Глава 12

Классы в C++

В этой главе...

- Введение в классы
- Формат класса
- Обращение к членам класса
- Активация наших объектов
- Добавление функции-члена
- Вызов функций-членов
- Разрешение области видимости
- Определение функции-члена
- Определение функций-членов вне класса
- Перегрузка функций-членов

Очень часто программы имеют дело с совокупностями данных: имя, должность, табельный номер и т.д. Каждая отдельная составляющая не описывает человека, смысл имеет только вся вместе взятая информация. Простая структура, такая как массив, прекрасно подходит для хранения отдельных значений, однако совершенно непригодна для хранения совокупности данных разных типов. Таким образом, массив недостаточен для хранения комплексной информации.

По причинам, которые вскоре станут понятными, я буду называть такие совокупности информации *объектами*. Микроволновая печь — объект (см. главу 11, "Знакомство с объектно-ориентированным программированием", если это кажется вам непонятным). Вы также объект (и я тоже, хотя уже и не так уверен в этом). Ваше имя, должность и номер кредитной карточки, содержащиеся в базе данных, тоже являются объектом.

Введение в классы

Было бы неплохо иметь возможность создания в C++ объектов, которые бы обладали свойствами моделируемых объектов реального мира. Для хранения разнотипной информации о физическом объекте нужна специальная структура. В C++ такая структура, которая может объединить разнотипные данные в одном объекте, называется *классом*.

Формат класса

Класс состоит из ключевого слова `class`, за которым следуют его имя и фигурные скобки. Класс, описывающий депозитный счет, и содержащий номер счета и остаток на счете, может иметь следующий вид:

```
class SavingsAccount
{
    public:
        unsigned accountNumber;
        double balance;
};
```

После открывающей скобки находится ключевое слово `public`. (Не спрашивайте меня сейчас, что оно значит, — я объясню его значение немного позже. В следующих главах поясняются разные ключевые слова, такие как `public` или `private`. А до тех пор пока я не сделаю `private` публичным, значение `public` останется приватным :-).)

Можно использовать альтернативное ключевое слово `struct`, которое полностью идентично `class` с предполагаемым использованием объявлений `public`. В случае применения ключевого слова `struct` предполагается наличие спецификатора `public`, который при этом можно опустить.

После ключевого слова `public` идет описание объекта. Класс `SavingsAccount` содержит два элемента — беззнаковое целое `accountNumber` и число с плавающей точкой `balance`. Можно сказать, что `accountNumber` и `balance` являются членами, или свойствами класса `SavingsAccount`.

Для создания объекта депозитного счета следует ввести нечто наподобие

```
SavingsAccount mySavingsAccount;
```

Мы говорим, что `mySavingsAccount` является экземпляром класса `SavngsAccount`.

Обычное соглашение об именовании в данном случае таково — имена классов обычно начинаются с прописной буквы. Если имя состоит из нескольких слов, как в случае `SavingsAccount`, каждое слово начинается с прописной буквы, а сами слова располагаются вместе, без символа подчеркивания. Имена объектов следуют тому же правилу, но начинаются со строчной буквы — как в случае `mySavingsAccount`. Как обычно, эти нормы (мне не хочется говорить “правила”) призваны помочь читающему программу человеку; ведь как именно вы назовете классы и объекты — C++ глубоко безразлично.

Обращение к членам класса

Синтаксис обращения к членам класса следующий:

```
// Создание объекта
SavingsAccount mySave;
mySave.accountNumber = 1234;
mySave.balance = 0;

// Ввод данных с клавиатуры
cout << "Введите номер счета и остаток на счету" << endl;
SavingsAccount urSave;
cin >> urSave.accountNumber;
cin >> urSave.balance;
```

В этом фрагменте кода объявляются два объекта класса `SavingsAccount`, `mySave` и `urSave`. Объект `mySave` инициализируется присваиванием значения 1234 номеру счета и 0 — остатку на счете (обычное явления для моего счета). Затем создается второй объект того же класса — `urSave`. Фрагмент считывает номер счета и остаток на счете с клавиатуры.

Важно отметить, что mySave и urSave — отдельные, независимые объекты. Работа с членами одного объекта никак не влияет на члены второго.

Кроме того, имя члена не имеет смысла без связанного с ним объекта. Так, нельзя записать следующее:

```
balance = 0;                 // Неверно: не указан объект
SavingsAccount.balance = 0; // Это класс, а не объект!
```

Каждый счет имеет собственный уникальный номер и поддерживает собственный баланс. (Могут иметься свойства, разделяемые всеми счетами, — но номер счета и баланс к ним не относятся.)

Активация наших объектов

Классы используются для моделирования объектов реального мира. Класс Savings пытается представить депозитный счет. Это позволяет вам мыслить в терминах объектов, а не строк кода. Чем лучше объекты C++ моделируют реальный мир, тем проще работать с ними в программах. Это звучит достаточно логично. Но класс Savings недостаточно хорошо моделирует депозитный счет.

Моделирование реальных объектов

Объекты реального мира обладают свойствами-данными, такими как номер счета или баланс, так же как и класс Savings. Это делает его хорошей стартовой точкой для описания реального объекта. Но реальные объекты еще и выполняют определенные действия. Печи готовят еду, счета накапливают прибыль, машины ездят…

Функциональные программы "выполняют действия" посредством функций. Программа на C++ может вызвать strcmp() для сравнения двух строк или max() для получения максимального из двух значений. В главе 23, "Использование потоков ввода-вывода", вы узнаете, что даже потоковый ввод-вывод (cin >> и cout <<) представляют собой специальный вид вызовов функций.

Классу Savings для представления реальной концепции требуются активные свойства:

```
class Savings
{
    public:
        double deposit(double amount)
        {
            balance += amount;
            return balance;
        }
        unsigned accountNumber;
        double balance;
};
```

В дополнение к номеру счета и балансу данная версия класса Savings включает функцию deposit(). Это дает классу Savings возможность управления собственными возможностями. Класс MicrowaveOven имеет функцию cook() (готовка), класс Savings — функцию accumulateInterest() (накапливание процентов), а класс Car — функцию move() (движение).

Функции, определенные в классе, называются *функциями-членами*.

Зачем нужны функции-члены

Почему мы должны возиться с функциями-членами? Что плохого в старом добром функциональном подходе?

```
class Savings
{
    public:
        unsigned accountNumber;
        double balance;
};
double deposit(Savings& s, double amount)
{
    s.balance += amount;
    return s.balance;
}
```

В этом фрагменте `deposit()` является функцией "вклада на счет". Эта функция поддержки реализована в виде внешней функции, которая выполняет необходимые действия с экземпляром класса `Savings`. Конечно, такой подход имеет право на существование, но он нарушает наши правила объектно-ориентированного программирования.

Микроволновая печь имеет свои внутренние компоненты, которые "знают", как разморозить и приготовить продукты или сделать картошку хрустящей. Данные-члены класса схожи с элементами микроволновой печи, а функции-члены — с программами приготовления.

Когда я делаю закуску, я не должен начинать приготовление с подключения внутренних элементов микроволновой печи. И я хочу, чтобы мои классы работали так же, т.е. чтобы они безо всякого внешнего вмешательства знали, как управлять своими "внутренними органами". Конечно, такие функции-члены класса `Savings`, как `deposit()`, можно реализовать и в виде внешних функций. Можно даже расположить все функции, необходимые для работы со счетами, в одном месте файла. Микроволновую печь можно заставить работать, соединив необходимые провода внутри нее, но я не хочу, чтобы мои классы (или моя микроволновая печь) работали таким образом. Я хочу иметь класс `Savings`, который буду использовать в своей банковской программе, не задумываясь над тем, какова его рабочая "кухня".

Добавление функции-члена

Чтобы продемонстрировать работу с функциями-членами, начнем с определения класса `Student`. Одно из возможных представлений такого класса имеет следующий вид (взятый из программы `CallMemberFunction`):

```
class Student
{
public:
    // Добавить пройденный курс к записи
    float addCourse(int hours, float grade)
    {
        // Вычислить среднюю оценку с учетом
        // времени различных курсов
        float weightedGPA;
        weightedGPA = semesterHours * gpa;

        // Добавить новый курс
        semesterHours += hours;
        weightedGPA   += grade * hours;
        gpa = weightedGPA / semesterHours;
```

```
        // Вернуть новую оценку
        return gpa;
    }
    int   semesterHours;
    float gpa;
};
```

Функция `addCourse(int, float)` является функцией-членом класса `Student`. По сути, это такое же свойство класса `Student`, как и свойства `semesterHours` и `gpa`.

Для функций или переменных в программе, которые не являются членом какого-либо класса, нет специального названия, однако в этой книге я буду называть переменные или функции *не членами* класса, если они не были явно описаны в составе какого-либо класса.

Функции-члены не обязаны предшествовать членам-данным в объявлении класса, как сделано в приведенном выше примере. Члены класса перечисляются в любом порядке. Просто я предпочитаю первыми размещать функции.

По историческим причинам функции-члены называют также *методами*. Такое название имеет смысл в других объектно-ориентированных языках программирования, но бессмысленно в C++. Несмотря на это, термин приобрел некоторую популярность и среди программистов на C++, наверное, поскольку его проще выговорить, чем выражение “функция-член” (то, что это звучит гораздо внушительнее, никого не волнует). Так что если во время вечеринки ваши друзья начнут сыпать словечками вроде “методы класса”, просто мысленно замените *методы* выражением *функции-члены*, и все встанет на свои места.

Вызов функций-членов

Приведенная далее программа `CallmemberFunction` показывает, как вызвать функцию-член `addCourse()`:

```
//  CallMemberFunction - определение и вызов
//                       функции-члена класса
//

#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
    public:
        // Добавить пройденный курс к записи
        float addCourse(int hours, float grade)
        {
            // Вычислить среднюю оценку с учетом
            // времени различных курсов
            float weightedGPA;
            weightedGPA = semesterHours * gpa;

            // Добавить новый курс
            semesterHours += hours;
            weightedGPA += grade * hours;
            gpa = weightedGPA / semesterHours;
```

```
            // Вернуть новую оценку
            return gpa;
        }

        int  semesterHours;
        float gpa;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.semesterHours = 10;
    s.gpa           = 3.0;

    // Значения до вызова
    cout << "До: s = (" << s.semesterHours
         << ", "    << s. gpa
         << endl;

    s.addCourse(3, 4.0); // Вызов функции-члена

    // Измененные значения
    cout << "После: s = (" << s.semesterHours
         << ", "    << s. gpa
         << ")"     << endl;

    // Пауза для того, чтобы посмотреть
    // на результат работы программы
    system("PAUSE");
    return 0;
}
```

Как видите, синтаксис вызова функции-члена такой же, как и синтаксис обращения к переменной-члену класса. Часть выражения, которая находится справа от точки, не отличается от вызова обычной функции. Единственное отличие — присутствие слева от точки имени объекта, которому принадлежит функция.

Вызов `s.addCourse()` можно определить так: “`s` является объектом, на который действует `addCourse()`”; или, другими словами, объект `s` представляет собой студента, к записи которого добавляется новый курс. Вы не можете получить информацию о студенте или изменить ее, не указав, о каком конкретно студенте идет речь. Вызов функции-члена без указания имени объекта имеет не больше смысла, чем обращение к данным-членам без указания объекта.

Доступ к членам из функции-члена

Я так и слышу, как вы повторяете про себя: “Нельзя обратиться к функции-члену без указания имени объекта! Нельзя обратиться к функции-члену без указания имени объекта! Нельзя...” Запомнив это, вы смотрите на тело функции-члена `Student::addCourse()` и… что это? Ведь `addCourse()` обращается к членам класса, не уточняя имени объекта!

Возникает вопрос: так все-таки можно или нельзя обратиться к члену класса, не указывая его объекта? Уж поверьте мне, что нельзя. Просто когда вы обращаетесь к члену класса `Student` из `addCourse()`, по умолчанию используется тот экземпляр класса, из которого вызвана функция `addCourse()`. Вы ничего не поняли? Вернемся к примеру.

```
int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.semesterHours = 10;
```

```
    s.gpa      = 3.0;
    s.addCourse(3, 4.0); // Вызов функции-члена

    Student t;
    t.semesterHours = 6;
    t.gpa           = 1.0;
    t.addCourse(3, 1.5);

    system("PAUSE");
    return 0;
}
```

Когда `addCourse()` вызывается для объекта `s`, все сокращенные имена в теле этой функции считаются членами объекта `s`. Таким образом, обращение к переменной `semesterHours` внутри функции `s.addCourse()` в действительности является обращением к переменной `s.semesterHours`, а обращение к `gpa` — обращением к `s.gpa`. В следующей строке функции `main()`, когда `addCourse()` вызывается для объекта `t` того же класса `Student`, происходит обращение к членам класса `t.semesterHours` и `t.gpa`.

Объект, для которого вызывается функция-член, называется "текущим", и все имена членов, записанные в сокращенном виде внутри функции-члена, считаются членами текущего объекта. Другими словами, сокращенное обращение к членам класса интерпретируется как обращение к членам текущего объекта.

Именование текущего объекта

Как функция-член определяет, какой объект является текущим? Это не магия и не шаманство — просто адрес этого объекта *всегда* передается функции-члену как *скрытый* первый аргумент. Другими словами, при вызове функции-члена происходит преобразование такого вида:

`s.addCourse(3,2.5)` равносильно `Student::addCourse(&s,3,2.5)`

(команда, приведенная в правой части выражения, синтаксически неправильна; она просто показывает, как компилятор видит выражение в левой части во внутреннем представлении).

Внутри функции, когда нужно узнать, какой именно объект является текущим, используется этот указатель. Тип текущего объекта — указатель на объект соответствующего класса.

Всякий раз, когда функция-член обращается к другому члену класса, не называя имени его объекта явно, компилятор считает, что данный член является членом *этого* (this) объекта. При желании вы можете явно обращаться к членам *этого* объекта, используя ключевое слово `this`. Так что функцию `Student::addCourse()` можно переписать следующим образом:

```
float Student::addCourse(int hours, float grade)
{
    float weightedGPA;
    weightedGPA = this->semesterHours * this->gpa;

    // добавим новый курс
    this->semesterHours += hours;
    weightedGPA += hours * grade;
    this->gpa = weightedGPA / this->semesterHours;
    return this->gpa;
}
```

Независимо от того, добавите ли вы оператор `this->` в тело функции явно или нет, результат будет одинаковым.

Разрешение области видимости

Символ `::` между именем класса и именем его члена называют *оператором разрешения области видимости*, поскольку он указывает, какой области видимости принадлежит член класса. Имя класса перед двоеточиями похоже на фамилию, тогда как название функции после двоеточия схоже с именем — такой порядок записи принят на Востоке.

С помощью оператора `::` можно также описать функцию — не член, использовав для этого пустое имя класса. В этом случае функция `addCourse()` должна быть описана как `::addCourse(int, float)` — эдакая бездомная функция…

Обычно оператор `::` не обязателен, однако в некоторых ситуациях это не так. Рассмотрим следующий фрагмент кода:

```
// addCourse — перемножает количество часов и оценку
float addCourse(in hours, float grade)
{
    return hours*grade;
}
class Student
{
public:
    // Добавить пройденный курс к записи
    float addCourse(int hours, float grade)
    {
        // Вызвать внешнюю функцию
        weightedGPA = ::addCourse(semesterHours, gpa);
        // Вызвать ту же функцию для подсчета
        // оценки с учетом нового курса
        weightedGPA += ::addCourse(hours, grade);
        gpa = weightedGPA / semesterHours;

       // Вернуть новую оценку
       return gpa;
    }

    int   semesterHours;
    float gpa;
};
```

В этом фрагменте я хотел, чтобы функция-член `Student::addCourse()` вызывала функцию — не член `::addCourse()`. Без оператора `::` вызов функции `addCourse()` внутри класса `Student` приведет к вызову функции `Student::addCourse()`, т.е. самой себя.

Определение функции-члена

Функция-член может быть определена как внутри класса, так и отдельно от него. Когда функция определяется внутри класса, это выглядит так, как в приведенном ниже включаемом файле `savings.h`:

```
// Savings - определение класса с
//           возможностью делать вклады
class Savings
{
  public:
    // Объявляем и определяем функции-члены
    float deposit(float amount)
```

```
    {
        balance += amount;
        return balance;
    }
    unsigned int accountNumber;
    float  balance;
};
```

Встраиваемые функции-члены

Функция-член, определенная непосредственно внутри класса, по умолчанию считается встраиваемой (подставляемой, inline) функцией (если только не оговорено обратное, например с помощью опций командной строки компилятора). Функции-члены, определенные в классе, по умолчанию считаются inline-функциями, потому что многие функции-члены, определенные внутри класса, довольно малы, а такие маленькие функции являются главными кандидатами на подстановку.

Тело встраиваемой функции подставляется компилятором непосредственно вместо оператора ее вызова. (В главе 10, "Препроцессор C++", сравнивались встраиваемые функции и макросы.) Такая функция выполняется быстрее, поскольку от процессора не требуется осуществлять переход к телу функции. Однако при этом программы, использующие встроенные функции, занимают больше места, поскольку копии таких встраиваемых функций определяются один-единственный раз, а подставляются вместо каждого вызова.

Есть еще одна техническая причина, по которой функции-члены класса лучше делать встраиваемыми. Как вы помните, все структуры языка C обычно определяются в составе включаемых файлов с последующим использованием в исходных `.cpp`-файлах при необходимости. Такие включаемые файлы не должны содержать данных или тел функций, поскольку могут быть скомпилированы несколько раз. Использование же подставляемых функций во включаемых файлах вполне допустимо, поскольку их тела, как и макросы, подставляются вместо вызова в исходном файле. То же относится и к классам C++. Подразумевая, что функции-члены, определенные в описании классов, встраиваемые, мы избегаем упомянутой проблемы многократной компиляции.

Использование такого заголовочного файла проще простого — его надо включить в программу и пользоваться определенным в нем классом, как вам заблагорассудится, например, как в приведенной ниже программе `SavingsClass_Inline`.

```
//
//  SavingsClassInline - вызов функции-члена,
//                       объявленной и определенной в
//                       классе Savings
//
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;
#include "Savings.h"

int main(int nNumberofArgs, char* pszArgs[])
{
    Savings s;
    s.accountNumber = 123456;
    s.balance = 0.0;
```

```
    // Добавляем немного на счет...
    cout << "Вкладываем на счет 10 монет"
         << s.accountNumber << endl;
    s.deposit(10);
    cout << "Состояние счета " << s.balance << endl;

    // Пауза для того, чтобы посмотреть
    // на результат работы программы
    system("PAUSE");
    return 0;
}
```

Так использовать класс `Savings` может теперь любой программист, которому доступен соответствующий заголовочный файл, причем ему совершенно не надо вдаваться в детали реализации этого класса.

Директива `#include` заставляет препроцессор перед началом компиляции вставить вместо нее содержимое указанного в ней файла. Подробно об этом рассказывалось в главе 10, “Препроцессор C++”.

Определение функций-членов вне класса

Для больших функций встраивание тела функции непосредственно в определение класса может привести к созданию очень больших и неудобочитаемых определений классов. Чтобы избежать этого, C++ предоставляет возможность определять тела функций-членов вне класса.

В этом случае в заголовочном файле имеется только объявление, но не определение функции.

```
// Savings - определение класса с
//           возможностью делать вклады
class Savings
{
  public:
    // Объявляем, но не определяем функции-члены
    float deposit(float amount);
    unsigned int accountNumber;
    float  balance;
};
```

Теперь объявление класса содержит только прототип функции `deposit()`. При этом само тело функции находится в другом месте. Для простоты я определил функцию в том же файле, где находится и функция `main()`.

Так можно делать, но подобное расположение функции не очень распространено. Обычно класс определяется в заголовочном файле, а тело функции находится в отдельном исходном файле. Сама же использующая этот класс программа располагается в файле, отличном от этих двух (подробнее об этом будет рассказано в главе 21, “Разложение классов”).

```
//
//  SavingsClassOutline - вызов функции-члена,
//                        объявленной и определенной в
//                        классе Savings
//
#include <cstdio>
#include <cstdlib>
```

```
#include <iostream>

using namespace std;
#include "Savings.h"

// Определение функции-члена Savings::deposit()
// (обычно содержится в отдельном файле)
float Savings::deposit(float amount)
{
    balance += amount;
    return balance;
}

// Основная программа
int main(int nNumberofArgs, char* pszArgs[])
{
    Savings s;
    s.accountNumber = 123456;
    s.balance = 0.0;

    // Добавляем немного на счет...
    cout << "Вкладываем на счет 10 монет"
         << s.accountNumber << endl;
    s.deposit(10);
    cout << "Состояние счета " << s.balance << endl;

    // Пауза для того, чтобы посмотреть
    // на результат работы программы
    system("PAUSE");
    return 0;
}
```

Определение класса содержит только прототип функции `deposit()`, а ее тело определено в другом месте. Такое объявление аналогично объявлению любого другого прототипа.

Обратите внимание, что при определении функции-члена `deposit()` потребовалось указание ее полного имени; сокращенного имени при определении вне класса недостаточно, и нам пришлось использовать имя `Savings::deposit()`.

Перегрузка функций-членов

Функции-члены могут перегружаться так же, как и обычные функции (обратитесь к главе 6, “Создание функций”, если забыли, что это значит). Как вы помните, имя класса является частью полного имени, и все приведенные ниже функции вполне корректны.

```
class Student
{
public:
    //grade – возвращает текущую среднюю оценку
    float grade();
    //grade – устанавливает новое значение
    //оценки и возвращает предыдущее
    float grade(float newGPA);
    //... прочие члены-данные ...
};
class Slope
{
public:
```

```
    //grade — возвращает снижение оценки
    float grade();
    //...прочие члены-данные...
};

//grade — возвращает символьный эквивалент оценки
char grade(float value);

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.grade(3.5);       //Student::grade(float)
    float v = s.grade();//Student::grade()

    char c = grade(v);  //::grade(float)

    Slope o;
    float m = o.grade();//Slope::grade()
    return 0;
}
```

Полные имена вызываемых из `main()` функций указаны в комментариях.

Когда происходит вызов перегруженной функции, составляющими ее полного имени считаются не только аргументы функции, но и тип объекта, который вызывает функцию (если она вызывается объектом). Такой подход позволяет устранить неоднозначность при вызове функции.

В приведенном примере первые два вызова обращаются к функциям-членам `Student::grade(float)` и `Student::grade()` соответственно. Эти функции отличаются списками аргументов. Вызов функции `s.grade()` обращается к `Student::grade()`, поскольку тип объекта `s` — `Student`.

Третья вызываемая функция в данном примере — функция `::grade(float)`, не имеющая вызывающего объекта. Последний вызов осуществляется объектом типа `Slope` и соответственно вызывается функция-член `Slope::grade(float)`. Этот процесс называется *разрешением* вызова (определением в процессе компиляции, какая из перегруженных функций должна вызываться).