

7. ФУНКЦИИ

Оглавление

§7.1 Общая форма определения функции.....	1
§7.2 Вызов функции и аргументы	3
§7.3 Возвращаемые значения	5
§7.4 Создание функций с несколькими параметрами	8
§7.6 Перегрузка функции	10
§7.7 Передача аргумента по значению.....	11
§7.8 Передача аргумента по адресу.....	13
§7.9 Возврат значений по ссылке, по адресу и по значению.....	17
§7.10 Указатели на функции	21
§7.11 Рекурсия	28
§7.12 Делегаты (C#).....	33
§7.13 Добавление методов в делегат.....	35
§7.14 Анонимные методы.....	37
§7.15 Лямбды	38

§7.1 Общая форма определения функции

До сих пор рассматривались простые программы, все действия которых содержались в функции `main()`. В небольших программах и приложениях это работает хорошо. Но чем больше и сложнее становится программа, тем длиннее и запутаннее становится содержимое функции `main()`, если только вы не решите структурировать свою программу с помощью функций.

Функции позволяют разделить и организовать логику выполнения программы. Они разграничивают содержимое приложения на логические блоки, которые вызываются при необходимости.

Функция (function) — это подпрограмма, которая может получать параметры на вход и возвращать выходное значение. Чтобы выполнить свою задачу, функция должна быть вызвана. Она является важнейшим элементом процедурного программирования, так как позволяет группировать и обобщать программный код, который может позднее использоваться произвольное число раз.

С точки зрения внешней программы функция — это «черный ящик». Функция определяет собственную (локальную) область видимости, куда входят входные параметры, а, также, те переменные, которые объявляются непосредственно в теле самой функции.

Все функции C++ имеют общую форму, приведенную ниже:

```
ТИП_ВОЗВРАТА ИМЯ_ФУНКЦИИ(СПИСОК_ПАРАМЕТРОВ)
{
    //тело функции
}
```

Пример объявления функции на C++:

```
void main()
{
}
```

При определении функции C# (функции в данном языке программирования называются методами) используется следующая форма:

```
[модификаторы] ТИП_ВОЗВРАТА ИМЯ_МЕТОДА (СПИСОК_ПАРАМЕТРОВ)
{
    //тело метода
}
```

Пример объявления функции на C#:

```
static void main()
{
}
```

Здесь ТИП_ВОЗВРАТА определяет тип данного, возвращаемых функцией. Если функция не возвращает никакого значения, то используется тип `void`.

Имя функции определяется элементом ИМЯ_ФУНКЦИИ. В качестве имени допускается использовать любой идентификатор, если он еще не занят.

СПИСОК_ПАРАМЕТРОВ представляет собой последовательность пар типов и идентификаторов, разделяемых запятыми. **Параметры** — это переменные, которые получают значения аргументов, передаваемых функции при ее вызове. **Аргументы**— это

значения, которые передаются из caller-а в вызываемую функцию. Если функции не требуется параметров, то список параметров будет пуст.

Ключевое слово `static` является модификатором.

Фигурные скобки окружают **тело функции**. Тело функции состоит из инструкций, которые определяют, что именно эта функция делает. Когда по ходу выполнения инструкций встречается закрывающаяся скобка, то функция завершается, и управление передается вызывающему коду.

§7.2 Вызов функции и аргументы

Применение функции называется **вызовом функции** (function call). Когда функция объявлена вместе со списком параметров, при ее вызове нужно указать имя вызываемой функции и передать аргументы, являющимися значениями, затребованными функцией в списке ее параметров.

Рассмотрим программу, запрашивающую у пользователя радиус круга, а затем вычисляющую его площадь и периметр. Можно, конечно, поместить весь код в функцию `main()`, но можно разделить это приложение на логические блоки, а именно: вычисляющий площадь по данному радиусу и, соответственно, периметр.

Листинг 7.1.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>const double Pi = 3.14159;</code>
4	<code>double Area(double InputRadius)</code>
5	<code>{</code>
6	<code>return Pi * InputRadius * InputRadius;</code>
7	<code>}</code>
8	<code>double Circumference(double InputRadius)</code>
9	<code>{</code>
10	<code>return 2 * Pi * InputRadius;</code>
11	<code>}</code>
12	<code>int main()</code>
13	<code>{</code>
14	<code>cout << "Enter radius: ";</code>
15	<code>double Radius = 0;</code>
16	<code>cin >> Radius;</code>
17	<code>cout << "Area is: " << Area(Radius) << endl;</code>
18	<code>cout << "Circumference is: " << Circumference(Radius) << endl;</code>
19	<code>return 0;</code>

20	}
----	---

Результат выполнения программы:

Enter radius: 6.5

Area is: 132.732

Circumference is: 40.8407

Разделение вычисления площади и периметра на отдельные функции позволяет многократно использовать этот код, поскольку функции можно вызывать неоднократно.

Функции `Area()` и `Circumference()` вызываются в функции в строках 17 и 18. `Radius` – аргумент, передаваемый в функцию `Area()` и `Circumference()`. При вызове выполнение переходит к функции `Area()`, которая использует переданный радиус для вычисления площади круга. Когда функция завершает работу, она возвращает значение типа `double`, которое затем отображается на экране оператором `cout`.

При вызове функции, все её параметры создаются как локальные переменные, а значение каждого из аргументов копируется в соответствующий параметр (локальную переменную). Этот процесс называется **передачей по значению**. Например, при вызове функции `Area()` значение передаваемого аргумента `Radius` копировалось в параметр `InputRadius`.

Также `main()` не является единственной функцией, которая может вызывать другие функции. Любая функция может вызывать любую другую функцию.

В C++ одни функции не могут быть объявлены внутри других функций (т.е. быть вложенными). Это приведет к ошибке компиляции.

В C++ вы не можете вызвать функцию до объявления самой функции. Все потому, что компилятор не будет знать полное имя функции (имя функции, число аргументов, типы аргументов). Таким образом в примере ниже компилятор сообщит нам об ошибке:

Листинг 7.2.

1	<code>int main() {</code>
2	<code> Sum_numbers(1, 2);</code>
3	<code> return 0;</code>
4	<code>}</code>
5	<code>int Sum_numbers(int a, int b) {</code>

6	<code>cout << a + b;</code>
7	<code>}</code>

Так, при вызове функции `Sum_numbers()` внутри функции `main()` компилятор не знает ее полное имя.

Конечно компилятор C++ мог просмотреть весь код и определить полное имя функции, но этого он делать не умеет и нам приходится с этим считаться.

Поэтому мы обязаны проинформировать компилятор о полном имени функции. Для этого мы будем использовать прототип функции.

Прототип функции — это функция, в которой отсутствует блок кода (тело функции). В прототипе функции находятся:

- Полное имя функции.
- Тип возвращаемого значения функции.

Вот как правильно должна была выглядеть программа, написанная выше:

Листинг 7.3.

1	<code>int Sum_numbers(int a, int b); // прототип функции</code>
2	<code>int main() {</code>
3	<code>Sum_numbers(1, 2);</code>
4	<code>return 0;</code>
5	<code>}</code>
6	<code>int Sum_numbers(int a, int b) { // сама функция</code>
7	<code>cout << a + b;</code>
8	<code>}</code>

Так, с помощью прототипа мы объясняем компилятору, что полным именем функции является `Sum_numbers(int a, int b)`, и кроме этого мы говорим компилятору о типе возвращаемого значения, у нас им является тип `int`.

§7.3 Возвращаемые значения

Когда функция `main()` завершает своё выполнение, она возвращает целочисленное значение обратно в операционную систему, используя оператор `return`.

Функции, которые мы пишем, также могут возвращать значения. Для этого указывается тип возвращаемого значения. Он указывается при объявлении функции, перед её именем. Обратите внимание, тип возврата не указывает, какое именно значение будет возвращаться. Он указывает только тип этого значения.

Затем, внутри вызываемой функции, мы используем оператор `return`, чтобы указать возвращаемое значение. Когда процессор встречает в функции оператор `return`, он немедленно выполняет возврат значения обратно в `caller` и точка выполнения также переходит в `caller`. Любой код, который находится за `return`-ом в функции — игнорируется.

Функция может возвращать только одно значение через `return` обратно в `caller`. Это может быть либо число (например, 7), либо значение переменной, либо выражение (у которого есть результат), либо определённое значение из набора возможных значений.

Между возвращаемым типом функции и возвращаемым значением после оператора `return` должно быть соответствие. Если у функции объявлен целочисленный тип возвращаемого значения, то она не может вернуть в качестве значения, например, строковый тип.

Рассмотрим простую функцию, которая возвращает целочисленное значение:

Листинг 7.4.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>// int означает, что функция возвращает целочисленное значение</code> <code>обратно в caller</code>
4	<code>int return7()</code>
5	<code>{</code>
6	<code>// Эта функция возвращает целочисленное значение, поэтому мы</code> <code>должны использовать оператор return</code>
7	<code>return 7; // возвращаем число 7 обратно в caller</code>
8	<code>}</code>
9	<code>int main()</code>
10	<code>{</code>
11	<code>cout << return7() << endl; // выведется 7</code>
12	<code>cout << return7() + 3 << endl; // выведется 10</code>
13	<code>return7(); // возвращаемое значение 7 игнорируется, так как</code> <code>main() ничего с ним не делает</code>
14	<code>return 0;</code>
15	<code>}</code>

Результат выполнения программы:

7

10

Разберемся детальнее:

- Первый вызов функции `return7()` возвращает 7 обратно в caller, которое затем передается в `cout` для вывода
- Второй вызов функции `return7()` опять возвращает 7 обратно в caller. Выражение `7 + 3` имеет результат 10, который затем выводится на экран.
- Третий вызов функции `return7()` опять возвращает 7 обратно в caller. Однако `main()` ничего с ним не делает, поэтому ничего и не происходит (возвращаемое значение игнорируется).

Обратите внимание, что возвращаемые значения не выводятся на экран, если их не передать объекту `cout`. В последнем вызове функции `return7()` значение не отправляется в `cout`, поэтому ничего и не происходит.

Одну и ту же функцию можно вызывать несколько раз, даже в разных программах, что очень полезно:

Листинг 7.5.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>int getValueFromUser()</code>
4	<code>{</code>
5	<code> cout << "Enter an integer: ";</code>
6	<code> int x;</code>
7	<code> cin >> x;</code>
8	<code> return x;</code>
9	<code>}</code>
10	<code>int main()</code>
11	<code>{</code>
12	<code>int a = getValueFromUser(); // первый вызов функции</code>
13	<code>int b = getValueFromUser(); // второй вызов функции</code>
14	<code>cout << a << " + " << b << " = " << a + b << endl;</code>
15	<code>return 0;</code>
16	<code>}</code>

Функция `getValueFromUser()` получает значение от пользователя, а затем возвращает его обратно в caller.

Результат выполнения программы:

Enter an integer: 10

Enter an integer: 5

$$10 + 5 = 15$$

Здесь `main()` прерывается 2 раза. Обратите внимание, в обоих случаях, полученное пользовательское значение сохраняется в переменной `x`, а затем передаётся обратно в `main()` с помощью `return`, где присваивается переменной `a` или `b`.

§7.4 Создание функций с несколькими параметрами

Предположим, вы пишете программу, которая вычисляет площадь цилиндра по известному радиусу и высоте.

Вы использовали бы для этого следующую формулу:

$$\begin{aligned} \text{Площадь цилиндра} &= \text{Площадь верхнего круга} + \text{Площадь нижнего круга} + \text{Площадь} \\ \text{боковой поверхности} &= \text{Pi} * \text{Radius}^2 + \text{Pi} * \text{Radius}^2 + 2 * \text{Pi} * \text{Radius} * \text{Height} = 2 * \\ &\quad \text{Pi} * \text{Radius}^2 + 2 * \text{Pi} * \text{Radius} * \text{Height} \end{aligned}$$

Таким образом, при вычислении площади цилиндра необходимо работать с двумя переменными — радиусом и высотой. Поэтому при объявлении такой функции в списке ее параметров определяется по крайней мере два параметра. В списке параметров их отделяют запятой.

Листинг 7.6.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>const double Pi = 3.14159;</code>
4	<code>// Объявление содержит два параметра</code>
5	<code>double SurfaceArea(double Radius, double Height)</code>
6	<code>{</code>
7	<code>double Area = 2 * Pi * Radius * Radius + 2 * Pi * Radius \ *</code> <code>Height;</code>
8	<code>return Area;</code>
9	<code>}</code>
10	<code>int main ()</code>
11	<code>{</code>
12	<code>cout « "Enter the radius of the cylinder: ";</code>
13	<code>double InRadius = 0;</code>
14	<code>cin » InRadius;</code>
15	<code>cout « "Enter the height of the cylinder: ";</code>
16	<code>double InHeight = 0;</code>
17	<code>cin » InHeight;</code>
18	<code>cout « "Surface Area: " « SurfaceArea(InRadius, InHeight) «</code> <code>endl;</code>
19	<code>return 0;</code>

Результат выполнения программы:

Enter the radius of the cylinder: 3

Enter the height of the cylinder: 6.5

Surface Area: 179.071

Строка 5 содержит объявление функции `SurfaceArea()` с двумя параметрами: `Radius` и `Height`, оба имеют тип `double` и отделены запятой.

Параметры функций похожи на локальные переменные. Они допустимы только в пределах функции. Так, параметры `Radius` и `Height` функции `Surface Area()` в листинге 7.6 допустимы и пригодны для использования только в пределах самой функции `SurfaceArea()`, но не вне ее.

§7.5 Параметры функций со значениями по умолчанию

До сих пор мы использовали в примерах фиксированное значение числа Π как константу и не предоставляли пользователю возможности изменить его. Однако пользователя функции может интересовать более или менее точное значение. Когда вы создаете функцию, использующую число Π , как позволить ее пользователю применить собственное значение, а при его отсутствии задействовать стандартное?

Один из способов решения этой проблемы подразумевает создание в функции `Area()` дополнительного параметра для числа Π и присвоение ему **значения по умолчанию** (default value). Такая адаптация функции `Area()` из листинга 7.1 выглядела бы так:

```
double Area(double InputRadius, double Pi = 3.14);
```

Обратите внимание на второй параметр `Pi` и присвоенное ему по умолчанию значение 3,14. Этот второй параметр является теперь **необязательным параметром** (optional parameter) для вызывающей стороны. Так, вызывающая функция все еще может вызвать функцию `Area()`, используя такой синтаксис:

```
Area(Radius);
```

В данном случае второй параметр был проигнорирован, поэтому используется значение по умолчанию 3,14. Но если пользователь захочет задействовать другое значение числа Π , то сделать это можно, вызвав функцию `Area()` так:

```
Area(Radius, Pi); // Pi определяется пользователем
```

У функции может быть несколько параметров со значениями по умолчанию; но все они должны быть расположены в заключительной части списка параметров.

§7.6 Перегрузка функции

Функции с одинаковым именем и типом возвращаемого значения, но с разными параметрами или набором параметров называют **перегруженными функциями** (overloaded function). Перегруженные функции могут быть весьма полезными в приложениях, где функция с определенным именем осуществляет определенный тип вывода, но может быть вызвана с различными наборами параметров. Предположим, необходимо написать приложение, которое вычисляет площадь круга и площадь цилиндра. Функция, которая вычисляет площадь круга, нуждается в одном параметре — радиусе. Другая функция, которая вычисляет площадь цилиндра, нуждается кроме радиуса во втором параметре — высоте цилиндра. Обе функции должны вернуть данные одинакового типа — площадь. Языки программирования позволяют определить две перегруженные функции, обе по имени `Area`, и обе, возвращающие значение типа `double`, однако одна из них получает только радиус, а другая радиус и высоту, как представлено в листинге 7.7.

Листинг 7.7.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>const double Pi = 3.14159;</code>
4	<code>double Area(double Radius); // для круга</code>
5	<code>double Area(double Radius, double Height); // перегружено для цилиндра</code>
6	<code>int main ()</code>
7	<code>{</code>
8	<code>cout << "Enter z for Cylinder, c for Circle: ";</code>
9	<code>char Choice = ' z';</code>
10	<code>cin >> Choice;</code>
11	<code>cout << "Enter radius: ";</code>
12	<code>double Radius = 0;</code>
13	<code>cin >> Radius;</code>
14	<code>if (Choice == 'z')</code>
15	<code>{</code>
16	<code>cout << "Enter height: ";</code>
17	<code>double Height = 0;</code>
18	<code>cin >> Height;</code>

19	// Вызов перегруженной версии Area для цилиндра
20	cout << "Area of cylinder is: " << Area (Radius, Height) << endl;
21	}
21	else
23	cout << "Area of cylinder is: " << Area (Radius) << endl;
24	return 0;
25	}
26	// для круга
27	double Area(double Radius)
28	{
29	return Pi * Radius * Radius;
30	}
31	// перегружено для цилиндра
32	double Area(double Radius, double Height)
33	{
34	return 2 * Area(Radius) + 2 * Pi * Radius * Height;
35	}

Результат выполнения программы:

Enter z for Cylinder, c for Circle: z

Enter radius: 2

Enter height: 5

Area of cylinder is: 87.9646

В строках 4 и 5 объявлены прототипы перегруженных версий функции Area(), первая принимает один параметр (радиус круга), а вторая — два параметра (радиус и высоту цилиндра). У обеих функций одинаковые имена (Area) и типы возвращаемого значения (double), но разные наборы параметров. Следовательно, функция перегружена. Определения перегруженных функций находятся в строках 27-35, где реализованы две функции вычисления площади: круга по радиусу и цилиндра по радиусу и высоте соответственно. Интересно то, что, поскольку площадь цилиндра состоит из площади двух кругов (один сверху, второй снизу) и площади боковой стороны, перегруженная версия для цилиндра способна повторно использовать функцию Area() для круга, как показано в строке 34.

§7.7 Передача аргумента по значению

По умолчанию, аргументы в C++ передаются по значению. Когда аргумент передается по значению, то его значение копируется в параметр функции. Например:

1	#include <iostream>
2	void boo(int y)
3	{
4	using namespace std;
5	std::cout << "y = " << y << std::endl;
6	}
7	int main()
8	{
9	boo(7); // 1-й вызов
10	int x = 8;
11	boo(x); // 2-й вызов
12	boo(x + 2); // 3-й вызов
13	return 0;
14	}

Результат выполнения программы:

y = 7

y = 8

y = 10

В первом вызове функции `boo()` аргументом является литерал 7. При вызове `boo()` создается переменная `y`, в которую копируется значение 7. Затем, когда `boo()` завершает свое выполнение, переменная `y` уничтожается.

Во втором вызове функции `boo()` аргументом является уже переменная `x = 8`. Когда `boo()` вызывается во второй раз, переменная `y` создается снова и значение 8 копируется в `y`. Затем, когда `boo()` завершает свое выполнение, переменная `y` снова уничтожается.

В третьем вызове функции `boo()` аргументом является выражение `x + 2`, которое вычисляется в значение 10. Затем это значение передается в переменную `y`. При завершении выполнения функции `boo()` переменная `y` вновь уничтожается.

Поскольку в функцию передается копия аргумента, то исходное значение не может быть изменено функцией. Например, при изменении в вышеприведенном примере значения параметра `y` в функции `boo()` не изменится значение аргумента `x`, который мы передаем при вызове функции.

Преимущества передачи аргументов по значению:

- Аргументы, переданные по значению, могут быть переменными (например, x), литералами (например, 8), выражениями (например, $x + 2$), структурами, классами или перечислителями (т.е. почти всем, чем угодно).
- Аргументы никогда не изменяются функцией, в которую передаются, что предотвращает возникновение побочных эффектов.

Недостатком передачи аргументов по значению является то, что копирование структур и классов может привести к значительному снижению производительности (особенно, когда функция вызывается много раз).

Передачу по значению стоит использовать тогда, когда предполагается, что функция не должна изменять аргумент.

§7.8 Передача аргумента по адресу

При передаче аргументов по значению, единственный способ вернуть значение обратно в вызывающий объект — это использовать возвращаемое значение функции. Но иногда случаются ситуации, когда нужно, чтобы функция изменила значение переданного аргумента. Передача по ссылке (параграф 6.8) и по адресу решает все эти проблемы.

Передача аргументов по адресу — это передача адреса переменной-аргумента (а не исходной переменной). Поскольку аргумент является адресом, то параметром функции должен быть указатель. Затем функция сможет разыменовать этот указатель для доступа или изменения исходного значения. Вот пример функции, которая принимает параметр, передаваемый по адресу:

Листинг 7.9.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>void boo(int *ptr)</code>
4	<code>{</code>
5	<code> *ptr = 7;</code>
6	<code>}</code>
7	<code>int main()</code>
8	<code>{</code>
9	<code> int value = 4;</code>
10	<code> cout << "value = " << value << '\n';</code>
11	<code> boo(&value);</code>
12	<code> cout << "value = " << value << '\n';</code>

13	<code>return 0;</code>
14	<code>}</code>

Результат выполнения программы:

value = 4

value = 7

Как вы можете видеть, функция `boo()` изменила значение аргумента (переменную `value`) через параметр-указатель `ptr`.

Передачу по адресу обычно используют с указателями на обычные массивы. Например, следующая функция выведет все значения массива:

Листинг 7.10

1	<code>void printArray(int *array, int length)</code>
2	<code>{</code>
3	<code>for (int index=0; index < length; ++index)</code>
4	<code>cout << array[index] << ' ';</code>
5	<code>}</code>

Вот пример программы, которая вызывает эту функцию:

Листинг 7.11.

1	<code>int main()</code>
2	<code>{</code>
3	<code>int array[7] = { 9, 8, 6, 4, 3, 2, 1 }; // помните, что массивы распадаются в указатели при передаче</code>
4	<code>printArray(array, 7); // поэтому здесь array - это указатель на первый элемент массива (в использовании оператора & нет необходимости)</code>
5	<code>}</code>

Результат:

9 8 6 4 3 2 1

Фиксированные массивы распадаются в указатели при передаче в функцию, поэтому их длину нужно передавать в виде отдельного параметра. Перед разыменованием параметров, передаваемых по адресу, не лишним будет проверить — не являются ли они нулевыми указателями. Разыменование нулевого указателя приведет к сбою в программе. Проверка нулевого указателя выглядит следующим образом:

```
if (!array)
    return;
```

Поскольку `printArray()` всё равно не изменяет значения получаемых аргументов, то хорошей идеей будет сделать параметр `array` константным:

```
const int *array
```

Когда вы передаете указатель в функцию по адресу, то значение этого указателя (адрес, на который он указывает) **копируется из аргумента в параметр** функции. Другими словами, он передается по значению! Если изменить значение параметра функции, то изменится только копия, исходный указатель-аргумент не будет изменен. Например:

Листинг 7.12.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>void setToNull(int *tempPtr)</code>
4	<code>{</code>
5	<code> tempPtr = nullptr;</code>
6	<code>}</code>
7	<code>int main()</code>
8	<code>{</code>
9	<code> int six = 6;</code>
10	<code> int *ptr = &six;</code>
11	<code> cout << *ptr << "\n";</code>
12	<code> setToNull(ptr);</code>
13	<code> if (ptr)</code>
14	<code> cout << *ptr << "\n";</code>
15	<code> else</code>
16	<code> cout << " ptr is null";</code>
17	<code> return 0;</code>
18	<code>}</code>

В `tempPtr` копируется адрес указателя `ptr`. Несмотря на то, что мы изменили `tempPtr` на нулевой указатель (присвоили ему `nullptr`), это никак не повлияло на значение, на которое указывает `ptr`. Следовательно, результат выполнения программы:

6

6

Обратите внимание, хотя сам адрес передается по значению, вы всё равно можете разыменовывать его для изменения значения исходного аргумента.

При передаче аргумента по адресу в переменную-параметр функции копируется адрес из аргумента. В этот момент параметр функции и аргумент указывают на одно и то же значение.

Если параметру функции присвоить другой адрес, то это никак не повлияет на аргумент, поскольку параметр функции является копией, а изменение копии не приводит к изменению оригинала. После изменения адреса параметра функции, параметр функции и аргумент будут указывать на разные значения, поэтому разыменование параметра и дальнейшее его изменение никак не повлияют на значение, на которое указывает аргумент.

Если параметр функции затем разыменовывать для изменения исходного значения, то это приведет к изменению значения, на которое указывает аргумент, поскольку параметр функции и аргумент указывают на одно и то же значение!

В следующей программе это всё хорошо проиллюстрировано:

Листинг 7.13.

1	#include <iostream>
2	using namespace std;
3	void setToSeven(int *tempPtr)
4	{
5	*tempPtr = 7;
6	}
7	int main()
8	{
9	int six = 6;
10	int *ptr = &six;
11	cout << *ptr << "\n";
12	setToSeven(ptr);
13	if (ptr)
14	cout << *ptr << "\n";
15	else
16	cout << " ptr is null";
17	return 0;
18	}

Результат выполнения программы:

6

7

Если необходимо изменить адрес, на который указывает аргумент, внутри функции, то необходимо передать этот адрес по ссылке. Тогда прототип функции был бы объявлен следующим образом:

```
void setToNull(int *&tempPtr)
```

§7.9 Возврат значений по ссылке, по адресу и по значению

Рассмотрим возврат значений обратно из функции в вызывающий объект. **Возврат значений** (с помощью оператора `return`) работает почти так же, как и передача значений в функцию. Все те же плюсы и минусы. Основное отличие состоит в том, что поток данных двигается уже в противоположную сторону. Однако здесь есть еще один нюанс — локальные переменные, которые выходят из области видимости и уничтожаются, когда функция завершает свое выполнение.

Возврат по значению — это самый простой и безопасный тип возврата. При возврате по значению, копия возвращаемого значения передается обратно в `caller`. Как и в случае с передачей по значению, вы можете возвращать литералы (например, `7`), переменные (например, `x`) или выражения (например, `x + 2`), что делает этот способ очень гибким.

Еще одним преимуществом является то, что вы можете возвращать переменные (или выражения), в вычислении которых задействованы и локальные переменные, объявленные в теле самой функции. При этом, можно не беспокоиться о проблемах, которые могут возникнуть с областью видимости. Поскольку переменные вычисляются до того, как функция производит возврат значения, то здесь не должно быть никаких проблем с областью видимости этих переменных, когда заканчивается блок, в котором они объявлены. Например:

```
int doubleValue(int a)
{
    int value = a * 3;
    return value; // копия value возвращается здесь
} // value выходит из области видимости здесь
```

Возврат по значению идеально подходит для возврата переменных, которые были объявлены внутри функции, или для возврата аргументов функции, которые были

переданы по значению. Однако, подобно передаче по значению, возврат по значению медленный при работе со структурами и классами.

Когда использовать возврат по значению:

- при возврате переменных, которые были объявлены внутри функции;
- при возврате аргументов функции, которые были переданы в функцию по значению.

Когда не использовать возврат по значению:

- при возврате стандартных массивов или указателей (используйте возврат по адресу);
- при возврате больших структур или классов (используйте возврат по ссылке).

Возврат по адресу — это возврат адреса переменной обратно в caller. Подобно передаче по адресу, возврат по адресу может возвращать только адрес переменной. Литералы и выражения возвращать нельзя, так как они не имеют адресов. Поскольку при возврате по адресу просто копируется адрес из функции в caller, то этот процесс также очень быстрый.

Тем не менее, этот способ имеет один недостаток, который отсутствует при возврате по значению: если вы попытаетесь вернуть адрес локальной переменной, то получите неожиданные результаты. Например:

```
int* doubleValue(int a)
{
    int value = a * 3;
    return &value; // value возвращается по адресу здесь
} // value уничтожается здесь
```

Как вы можете видеть, `value` уничтожается сразу после того, как его адрес возвращается в caller. Конечным результатом будет то, что caller получит адрес освобожденной памяти (висячий указатель), что, несомненно, вызовет проблемы. Это одна из самых распространенных ошибок, которую делают новички. Большинство современных компиляторов выдадут предупреждение (а не ошибку), если программист попытается вернуть локальную переменную по адресу. Однако есть несколько способов обмануть компилятор, чтобы сделать что-то «плохое», не генерируя при этом

предупреждения, поэтому вся ответственность лежит на программисте, который должен гарантировать, что возвращаемый адрес будет корректен.

Возврат по адресу часто используется для возврата динамически выделенной памяти обратно в caller.

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(20);
    // Делаем что-нибудь с array
    delete[] array;
    return 0;
}
```

Здесь не возникнет никаких проблем, так как динамически выделенная память не выходит из области видимости в конце блока, в котором объявлена и всё еще будет существовать, когда адрес будет возвращаться в caller.

Когда использовать возврат по адресу:

- при возврате динамически выделенной памяти;
- при возврате аргументов функции, которые были переданы по адресу.

Когда не использовать возврат по адресу:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате большой структуры или класса, который был передан по ссылке (используйте возврат по ссылке).

Подобно передаче по ссылке, значения, **возвращаемые по ссылке**, должны быть переменными (вы не сможете вернуть ссылку на литерал или выражение). При возврате по ссылке в caller возвращается ссылка на переменную. Затем caller может её

использовать для продолжения изменения переменной, что может быть иногда полезно. Этот способ также очень быстрый и при возврате больших структур или классов.

Однако, как и в возврате по адресу, вы не должны возвращать локальные переменные по ссылке. Рассмотрим следующий фрагмент кода:

```
int& doubleValue(int a)
{
    int value = a * 3;
    return value; // value возвращается по ссылке здесь
} // value уничтожается здесь
```

В программе, приведенной выше, возвращается ссылка на переменную value, которая уничтожится, когда функция завершит свое выполнение. Это означает, что caller получит ссылку на мусор. Компилятор, вероятнее всего, выдаст предупреждение или ошибку, если вы попытаетесь это сделать.

Возврат по ссылке обычно используется для возврата аргументов, переданных в функцию по ссылке. В следующем примере мы возвращаем (по ссылке) элемент массива, который был передан в функцию по ссылке:

Листинг 7.14.

1	#include <iostream>
2	#include <array>
3	using namespace std;
4	int& getElement(std::array<int, 20> &array, int index)
5	{
6	return array[index];
7	}
8	int main()
9	{
10	array<int, 20> array;
11	getElement(array, 15) = 7;
12	cout << array[15] << '\n';
13	return 0;
14	}

Результат выполнения программы:

Когда мы вызываем `getElement(array, 15)`, то `getElement()` возвращает ссылку на элемент массива под индексом 15, а затем `main()` использует эту ссылку для присваивания этому элементу значения 7.

Когда использовать возврат по ссылке:

- при возврате ссылки-параметра;
- при возврате элемента массива, который был передан в функцию;
- при возврате большой структуры или класса, который не уничтожается в конце функции (например, тот, который был передан в функцию).

Когда не использовать возврат по ссылке:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате стандартного массива или значения указателя (используйте возврат по адресу).

§7.10 Указатели на функции

Указатель на функцию (function pointer) хранит адрес функции. По сути указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя. С помощью имени функции можно вызывать ее и получать результат ее работы.

Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

тип (*имя_указателя) (параметры)

Здесь тип представляет тип возвращаемого функцией значения.

`имя_указателя` представляет произвольно выбранный идентификатор в соответствии с правилами о наименовании переменных.

И параметры определяют тип и название параметров через запятую при их наличии.

Например, определим указатель на функцию:

```
void (*message) ()
```

В данном случае определен указатель, который имеет имя `message`. Он может указывать на функции без параметров, которые возвращают тип `void` (то есть ничего не возвращают).

Используем указатель на функцию:

Листинг 7.15.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>void hello();</code>
4	<code>void goodbye();</code>
5	<code>int main()</code>
6	<code>{</code>
7	<code>void (*message)();</code>
8	<code>message=hello;</code>
9	<code>message();</code>
10	<code>message = goodbye;</code>
11	<code>message();</code>
12	<code>return 0;</code>
13	<code>}</code>
14	<code>void hello()</code>
15	<code>{</code>
16	<code>cout << "Hello, World" << endl;</code>
17	<code>}</code>
18	<code>void goodbye()</code>
19	<code>{</code>
20	<code>cout << "Good Bye, World" << endl;</code>
21	<code>}</code>

Указателю на функцию можно присвоить функцию, которая соответствует указателю по возвращаемому типу и спецификации параметров:

```
message=hello;
```

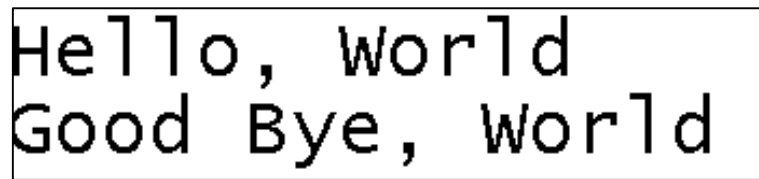
То есть в данном случае указатель `message` теперь хранит адрес функции `hello`. И посредством обращения к указателю мы можем вызвать эту функцию:

```
message();
```

В качестве альтернативы мы можем обращаться к указателю на функцию следующим образом:

```
(*message)();
```

Впоследствии мы можем присвоить указателю адрес другой функции, как в данном случае. В итоге результатом данной программы будет следующий вывод (см. Рисунок 7.1):



```
Hello, World
Good Bye, World
```

Рисунок 7.1

При определении указателя стоит обратить внимание на скобки вокруг имени. Так, использованное выше определение

```
void (*message) ();
```

НЕ будет аналогично следующему определению:

```
void *message ();
```

Во втором случае определен не указатель на функцию, а прототип функции `message`, которая возвращает указатель типа `void*`.

Рассмотрим еще один указатель на функцию:

Листинг 7.16.

1	<code>#include <iostream></code>
2	<code>using namespace std;</code>
3	<code>int add(int, int);</code>
4	<code>int subtract(int, int);</code>
5	<code>int main()</code>
6	<code>{</code>
7	<code>int a = 10;</code>
8	<code>int b = 5;</code>
9	<code>int result;</code>
10	<code>int (*operation)(int a, int b);</code>
11	<code>operation=add;</code>
12	<code>result = operation(a, b);</code>
13	<code>cout << "result=" << result << endl; // result=15</code>
14	<code>operation = subtract;</code>
15	<code>result = operation(a, b);</code>
16	<code>cout << "result=" << result << endl; // result=5</code>
17	<code>return 0;</code>
18	<code>}</code>
19	<code>int add(int x, int y)</code>
20	<code>{</code>
21	<code>return x+y;</code>

22	}
23	int subtract(int x, int y)
24	{
25	return x-y;
26	}

Здесь определен указатель `operation`, который может указывать на функцию с двумя параметрами типа `int`, возвращающую также значение типа `int`. Соответственно мы можем присвоить указателю адреса функций `add` и `subtract` и вызвать их, передав при вызове указателю некоторые значения для параметров.

Кроме одиночных указателей на функции мы можем определять их массивы. Для этого используется следующий формальный синтаксис:

тип (*имя_массива[размер]) (параметры)

Например:

double (*actions[]) (int, int)

Здесь `actions` представляет массив указателей на функции, каждая из которых обязательно должна принимать два параметра типа `int` и возвращать значение типа `double`.

Посмотрим применение массива указателей на функции на примере:

Листинг 7.17.

1	#include <iostream>
2	using namespace std;
3	void add(int, int);
4	void subtract(int, int);
5	void multiply(int, int);
6	int main()
7	{
8	int a = 10;
9	int b = 5;
10	void (*operations[3])(int, int) = {add, subtract, multiply};
11	// получаем длину массива
12	int length = sizeof(operations)/sizeof(operations[0]);
13	for(int i=0; i < length;i++)
14	{
15	operations[i](a, b); // вызов функции по указателю
16	}
17	return 0;
18	}

19	void add(int x, int y)
20	{
21	cout << "x + y = " << x + y << endl;
22	}
23	void subtract(int x, int y)
24	{
25	int result = x - y;
26	cout << "x - y = " << x - y << endl;
27	}
28	void multiply(int x, int y)
29	{
30	cout << "x * y = " << x * y << endl;
31	}

Здесь массив operations содержит три функции add, subtract и multiply, которые последовательно вызываются в цикле через перебор массива в функции main.

Консольный вывод программы представлен на Рисунке 7.2.

x	+	y	=	15
x	-	y	=	5
x	*	y	=	50

Рисунок 7.2

Указатель на функцию может передаваться в другую функцию в качестве параметра. Например:

Листинг 7.18.

1	#include <iostream>
2	using namespace std;
3	int add(int, int);
4	int subtract(int, int);
5	int operation(int(*)(int, int), int, int);
6	int main()
7	{
8	int a = 10;
9	int b = 5;
10	int result;
11	result = operation(add, a, b);
12	cout << "result: " << result << std::endl;
13	result = operation(subtract, a, b);
14	cout << "result: " << result << std::endl;
15	return 0;
16	}

17	int add(int x, int y)
18	{
19	return x + y;
20	}
21	int subtract(int x, int y)
22	{
23	return x - y;
24	}
25	int operation(int(*op)(int, int), int a, int b)
26	{
27	return op(a, b);
28	}

В данном случае первый параметр функции `operation` - `int (*op)(int, int)` - представляет указатель на функцию, которая возвращает значение типа `int` и принимает два параметра типа `int`. Результатом функции является вызов той функции, на которую указывает указатель.

Определению указателя соответствуют две функции: `add` и `subtract`, поэтому их адрес можно передать в вызов функции `operation`: `operation(add, a, b);`.

Функция может **возвращать указатель на другую функцию**. Это может быть актуально, если имеется ограниченное количество вариантов - выполняемых функций, и надо выбрать одну из них. Но при этом набор вариантов и выбор из них определяется в промежуточной функции.

Рассмотрим простейший пример:

Листинг 7.19.

1	#include <iostream>
2	using namespace std;
3	void goodmorning();
4	void goodevening();
5	void(*message(int))();
6	int main()
7	{
8	void(*action)(); // указатель на выбранную функцию
9	action = message(15);
10	action(); // выполняем полученную функцию
11	return 0;
12	}
13	void(*message(int hour))()
14	{

15	if (hour > 12)
16	return goodevening;
17	else
18	return goodmorning;
19	}
20	void goodmorning()
21	{
22	cout << "Good Morning!" << endl;
23	}
24	void goodevening()
25	{
26	cout << "Good Evening!" << endl;
27	}

Здесь определена функция `message`, которая в зависимости от переданного числа возвращает одну из двух функций `goodmorning` или `goodevening`. Рассмотрим объявление функции `message`:

```
void(*message(int hour))()
```

Вначале указан тип, который возвращается функцией, которая возвращается из `message`, то есть тип `void` (функции `goodmorning` и `goodevening` имеют тип `void`). Далее идет в скобках имя функции со списком параметров, то есть функция `message` принимает один параметр типа `int`: `(*message(int hour))`. После этого отдельно в скобках идет спецификация параметров функции, которая будет возвращаться из `message`. Поскольку функции `goodmorning` и `goodevening` не принимают никаких параметров, то указываются пустые скобки.

Имя функции фактически представляет указатель на нее, поэтому в функции `message` мы можем вернуть нужную функцию, указав после оператора `return` ее имя.

Для получения указателя на функцию определяем переменную `action`:

```
void(*action)();
```

Эта переменная представляет указатель на функцию, которая не принимает параметров и имеет в качестве возвращаемого типа тип `void`, то есть она соответствует функциям `goodmorning` и `goodevening`.

Затем вызываем функцию `message` и получаем указатель на функцию в переменную `action`:

```
action = message(15);
```

Далее, используя указатель `action`, вызываем полученную функцию:

```
action();
```

Поскольку в функцию `message` передается число 15, то она будет возвращать указатель на функцию `goodevening`, поэтому при ее вызове на консоль будет выведено сообщение "Good Evening!".

§7.11 Рекурсия

В некоторых случаях функция может фактически вызывать сама себя. Такая функция называется **рекурсивной** (recursive function).

Во-первых, нужно уяснить, что рекурсия — это, по сути, перебор. Задачи, которые можно решить итеративно, можно решить и рекурсивно, используя рекурсивную функцию. То есть практически любой алгоритм, который реализован в рекурсивной форме, можно переписать в итерационной форме и наоборот. Вопрос лишь в том, зачем это нужно и насколько эффективно.

Идём дальше. Так же, как и у цикла (перебора), рекурсия должна иметь условие остановки, называемое базовым случаем. Иначе и цикл, и рекурсия будут работать бесконечно. Условие остановки определяет шаг рекурсии. Рекурсивная функция вызывается, пока не произойдёт остановка рекурсии (не сработает базовое условие и не произойдёт возврат к последнему вызову функции). Именно поэтому решение задачи на рекурсию сводится к решению базового случая.

Если мы решаем сложную задачу (небазового случая), мы выполняем несколько рекурсивных вызовов либо шагов, т. к. наша цель — упростить задачу. И делать это до тех пор, пока не придём к базовому решению.

Ещё раз. Рекурсивная функция включает в себя:

- базовый случай, он же условие остановки;
- шаг рекурсии, он же условие продолжения.

Обратите внимание, что у рекурсивной функции должно быть четко определенное условие выхода, когда она завершает работу и больше себя не вызывает.

При отсутствии условия выхода или при ошибке в нем выполнение программы погрязнет в рекурсивном вызове функции, которая непрерывно будет вызывать сама

себя, пока в конечном счете не приведет к переполнению стека и аварийному завершению приложения.

Давайте рассмотрим принцип работы рекурсивных функций на примере нахождения факториала числа. Но прежде давайте вспомним, что собой представляет факториал и как он вычисляется.

$$\text{factorial}(N) = 1 * 2 * 3 * \dots * N - 1 * N$$

Проще говоря, факториал числа — это просто произведение всех чисел от 1 до N.

Чтобы перемножить всю заданную последовательность чисел, можно использовать цикл `for`.

Но если приглядеться, можно заметить, что для вычисления факториала можно применить рекурсивную структуру.

$$\text{factorial}(N) = N * \text{factorial}(N - 1)$$

Это все равно что передать задачу по вычислению другому вызову функции, работающему с уменьшенной версией исходной задачи. Давайте проверим, соответствует ли это решение тому, которое мы имеем с циклом `for`, а для этого развернем наше равенство.

$$\begin{aligned} f(N) &= N * f(N-1) \\ f(N-1) &= (N-1) * f(N-2) \\ f(N-2) &= (N-2) * f(N-3) \\ &\vdots \\ f(2) &= 2 * f(1) \\ \boxed{f(1) = 1} & \text{ Base case of recursion} \end{aligned}$$

Рисунок 7.3 – Шаги рекурсивной функции для вычисления факториала

$\rightarrow f(2) = 2 * f(1)$
 $= 2 * 1$
 $\rightarrow f(3) = 3 * f(2)$
 $= 3 * 2 * 1$
 $\rightarrow f(4) = 4 * f(3)$
 $= 4 * 3 * 2 * 1$
 $\rightarrow f(5) = 5 * f(4)$
 $= 5 * 4 * 3 * 2 * 1$
 Answer returned by smaller subproblem.

Рисунок 7.4 - Проверка правильности вычислений

Как видно по рисункам, наша рекуррентная формула вычисления факториала верна. Формализуем аналитическое представление рекуррентной формулы в виде программного кода.

Листинг 7.20.

1	#include <iostream>
2	using namespace std;
3	int factorial(int N)
4	{
5	if (N == 1)
6	return 1;
7	return N * factorial(N - 1);
8	}
9	int main()
10	{
11	int N = 0;
12	cin >> N;
13	cout << "factorial(N) = " << factorial(N) << endl;
14	return 0;
15	}

Рекурсивные функции могут пригодиться при вычислении чисел прогрессии Фибоначчи.

1 1 2 3 5 8 13

В последовательности Фибоначчи каждое число представляет собой сумму двух предыдущих чисел, а первые два числа — это либо 0 и 1, либо 1 и 1.

Общая формула для вычисления n -го числа Фибоначчи выглядит следующим образом:

$$F(n) = F(n - 1) + F(n - 2)$$

$$\text{где } F(1) = F(2) = 1$$

Определение последовательности Фибоначчи рекурсивно по своей природе, поскольку n -е число зависит от двух предыдущих чисел. Это означает, что задачу можно разделить на меньшие подзадачи, а это рекурсия.

Решение задачи представлено в примере:

Листинг 7.21.

1	#include <iostream>
2	using namespace std;
3	int GetFibNumber(int FibIndex)
4	{
5	if (FibIndex < 2)
6	return FibIndex;
7	else // рекурсия, если FibIndex >= 2
8	return GetFibNumber(FibIndex - 1) + GetFibNumber(FibIndex - 2);
9	}
10	int main()
11	{
12	cout << "Enter 0-based index of desired Fibonacci Number: ";
13	int Index = 0;
14	cin >> Index;
15	cout << "Fibonacci number is: " << GetFibNumber(Index) << endl;
16	return 0;
17	}

Результат выполнения программы:

Enter 0-based index of desired Fibonacci Number: 4

Fibonacci number is: 3

Функция `GetFibNumber()`, определенная в строках 3-9, рекурсивна, поскольку она вызывает сама себя в строке 8. Обратите внимание на условие выхода, расположенное в строках 5 и 6; когда индекс становится меньше двух, функция перестает быть рекурсивной. С учетом того, что функция вызывает сама себя, последовательно уменьшая значение индекса `FibIndex`, в определенный момент это

значение достигает уровня, когда срабатывает условие выхода и рекурсия останавливается.

Каждая рекурсивная задача имеет две необходимые вещи:

- Рекуррентное соотношение, определяющее состояния задачи и то, как основная задача может быть разделена на меньшие подзадачи. Сюда также входит базовый случай для прекращения рекурсии.
- Дерево рекурсии, демонстрирующее несколько первых вызовов рассматриваемой функции (если не все). Взгляните на дерево рекурсии для рекурсивного соотношения последовательности Фибоначчи:

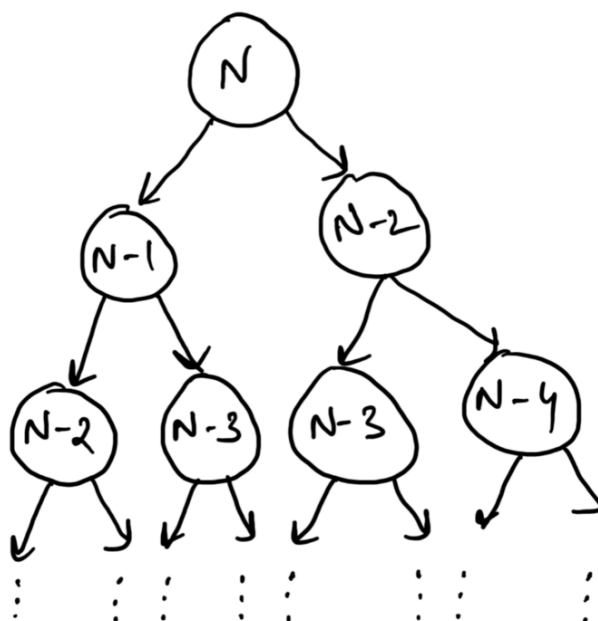


Рисунок 7.5 - Дерево рекурсии

Это дерево рекурсии показывает нам, что результаты, получаемые при обработке двух поддеревьев корня N , могут быть использованы для вычисления результата всего дерева. И так для каждого его узла.

Листьями этого дерева рекурсии будут `GetFibNumber(1)` и `GetFibNumber(0)`, которые представляют собой базовые случаи для этой рекурсии.

На рисунке 7.6 представлено рекурсивное дерево для программы из листинга 7.21.

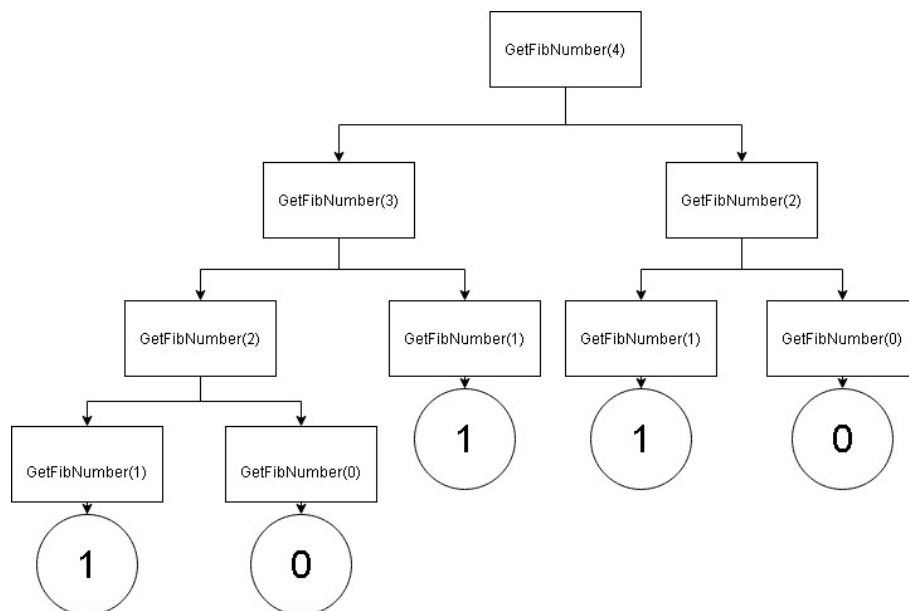


Рисунок 7.6

§7.12 Делегаты (C#)

Делегаты представляют такие объекты, которые указывают на методы. То есть **делегаты** - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры. Например:

```
delegate void Message();
```

Делегат **Message** в качестве возвращаемого типа имеет тип **void** (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим применение этого делегата:

Листинг 7.22.

1	<code>class Program</code>
2	<code>{</code>
3	<code> delegate void Message(); // 1. Объявляем делегат</code>
4	<code> static void Main(string[] args)</code>
5	<code> {</code>
6	<code> Message mes; // 2. Создаем переменную делегата</code>
7	<code> if (DateTime.Now.Hour < 12)</code>
8	<code> {</code>
9	<code> mes = GoodMorning; // 3. Присваиваем этой переменной</code>
	<code> адрес метода</code>

10	}
11	else
12	{
13	mes = GoodEvening;
14	}
15	mes(); // 4. Вызываем метод
16	Console.ReadKey();
17	}
18	private static void GoodMorning()
19	{
20	Console.WriteLine("Good Morning");
21	}
22	private static void GoodEvening()
23	{
24	Console.WriteLine("Good Evening");
25	}
26	}

В строке 3 происходит определение делегата Message. В данном случае делегат определяется внутри класса, но также можно определить делегат вне класса внутри пространства имен. Для использования делегата объявляется переменная этого делегата в строке 6. С помощью свойства `DateTime.Now.Hour` получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода (строка 10, 14). Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров (в данном случае отсутствие параметров), что и делегат. Затем, в строке 15, через делегат вызываем метод, на который ссылается данный делегат. Вызов делегата производится подобно вызову метода.

Посмотрим на примере другого делегата:

Листинг 7.23.

1	class Program
2	{
3	delegate int Operation(int x, int y);
4	static void Main(string[] args)
5	{
6	Operation del = Add; // делегат указывает на метод Add
7	int result = del(4,5); // фактически Add(4, 5)
8	Console.WriteLine(result);
9	del = Multiply; // теперь делегат указывает на метод
	Multiply
10	result = del(6, 3); // фактически Multiply(6, 3)

11	Console.WriteLine(result);
12	Console.Read();
13	}
14	private static int Add(int x, int y)
15	{
16	return x+y;
17	}
18	private static int Multiply (int x, int y)
19	{
20	return x * y;
21	}
22	}

В данном случае делегат Operation возвращает значение типа `int` и имеет два параметра типа `int`. Поэтому этому делегату соответствует любой метод, который возвращает значение типа `int` и принимает два параметра типа `int`. В данном случае это методы `Add` и `Multiply`. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа `int`, то при его вызове необходимо передать значения для этих параметров: `del(4,5)`.

Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

§7.13 Добавление методов в делегат

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - **список вызова** или `invocation list`. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот список не один, а несколько методов:

Листинг 7.24.

1	class Program
2	{
3	delegate void Message();
4	
5	static void Main(string[] args)
6	{

7	Message mes1 = Hello;
8	mes1 += HowAreYou; // теперь mes1 указывает на два метода
9	mes1(); // вызываются оба метода - Hello и HowAreYou
10	Console.Read();
11	}
12	private static void Hello()
13	{
14	Console.WriteLine("Hello");
15	}
16	private static void HowAreYou()
17	{
18	Console.WriteLine("How are you?");
19	}
20	}

В данном случае в список вызова делегата `mes1` добавляются два метода - `Hello` и `HowAreYou`. И при вызове `mes1` вызываются сразу оба этих метода.

Для добавления делегатов применяется операция `+=`. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делегата будет присвоен переменной `mes1`.

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен.

Подобным образом мы можем удалять методы из делегата с помощью операции `-=`.

При удалении методов из делегата фактически будет создаваться новый делегат, который в списке вызова методов будет содержать на один метод меньше.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция `-=` начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция `-=` не имеет никакого эффекта.

§7.14 Анонимные методы

С делегатами тесно связаны анонимные методы. Анонимные методы используются для создания экземпляров делегатов.

Определение анонимных методов начинается с ключевого слова `delegate`, после которого идет в скобках список параметров и тело метода в фигурных скобках:

```
delegate(параметры)
{
    // инструкции
}
```

Например:

Листинг 7.25.

1	<code>class Program</code>
2	<code>{</code>
3	<code> delegate void MessageHandler(string message);</code>
4	<code> static void Main(string[] args)</code>
5	<code> {</code>
6	<code> MessageHandler handler = delegate(string mes)</code>
7	<code> {</code>
8	<code> Console.WriteLine(mes);</code>
9	<code> };</code>
10	<code> handler("hello world!");</code>
11	<code> Console.Read();</code>
12	<code> }</code>
13	<code>}</code>

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная `handler` представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

Листинг 7.26.

1	<code>class Program</code>
2	<code>{</code>
3	<code> delegate void MessageHandler(string message);</code>
4	<code> static void Main(string[] args)</code>

5	{
6	ShowMessage("hello!", delegate(string mes)
7	{
8	Console.WriteLine(mes);
9	});
10	Console.Read();
11	}
12	static void ShowMessage(string mes, MessageHandler handler)
13	{
14	handler(mes);
15	}
16	}

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.

То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором `out`.

§7.15 Лямбды

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Лямбда-выражения имеют следующий синтаксис: слева от лямбда-оператора `=>` определяется список параметров, а справа блок выражений, использующий эти параметры:

(список_параметров) => выражение.

Например:

Листинг 7.27.

1	class Program
2	{

3	delegate int Operation(int x, int y);
4	static void Main(string[] args)
5	{
6	Operation operation = (x, y) => x + y;
7	Console.WriteLine(operation(10, 20)); // 30
8	Console.WriteLine(operation(40, 20)); // 60
9	Console.Read();
10	}
11	}

Здесь код

(x, y) => x + y;

представляет лямбда-выражение, где x и y - это параметры, а $x + y$ - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор `return`.

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. То есть в данном случае использованное лямбда-выражение соответствует делегату `Operation` как по типу возвращаемого значения, так и по типу и количеству параметров.

Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить.

Если параметры в лямбда-выражении не требуется, то вместо параметра в выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения.