack(athlete somebody)
{
    somebody.up( );
    somebody.ready( );
}
void mainprog()
{
    athlete Sal, Sally;
    Sal.ready();
    JmpJack(Sal);
    Sally.ready();
    JmpJack(Sally);
}
```

Обратите внимание, что теперь объекты Sal и Sally известны только внутри функции mainprog(). И это правильно, поскольку другим функциям они не нужны.

Повторение имен локальных объектов

Рассмотрим несколько измененный вариант этой программы. Единственное отличие состоит в имени параметра функции JmpJack(). Вместо параметра somebody используем параметр Sal.

В следующем примере задается область видимости объекта Sal:

```
void JmpJack(athlete Sal)
{
    Sal.up();
    Sal.ready();
}
```

А здесь задается область видимости объектов Sal и Sally:

```
void mainprog()
{
    athlete Sal, Sally;
    Sal.ready();
    JmpJack(Sal);
    Sally.ready();
    JmpJack(Sally);
}
```

Вообще-то использовать одно и то же имя для обозначения разных объектов не следует. Но даже если вы забудете об этом, компьютер не растеряется. Внутри функции JmpJack() объект Sal будет обозначать параметр, который должен заменяться копией аргумента. Этот объект сначала играет роль объекта Sal, объявленного в функции mainprog(), а затем — роль объекта Sally.

В этом и заключается основная польза области видимости. Вам не нужно помнить об именах объектов всех своих функций. Каждое имя действует только внутри своей области видимости.

Ниже показан еще один вариант программы (тоже не слишком удачный). В этом случае одно и то же имя Sal имеют глобальный и локальный объекты. Внутри функции JmpJack() области видимости обоих объектов перекрываются, но программа тоже будет работать правильно, поскольку локальный объект имеет приоритет над глобальным. Однако, создавая программу, в которой разные объекты имеют перекрывающиеся области видимости, вы можете легко запутаться, поэтому использовать одно и то же имя для обозначения разных объектов не следует.

В следующей инструкции задается область видимости объекта Sal (глобальная):

```
athlete Sal;
```

А здесь области видимости глобального и локального объектов Sal перекрываются:

```
void JmpJack(athlete Sal)
{
    Sal.up();
    Sal.ready();
}
```

В последнем примере показана область видимости объектов Sal и Sally:

```
void mainprog()
{
    athlete Sally;
    Sal.ready();
    JmpJack(Sal);
    Sally.ready();
    JmpJack(Sally);
}
```

Когда области видимости двух объектов или переменных, имеющих одинаковое имя, перекрываются, то используется более локальный объект. В одной области видимости нельзя объявлять объекты с одинаковыми именами.

Обычно новички забывают, что аргумент функции уже был объявлен в списке ее параметров. Например, в приведенном ниже примере гимнаст somebody объявлен в списке параметров. Поэтому объявлять этот объект снова нельзя:

```
void JmpJack(athlete sombody)
{
    athlete sombody;    // Неправильно! Объект somebody уже существует!
    ...
```

Примеры области видимости

В следующем примере показано неправильное объявление двух объектов с одинаковым именем. Первое объявление объекта Sal находится внутри функции mainprog(). Область видимости функции ограничивается закрывающей скобкой в конце списка ее инструкций. В этой области объявлять еще один объект Sal нельзя.

```
void mainprog()
{
    athlete Sal;
    jmpjack(Sal);
    athlete Sal;  // Ошибка! Первое объявление объекта Sal
    ...           // находится в той же области видимости
}
```

Как будет видно в дальнейшем, другие конструкции C++ также ограничиваются фигурными скобками. В этом случае каждая пара открывающих и закрывающих скобок задает некоторую новую область видимости. Ниже приведен правильный пример. Второе объявление объекта Sal находится внутри новой области видимости. Этот новый объект будет существовать только во внутренней области видимости. Во всех обращениях к объекту Sal внутри нее вместо определенного первоначально объекта используется новый объект. Во всех же обращениях к объекту Sal до или после внутренних скобок используется исходный объект.

```
void mainprog()
{
    athlete Sal;
    Sal.ready()         // Использование первого объекта Sal
    {                   // Задание новой области видимости
        athlete Sal;
        athlete Sally;
        Sal.up();       // Использование нового объекта Sal
        Sally.left( );
    }                   // Конец внутренней области видимости
    Sal.right();        // Новое использование первого объекта Sal
}
```

В приведенном выше примере во внутренней области видимости был объявлен еще один объект Sally. Поэтому все обращения к объекту Sally возможны только внутри этой области видимости (напоминаем, что область видимости ограничена парой фигурных скобок). Все обращения к объекту Sally до внутренней открывающей скобки или после внутренней закрывающей скобки ошибочны.

Что нового мы узнали?

На этом уроке мы научились:

- ✓ Писать функции.
- ✓ Использовать аргументы функций.
- ✓ Передавать аргументы по значению и по ссылке.
- ✓ Создавать заголовочные файлы.
- ✓ Задавать области видимости.

5

Числа

- Хранение в переменных целых, чисел с плавающей точкой и алфавитно-цифровых символов
- Работа с числовыми переменными
- Ввод и вывод значений

В некоторых ситуациях в программах приходится использовать числа. В частности, с помощью чисел учитывают количество повторений, засекают время, а также представляют некоторые глобальные значения.

Как отмечалось в части I, *переменными* называют простые объекты. Переменные предназначены для хранения чисел.

> В названии **переменная** нашел отражение тот факт, что хранящееся в переменной значение меняется в процессе работы программы.

Числа и числовые переменные

Программа различает переменные по их именам. Правила работы с именами переменных те же, что и с именами объектов. *Переменная* — это ячейка (или набор ячеек) в памяти компьютера, где можно хранить число. Это число, в свою очередь, может обозначать буквы, цвета и вообще все, что мы захотим. В C++ имеются три основных типа переменных.

- *Целые*. В целых числах отсутствует дробная часть. Есть два типа целых — это типы int и long.
  - В переменной типа int могут храниться целые числа, значения которых лежат в интервале от —32768 до +32767.