ЗАНЯТИЕ 9
Подробней о классах

На этом занятии вы узнаете...

- что такое константная функция-член;
- как отделить интерфейс класса от его реализации;
- как манипулировать классами;
- как находить ошибки и как не допускать их.

Константные функции-члены

Если объявить функцию-член класса константной, используя ключевое слово `const`, то она не сможет изменить значение ни одного из членов класса. Для того чтобы объявить функцию-член класса как константную, необходимо поместить ключевое слово `const` после круглых скобок, но перед точкой с запятой:

```
void displayPage() const;
```

Функции-доступа, предназначенные для извлечения значений (их также называют функциями чтения (getter)), часто объявляют константными. Класс `Tricycle` имеет две функции доступа:

```
void setSpeed(int newSpeed);
int getSpeed();
```

Функция `setSpeed()` не может быть объявлена константной, поскольку она изменяет значение переменной-члена `speed`. С другой стороны, функция `getSpeed()` может быть объявлена константной, поскольку она ничего не изменяет в классе. Она просто возвращает текущее значение переменной-члена `speed`. Следовательно, объявление этой функций можно записать так:

```
int getSpeed() const;
```

Если объявить функцию константной, а затем в ее реализации каким-либо образом изменить объект, поменяв значение любого из его членов, то компилятор сообщит об ошибке.

Рекомендуется как можно чаще использовать ключевое слово `const` в объявлениях функций. Это позволит компилятору обнаруживать непреднамеренное

изменение переменных-членов до того, как эти ошибки проявятся на этапе выполнения программы.

Интерфейс и реализация

Часть программы, которая создает и использует объекты класса, называется *клиентом*. Объявление класса можно интерпретировать как контракт с клиентами, который сообщает клиентам, какие данные класса являются доступными и что класс может делать.

Например, объявление класса `Tricycle` представляет собой контракт, указывающий, что каждый объект класса `Tricycle` может определять свою скорость, которую можно инициализировать в конструкторе, а затем устанавливать или определять, а также что каждый объект класса `Tricycle` умеет ускоряться и тормозить с помощью функций `pedal()` и `brake()`.

Объявив функцию `getSpeed()` константной, контракт оговаривает, что она не будет изменять объект класса `Tricycle`, из которого будет вызвана.

Расположение объявлений классов и определений функций

В исходном коде программ на языке C++ определения классов часто отделяются от их реализаций. Каждая объявленная в классе функция должна иметь определение. Как и у всех остальных функций, определение функции-члена класса имеет заголовок и тело.

Определение должно находиться в файле, доступном компилятору. Для большинства компиляторов C++ такой файл должен иметь расширение `.cpp`.

Несмотря на то что объявления классов можно поместить в один файл с программой, рекомендуется записывать определения функций-членов в заголовочный файл с именем класса и расширением `.hpp`, `.h` или `.hp`.

Например, если поместить объявление класса `Tricycle` в файл `Tricycle.hpp`, то определения его функций-членов следует включить в файл `Tricycle.cpp`. Заголовочный файл включается в файл `.cpp` с помощью директивы препроцессора:

```
#include "Tricycle.hpp"
```

Отделение объявления класса от определений его функций-членов объясняется тем, что клиентов класса не должны интересовать детали его реализации. Им достаточно знать, что оно находится в заголовочном файле.

Подставляемая реализация

Как и обычные функции, функции-члены класса можно сделать подставляемыми. Для этого перед типом возвращаемого значения необходимо поместить ключевое слово `inline`:

```
inline int Tricycle::getSpeed()
{
   return speed;
}
```

Можно также поместить реализацию функции в объявление класса, что автоматически сделает функцию подставляемой, например:

```
class Tricycle
{
public:
   int getSpeed() const
   {
      return speed;
   }
   void setSpeed(int newSpeed);
};
```

В листингах 9.1 и 9.2 вновь создается класс `Tricycle`, но теперь его объявление перенесено в файл `Tricycle.hpp`, а реализации функций — в файл `Tricycle.cpp`. Кроме того, в листинге 9.1 функции `getSpeed()`, `pedal()` `brake()` сделаны подставляемыми.

ЛИСТИНГ 9.1. Полный текст файла **`Tricycle.hpp`**

```
1: #include <iostream>
2:
3: class Tricycle
4: {
5:    public:
6:       Tricycle(int initialSpeed);
7:       ~Tricycle();
8:       int getSpeed() const { return speed; }
9:       void setSpeed(int speed);
10:      void pedal()
11:      {
12:           setSpeed(speed + 1);
13:           std::cout << "Pedaling " << getSpeed() << " mph\n\n";
14:      }
15:      void brake()
16:     {
17:        setSpeed(speed - 1);
18:        std::cout << "Pedaling " << getSpeed() << " mph\n";
19:     }
20:     private:
21:        int speed;
22: };
```

ЛИСТИНГ 9.2. Полный текст файла **Tricycle.cpp**

```
1: #include "Tricycle.hpp"
2:
3: // Конструктор объекта
4: Tricycle::Tricycle(int initialSpeed)
5: {
6:    setSpeed(initialSpeed);
7: }
8:
9: // Деструктор объекта
10: Tricycle::~Tricycle()
11: {
12:    // Делаем что-то
13: }
14:
15: // Устанавливаем скорость
16: void Tricycle::setSpeed(int newSpeed)
17: {
18:    if (newSpeed >= 0)
19:    speed = newSpeed;
20: }
21:
22: // Создаем объект и едем
23: int main()
24: {
25:     Tricycle wichita(5);
26:     wichita.pedal();
27:     wichita.pedal();
28:     wichita.brake();
29:     wichita.brake();
30:     wichita.brake();
31:     return 0;
32: }
```

```
Pedaling 6 mph

Pedaling 7 mph

Pedaling 6 mph
Pedaling 5 mph
Pedaling 4 mph
```

Функция `getSpeed()` объявлена в строке 8 *листинга* 9.1 и реализована как подставляемая. Остальные подставляемые функции объявлены в строках 9–19.

Строка 1 в *листинге* 9.2 содержит директиву препроцессора включить заголовочный файл `Tricycle.hpp` в исходный код.

Классы, содержащие другие классы как данные-члены

При создании сложных классов нередко приходится объявлять относительно простые классы, которые впоследствии включаются в состав более сложных классов.

Например, можно объявить класс `Wheel` (Колесо), `Motor` (Двигатель), `Transmission` (Коробка передач) и т.д., а затем объединить их в класс `Car` (Автомобиль). Таким образом, между классами создается отношение "имеет" (has-a): автомобиль имеет двигатель, колеса и коробку передач.

Рассмотрим второй пример. Прямоугольник состоит из четырех отрезков, и каждый из них определяется двумя точками. Каждая точка определяется координатами `x` и `y`. В листинге 9.3 приведено объявление класса `Rectangle`, содержащегося в файле `Rectangle.hpp`.

Поскольку прямоугольник определяется четырьмя отрезками, соединяющими четыре точки, а каждая точка определяется координатами на плоскости, сначала объявляется класс `Point` для хранения координат `x` и `y` каждой точки. Полное определение обоих классов приведено в листинге 9.4.

ЛИСТИНГ 9.3. Полный текст файла **`Rectangle.hpp`**

```
1: #include <iostream>
2:
3: class Point
4: {
5:     // конструктора нет, используется конструктор по умолчанию
6: public:
7:     void setX(int newX) { x = newX; }
8:     void setY(int newY) { y = newY; }
9:     int getX() const { return x; }
10:    int getY() const { return y; }
11: private:
12:    int x;
13:    int y;
14: };
15:
16: class Rectangle
17: {
18: public:
19:    Rectangle(int newTop, int newLeft, int newBottom, int newRight);
20:    ~Rectangle() {}
21:
22:    int getTop() const { return top; }
23:    int getLeft() const { return left; }
24:    int getBottom() const { return bottom; }
25:    int getRight() const { return right; }
26:
27:    Point getUpperLeft() const { return upperLeft; }
```

```
28:    Point getLowerLeft() const { return lowerLeft; }
29:    Point getUpperRight() const { return upperRight; }
30:    Point getLowerRight() const { return lowerRight; }
31:
32:    void setUpperLeft(Point location);
33:    void setLowerLeft(Point location);
34:    void setUpperRight(Point location);
35:    void setLowerRight(Point location);
36:
37:    void setTop(int newTop);
38:    void setLeft (int newLeft);
39:    void setBottom (int newBottom);
40:    void setRight (int newRight);
41:
42:    int getArea() const;
43:
44: private:
45:    Point upperLeft;
46:    Point upperRight;
47:    Point lowerLeft;
48:    Point lowerRight;
49:    int top;
50:    int left;
51:    int bottom;
52:    int right;
53: };
```

ЛИСТИНГ 9.4. Полный текст программы `Rectangle.cpp`

```
1: #include "Rectangle.hpp"
2:
3: Rectangle::Rectangle(int newTop, int newLeft, int newBottom, int newRight)
4: {
5:    top = newTop;
6:    left = newLeft;
7:    bottom = newBottom;
8:    right = newRight;
9:
10:   upperLeft.setX(left);
11:   upperLeft.setY(top);
12:
13:   upperRight.setX(right);
14:   upperRight.setY(top);
15:
16:   lowerLeft.setX(left);
17:   lowerLeft.setY(bottom);
18:
19:   lowerRight.setX(right);
20:   lowerRight.setY(bottom);
21: }
```

```
22:
23: void Rectangle::setUpperLeft(Point location)
24: {
25:    upperLeft = location;
26:    upperRight.setY(location.getY());
27:    lowerLeft.setX(location.getX());
28:    top = location.getY();
29:    left = location.getX();
30: }
31:
32: void Rectangle::setLowerLeft(Point location)
33: {
34:    lowerLeft = location;
35:    lowerRight.setY(location.getY());
36:    upperLeft.setX(location.getX());
37:    bottom = location.getY();
38:    left = location.getX();
39: }
40:
41: void Rectangle::setLowerRight(Point location)
42: {
43:    lowerRight = location;
44:    lowerLeft.setY(location.getY());
45:    upperRight.setX(location.getX());
46:    bottom = location.getY();
47:    right = location.getX();
48: }
49:
50: void Rectangle::setUpperRight(Point location)
51: {
52:    upperRight = location;
53:    upperLeft.setY(location.getY());
54:    lowerRight.setX(location.getX());
55:    top = location.getY();
56:    right = location.getX();
57: }
58:
59: void Rectangle::setTop(int newTop)
60: {
61:    top = newTop;
62:    upperLeft.setY(top);
63:    upperRight.setY(top);
64: }
65:
66: void Rectangle::setLeft(int newLeft)
67: {
68:    left = newLeft;
69:    upperLeft.setX(left);
70:    lowerLeft.setX(left);
71: }
72:
```

```
73: void Rectangle::setBottom(int newBottom)
74: {
75:    bottom = newBottom;
76:    lowerLeft.setY(bottom);
77:    lowerRight.setY(bottom);
78: }
79:
80: void Rectangle::setRight(int newRight)
81: {
82:    right = newRight;
83:    upperRight.setX(right);
84:    lowerRight.setX(right);
85: }
86:
87: int Rectangle::getArea() const
88: {
89:    int width = right - left;
90:    int height = top - bottom;
91:    return (width * height);
92: }
93:
94: // Вычисляем площадь прямоугольника,
95: // перемножая его ширину и высоту
96: int main()
97: {
98:     // Инициализируем локальную переменную variable
99:     Rectangle myRectangle(100, 20, 50, 80);
100:
101:    int area = myRectangle.getArea();
102:
103:    std::cout << "Area: " << area << std::endl;
104:    std::cout << "Upper Left X Coordinate: ";
105:    std::cout << myRectangle.getUpperLeft().getX() << std::endl;
106:    return 0;
107: }
```

Программа, представленная в листинге 9.4, выводит на экран следующий результат:

```
Area: 3000
Upper Left X Coordinate: 20
```

В строках 3–14 листинга 9.3 объявлен класс `Point` (`Точка`), который используется для хранения координат `x` и `y` на плоскости.

В классе `Point` в строках 12 и 13 объявлены две переменные-члена для хранения координат точки. При увеличении координаты `x` точка на декартовой плоскости перемещается вправо. При увеличении координаты `y` точка перемещается вверх.

Класс Point использует для чтения и записи координат x и y подставляемые функции доступа, объявленные в строках 7–10. Объекты класса Point используют конструктор и деструктор по умолчанию, предоставляемые компилятором. Следовательно, координаты точек необходимо задавать явно.

В строке 16 начинается объявление класса Rectangle, который содержит четыре точки, представляющие углы прямоугольника.

Конструктор класса Rectangle (строка 19) получает четыре целочисленных параметра: newTop (верхний), newLeft (левый), newBottom (нижний) и newRight (правый). Их значения присваиваются четырем переменным-членам, после чего устанавливаются значения четырех объектов класса Point.

Помимо обычных функций доступа к данным-членам класса, в классе Rectangle предусмотрена функция getArea(), объявленная в строке 42. Вместо хранения значения площади в виде переменной эта функция вычисляет ее в строках 87–92 листинга 9.4. Для этого она сначала вычисляет ширину и высоту прямоугольника, а затем перемножает эти значения.

Для того чтобы получить координаты верхнего левого угла прямоугольника, необходимо вернуть значение x точки UpperLeft. Поскольку функция getUpperLeft() является членом класса Rectangle, она может получить доступ ко всем закрытым данным этого класса непосредственно, включая и доступ к переменной upperLeft. Поскольку переменная upperLeft является объектом класса Point, а переменная x этого объекта закрыта, функция getUpperLeft() не может обратиться к ней непосредственно. Для того чтобы получить значение переменной x, она вынуждена использовать открытую функцию доступа getX().

Тело программы начинается в строке 96 листинга 9.4. Вплоть до строки 99 память не выделяется, и ничего не происходит. Все предыдущие строки просто сообщали компилятору, как создаются объекты Point и Rectangle, если они потребуются.

В строке 99 при передаче реальных значений параметров top, left, bottom и Right создается прямоугольник.

В строке 101 создается локальная переменная area типа int, предназначенная для хранения площади созданного прямоугольника. Переменная area инициализируется значением, возвращаемым функцией-членом getArea() класса Rectangle.

Клиент класса Rectangle может создавать его объекты и вычислять их площадь, не интересуясь реализацией функции getArea().

Просмотрев заголовочный файл в листинге 9.3, содержащий объявление класса Rectangle, программист может узнать о том, что функция getArea() возвращает значение типа int. Клиентов класса Rectangle не интересует, как именно устроена функция getArea(), и разработчик класса может изменить ее реализацию, не оказывая никакого влияния на программы, использующие класс Rectangle.

Резюме

Программисты на языке C++ не обязаны использовать классы и объекты. Они могут писать программы, состоящие только из простых переменных и функций, не прибегая к сложным концепциям объектно-ориентированного программирования.

Однако так поступает очень мало программистов. Почему? Потому что существует намного более эффективный способ организации программ, основанный на совокупности взаимодействующих классов.

Разработка классов облегчает повторное использование кода. Если объект оказался полезным в одной программ, он может быть полезным и в другой. Для этого можно повторно использовать его класс. Например, в веб-браузер можно добавить класс текстового процессора `Spellchecker` и любой другой полезный инструмент для работы с текстом. Объекты класса `Spellchecker` могут работать одинаково хорошо в любой программе.

Кроме того, классы облегчают понимание программ. Задачи, решаемые программой, упаковываются вместе с данными, которые необходимо обработать. Рассматривая программу как совокупность объектов, каждый из которых выполняет определенное задание, можно повысить ее эффективность. Если же возникнут проблемы, классы упростят поиск ошибок.

Вопросы и ответы

В. Если объявление константной функции приводит к ошибке компиляции при изменении ею объекта, то почему бы не отказаться от использования ключевого слова `const`, избежав сообщений об ошибках?

О. Если функция-член теоретически не должна изменять класс, то использование ключевого слова `const` — хороший способ привлечь компилятор к поиску случайных ошибок. Например, посмотрим, что случится, если функция `getAge()` будет содержать следующую строку:

```
if (speed = 100) std::cout << "Maximum speed reached\n";
```

Объявление константной функции `getAge()` позволило бы компилятору обнаружить ошибку. Инструкция, которая должна была производить сравнение значения переменной `speed` с числом `100`, вместо этого случайно присваивает ей число `100`, потому что вместо оператора проверки равенства (==) в инструкции использован оператор присваивания (=). Поскольку такое присвоение изменяет класс, благодаря ключевому слову `const` компилятор сможет обнаружить ошибку.

О. Такие ошибки очень трудно обнаружить, просто просматривая код, потому что читающие обычно видят то, что хотят увидеть. Гораздо хуже, если окажется, что программа работает вроде бы правильно, несмотря на то, что переменной присвоено странное значение.

Коллоквиум

Изучив классы подробнее, вы можете ответить на несколько вопросов и выполнить ряд упражнений, чтобы закрепить полученные знания.

Контрольные вопросы

1. Какое расширение обычно не используется в качестве расширения заголовочных файлов?
   - **А.** `.hpp`
   - **Б.** `.cpp`
   - **В.** `.h`
2. Что случится, если класс `Point` не будет определен ни в файле `Rectangle.cpp`, ни в файле `Rectangle.hpp`?
   - **А.** Возникнет ошибка компиляции.
   - **Б.** Программа будет работать неправильно.
   - **В.** Что угодно.
3. Где компилятор может искать определения подставляемых функций?
   - **А.** В объявлении класса.
   - **Б.** В реализации класса.
   - **В.** Верны оба ответа.

Ответы

1. **б.** Заголовочные файлы в языке C++ обычно имеют расширение `.hpp` (иногда `.h`). Расширение `.cpp` обычно имеют файлы с исходным кодом программ на языке C++.
2. **а.** Если класс `Point` нигде не был определен, то компилятор выдаст сообщение об ошибке, свидетельствующее о наличии неопределенной ссылки (undefined reference). Классы довольно часто используют другие классы.
3. **в.** Функция-член класса, определение которой содержится в объявлении класса, считается подставляемой. В противном случае, для того чтобы ускорить выполнение функции-члена класса, ее объявление должно содержать ключевое слово `inline`.

Упражнения

1. Перенесите класс `Point` из файла `Rectangle.hpp` в отдельный заголовочный файл и включите его в файл `Rectangle.hpp` с помощью директивы `#include`. Изменило ли это процесс компиляции файла `Rectangle.cpp`? Изменился ли результат выполнения программы?
2. Напишите класс `Line`, состоящий из двух объектов класса `Point`.

Для того чтобы увидеть ответы на эти упражнения, посетите веб-сайт книги по адресу `http://cplusplus.cadenhead.org`.