Глава 2

Введение в классы

В этой главе вводятся понятия классов и объектов. В следующих нескольких важнейших разделах фактически описаны почти все аспекты программирования на C++, поэтому советуем вам читать повнимательнее.

2.1. Конструкторы и деструкторы

Если вы писали очень длинные программы, то знаете, что в некоторых частях программы обычно требуется инициализация. Необходимость в инициализации еще более часто проявляется при работе с объектами. Действительно, если обратиться к реальным проблемам, то, фактически, для каждого создаваемого вами объекта требуется какого-то вида инициализация. Для разрешения этой проблемы в C++ имеется *функция-конструктор (constructor function),* включаемая в описание класса. Конструктор класса вызывается всякий раз при создании объекта этого класса. Таким образом, любая необходимая объекту инициализация при наличии конструктора выполняется автоматически.

Конструктор имеет то же имя, что и класс, частью которого он является, и не имеет возвращаемого значения. Например, ниже представлен небольшой класс с конструктором:

```
#include <iostream>
using namespace std;
class myclass {
        int a;
public:
        myclass(); // конструктор
        void show() ;
};
myclass::myclass()
(
        cout << "В конструкторе\n" ;
        a=10;
} void myclass::show()
{
        cout << a;
}
int main()
{
       myclass ob;
        ob.show() ;
return 0;
}
```

В этом простом примере значение a инициализируется конструктором myclass(). Конструктор вызывается тогда, когда создается объект ob. Объект, в свою очередь, создается при выполнений инструкции объявления объекта. Важно понимать, что в C++ инструкция объявления переменной является "инструкцией действия". При программировании на C инструкции объявления переменных понимаются просто как создание переменных. Однако в C++, поскольку объект может иметь конструктор,

инструкция объявления переменной может вызывать выполнение записанных в конструкторе действий.

Обратите внимание, как определяется конструктор myclass(). Как уже говорилось, он не имеет возвращаемого значения. В соответствии с формальными правилами синтаксиса C++ конструктор не должен иметь возвращаемого значения.

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

Функцией, обратной конструктору, является *деструктор (destructor).* Эта функция вызывается при удалении объекта. Обычно при работе с объектом в момент его удаления должны выполняться некоторые действия. Например, при создании объекта для него выделяется память, которую необходимо освободить при его удалении. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале. Пример класса с деструктором:

```
#include <iostream>
using namespace std;
class myclass {
        int a;
public:
        myclass(); // конструктор
        ~myclass'(); // деструктор
        void show() ;
};
myclass::myclass()
{
        cout « "Содержимое конструктора\n" ;
        a = 10;
}
myclass:: myclass ()
{
        cout << "Удаление...\n";
}
void myclass::show()
{
        cout << a << "\n";
}
int main()
{
        myclass ob;
        ob.show() ;
        return 0;
}
```

Деструктор класса вызывается при удалении объекта. Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы.

Адреса конструктора и деструктора получить невозможно.

Примеры

1. Вспомните, что в созданном в главе 1 классе **stack** для установки переменной индекса

стека требовалась функция инициализации. Это Именно тот тип действия, для выполнения которого и придуман конструктор. Здесь представлена улучшенная версия класса **stack,** где для автоматической инициализации объекта стека при его создании используется конструктор:

```
#include <iostream>
using namespace std;

#define SIZE 10

// Объявление класса stack для символов
class stack {
   char stck[SIZE]; // содержит стек
   int tos; // индекс вершины стека
public:
   stack(); // конструктор
   void push(char ch); // помещает в стек символ
   char pop(); // выталкивает из стека символ
};

// Инициализация стека
stack::stack()
{
   cout << "Работа конструктора стека \n";
   tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
{
   if (tos==SIZE) {
      cout << "Стек полон";
      return;
   }
   stck[tos]=ch;
   tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
   if (tos==0) {
      cout << "Стек пуст";
      return 0; // возврат нуля при пустом стеке
   }
   tos--;
   return stck[tos];
}

int main()
{
   // образование двух, автоматически инициализируемых, стеков
   stack s1, s2;
   int i;

   s1.push('a');
   s2.push('x');
   s1.push('b');
   s2.push('y');
   s1.push('c');
   s2.push('z');
```

```
   for(i=0; i<3; i++) cout << "символ из s1:" << s1.pop() << "\n";
   for(i=0; i<3; i++) cout << "символ из s2:" << s2.pop() << "\n";

   return 0;
}
```

Обратите внимание, что теперь вместо отдельной, специально вызываемой программой функции задача инициализации выполняется конструктором автоматически. Это важное положение. Если инициализация выполняется автоматически при создании объекта, то это исключает любую возможность того, что по ошибке инициализация не будет выполнена. Вам, как программисту, не нужно беспокоиться об инициализации — она автоматически выполнится при появлении объекта.

2. В следующем примере показана необходимость не только конструктора, но и деструктора. В примере создается простой класс для строк, который содержит саму строку и ее длину. Когда создается объект **strtype**, для хранения строки выделяется память, и начальная длина строки устанавливается равной нулю. Когда объект **strtype** удаляется, эта память освобождается.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

#define SIZE 255

class strtype {
   char *p;
   int len;
public:
   strtype(); // конструктор
   ~strtype(); // деструктор
   void set(char *ptr);
   void show();
};

// Инициализация объекта строка
strtype::strtype()
{
   p=(char *) malloc(SIZE);
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }
   *p='\0';
   len=0;
}

// Освобождение памяти при удалении объекта строка
strtype::~strtype()
{
   cout << "Освобождение p\n";
   free(p);
}

void strtype::set(char *ptr)
{
   if(strlen(ptr) > SIZE) {
      cout << "Строка слишком велика\n";
      return;
```

```
    }
    strcpy(p, ptr);
    len=strlen(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1,s2;

    s1.set("Это проверка");
    s2.set("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

В этой программе для выделения и освобождения памяти используются функции **malloc()** и **free().** Хотя этот пример совершенно правилен, как вы увидите далее в этой книге, в C++ есть и иной путь управления распределением динамической памяти.

3. В следующем примере приведен интересный способ использования конструктора и деструктора объекта. В программе объект класса **timer** предназначен для измерения временного интервала между его созданием и удалением. При вызове деструктора на экран выводится прошедшее с момента создания объекта время. Вы могли бы воспользоваться подобным объектом для измерения времени работы программы или времени работы функции внутри блока. Просто убедитесь, что объект исчезает в момент завершения временного интервала.

```
#include <iostream>
#include <ctime>
using namespace std;

class timer {
    clock_t start;
public:
    timer(); // конструктор
    ~timer(); // деструктор
};

timer::timer()
{
    start=clock();
}

timer::~timer()
{
    clock_t end;

    end=clock();
    cout << "Затраченное время: " << (end-start) / CLOCKS_PER_SEC <<"\n";
}
```

```
int main()
{
   timer ob;
   char c;

   // Пауза ...
   cout << "Нажмите любую клавишу, затем ENTER: ";
   cin >> c;

   return 0;
}
```

В программе используется стандартная библиотечная функция clock(), которая возвращает число временных циклов с момента запуска программы. Если разделить это число на **CLOCKS_PER_SEC,** можно получить значение в секундах.

2.2. Конструкторы с параметрами

Конструктору можно передавать аргументы- Для этого просто добавьте необходимые параметры в объявление и определение конструктора. Затем при объявлении объекта задайте параметры в качестве аргументов. Чтобы понять, как это делается, начнем с короткого примера:

```
   #include <iostream>
   using namespace ,std;
   class myclass {
         int a;
   public:
        myclass.(int x); // конструктор
        void show();
   };
   myclass::myclass(int x)
   {
         cout « "В конструкторе\n";
         a = x;
void myclass::show()
cout << a << "\n";

int main()
{
         myclass ob(4);
         ob.show() ;
         return 0;
}
```

Здесь конструктор класса **myclass** имеет один параметр. Значение, передаваемое в **myclass(),** используется для инициализации переменной а. Обратите особое внимание на то, как в функции **main()** объявляется объект **ob.** Число 4, записанное в круглых скобках, является аргументом, передаваемым параметру х конструктора **myclassQ,** который используется для инициализации переменной **а**.

Фактически синтаксис передачи аргумента конструктору с параметром является сокращенной формой записи следующего, более длинного выражения:

```
myclass ob = myclass(4);
```

Однако большинство программистов C++ пользуются сокращенной формой записи. На самом деле, с технической точки зрения между этими двумя формами записи имеется небольшое отличие, связанное с конструктором копий (copy constructor), о котором будет рассказано позже. На данном этапе об этом отличии можно не беспокоиться.

Примеры

1. Вполне допустимо передавать конструктору несколько аргументов. В этом примере конструктору **myclass()** передается два аргумента:

```
#include <iostream>
using namespace std;

class myclass {
   int a, b;
public:
   myclass(int x, int y); // конструктор
   void show();
};

myclass::myclass(int x, int y)
{
   cout << "В конструкторе\n";
   a = x;
   b = y;
}

void myclass::show()
{
   cout << a << ' ' << b << "\n";
}

int main()
{
   myclass ob(4, 7);

   ob.show();

   return 0;
}
```

Здесь 4 передается в **x**, а 7 передается в **y**. Такой же общий подход используется для передачи любого необходимого числа аргументов (ограниченного, разумеется, возможностями компилятора).

2. Здесь представлена следующая версия класса **stack,** в котором конструктор с параметром используется для присвоения стеку "имени". Это односимвольное имя необходимо для идентификации стека в случае возникновения ошибки.

```
#include <iostream>
using namespace std;

#define SIZE 10

// Объявление класса stack для символов
class stack {
   char stck[SIZE]; // содержит стек
   int tos; // индекс вершины стека
```

```
    char who; // идентифицирует стек
public:
    stack(char c); // конструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};

// Инициализация стека
stack::stack(char c)
{
    tos = 0;
    who = c;
    cout << "Работа конструктора стека " << who << "\n";
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек " << who << " полон \n";
        return;
    }
    stck[tos]=ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек " << who << " пуст ";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // образование двух, автоматически инициализируемых, стеков
    stack s1('A'), s2('B');
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    // Это вызовет сообщения об ошибках
    for(i=0; i<5; i++) cout << "символ из стека s1: " << s1.pop() << "\n";
    for(i=0; i<5; i++) cout << "символ из стека s2: " << s2.pop() << "\n";

    return 0;
}
```

Присвоение "имени" объекту, как показано в примере, является особенно полезным при отладке, когда важно выяснить, какой из объектов вызывает ошибку.

3. Здесь показан новый вариант разработанного ранее класса **strtype**, в котором используется конструктор с параметром:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
   char *p;
   int len;
public:
   strtype(char *ptr);
   ~strtype();
   void show();
};

strtype::strtype(char *ptr)
{
   len = strlen(ptr);
   p = (char *) malloc(len + 1);
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }
   strcpy(p, ptr);
}

strtype::~strtype()
{
   cout << "Освобождение p\n";
   free(p);
}

void strtype::show()
{
   cout << p << "- длина: " << len;
   cout << "\n";
}

int main()
{
   strtype s1("Это проверка"), s2("Мне нравится C++");

   s1.show();
   s2.show();

   return 0;
}
```

В этой версии класса **strtype** строка получает свое начальное значение с помощью конструктора.

4. Конструктору объекта можно передать не только константы, но и любые допустимые выражения с переменными. Например, в следующей программе для создания объекта используется пользовательский ввод:

```
#include <iostream>
using namespace std;

class myclass {
   int i, j;
public:
   myclass(int a, int b);
```

```
  void show();
};

myclass::myclass(int a, int b)
{
  i = a;
  j = b;
}

void myclass::show()
{
  cout << i << ' ' << j << "\n";
}

int main()
{
  int x, y;

  cout << "Введите два целых: ";
  cin >> x >> y;

  // использование переменных для создания ob
  myclass ob(x, y);

  ob.show();

  return 0;
}
```

2.3. Введение в наследование

Хотя более полно наследование (inheritance) обсуждается в главе 7, предварительно с ним необходимо познакомиться уже сейчас. Применительно к C++ наследование — это механизм, посредством которого один класс может наследовать свойства другого. Наследование позволяет строить иерархию классов, переходя от более общих к более специальным.

Для начала необходимо определить два термина, обычно используемые при обсуждении наследования. Когда один класс наследуется другим, класс, который наследуется, называют *базовым классом (base class).* Наследующий класс называют *производным классом (derived class).* Обычно процесс наследования начинается с задания базового класса. Базовый класс определяет все те качества, которые будут общими для всех производных от него классов. В сущности, базовый класс представляет собой наиболее общее описание ряда характерных черт. Производный класс наследует эти общие черты и добавляет свойства, характерные только для него.

Чтобы понять, как один класс может наследовать другой, давайте начнем с .примера, который, несмотря на свою простоту, иллюстрирует несколько ключевых положений наследования.

Для начала — объявление базового класса:

```
// Определение базового класса
class B {
        int i;
public:
```

```
        void set i(int n);
        int get_i();
};
```

Теперь объявим производный класс, наследующий этот базовый:

```
// Определение производного класса

class D: public B {
        int j;
public:
        void set j(int n);
        int mul();
};
```

Внимательно посмотрите на это объявление. Обратите внимание, что после имени класса D имеется двоеточие, за которым следует ключевое слово **public** и имя класса B. Для компилятора это указание на то, что класс D будет наследовать все компоненты класса B. Само ключевое слово **public** информирует компилятор о том, что, поскольку класс B будет наследоваться, значит, все открытые элементы базового класса будут также открытыми элементами производного класса. Однако все закрытые элементы базового класса останутся закрытыми и к ним не будет прямого доступа из производного класса.
Ниже приводится законченная программа, в которой используются классы **B** и **D**:

```
//Простой пример наследования
#include <iostream>
using namespace std;
//.Определение базового класса
class B {
        int i;
public:
        void set (int n) ;
        int get i();
'}
//Определение производного класса
class D; public B {
        int j;
public:
        voidset_j (int n) ;
        int mul();
   };
// Задание значения i в базовом классе
void B::set_i(int n)
   {
        i = n;
  // Возвращение значения i в базовом классе
  int B:;get_i()
   }
        return i;
}
    // Задание значения j в производном классе
void D::set j(int n)
{
        j = n;
}
    // Возвращение значения i базового класса и j — производного
int D: :mul()
{
        // производный класс может
        // вызывать функции-члены базового класса
        return j * get_i {) ;
}
```

```
int main()
{
        D ob;

        ob.set_i(10); // загрузка i в базовый класс
        ob.set_j(4); // загрузка j в производный класс
        cout << ob.mul(); // вывод числа 40
        return 0;
}
```

Обратите внимание на определение функции **mul().** Отметьте, что функция **get_i**, которая является членом базового класса **B,** а не производного **D,** вызывается внутри класса D без всякой связи с каким бы то ни было объектом. Это возможно потому, что открытые члены класса B становятся открытыми членами класса D. В функции **mul()** вместо прямого доступа к **i,** необходимо вызывать функцию **get_i(),** поскольку закрытые члены базового класса (в данном случае i) остаются закрытыми для нее и недоступными из любого производного класса. Причина, по которой закрытые члены класса становятся недоступными для производных классов — поддержка инкапсуляции. Если бы закрытые члены класса становились открытыми просто посредством наследования этого класса, инкапсуляция была бы совершенно несостоятельна.

Здесь показана основная форма наследования базового класса:

class имя_производного_класса: с_д иля_базового_класса {
.
.
.
};

Здесь *с_д* (спецификатор доступа) — это одно из следующих трех ключевых слов: **public (открытый), private (закрытый) или protected (защищенный).**

В данном случае для наследования класса используется именно public. Полное описание этих спецификаторов доступа будет дано позже.

Примеры

1. Ниже приведена программа, которая определяет общий базовый класс fruit, описывающий некоторые характеристики фруктов. Этот класс наследуется двумя производными классами **Apple** и **Orange**. Эти классы содержат специальную информацию о конкретном фрукте (яблоке или апельсине).

```
// Пример наследования классов
#include <iostream>
#include <cstring>
using namespace std;

enum yn {no, yes};
enum color {red, yellow, green, orange};

void out(enum yn x);

char *c[ ] = {
   "red", "yellow", "green", "orange"};
```

```
// Родовой класс фруктов
class fruit {
// В этом базовом классе все элементы открыты
public:
   enum yn annual;
   enum yn perennial;
   enum yn tree;
   enum yn tropical;
   enum color clr;
   char name[40];
};

// Производный класс яблок
class Apple : public fruit {
   enum yn cooking;
   enum yn crunchy;
   enum yn eating;
public:
   void seta(char *n, enum color c, enum yn ck, enum yn crchy, enum yn e);
   void show();
};

// Производный класс апельсинов
class Orange : public fruit {
   enum yn juice;
   enum yn sour;
   enum yn eating;
public:
   void seto(char *n, enum color c, enum yn j, enum yn sr, enum yn e);
   void show();
};

void Apple::seta(char *n, enum color c, enum yn ck, enum yn crchy, enum yn e)
{
   strcpy(name, n);
   annual = no;
   perennial = yes;
   tree = yes;
   tropical = no;
   clr = c;
   cooking = ck;
   crunchy = crchy;
   eating = e;
}

void Orange::seto(char *n, enum color c, enum yn j, enum yn sr, enum yn e)
{
   strcpy(name, n);
   annual = no;
   perennial = yes;
   tree = yes;
   tropical = yes;
   clr = c;
   juice = j;
   sour = sr;
   eating = e;
}

void Apple::show()
{
   cout << name << " яблоко - это: " << "\n";
   cout << "Однолетнее растение: "; out(annual);
   cout << "Многолетнее растение: "; out(perennial);
```

```
    cout << "Дерево: "; out(tree);
    cout << "Тропическое: "; out(tropical);
    cout << "Цвет: " << c[clr] << "\n";
    cout << "Легко приготавливается: "; out(cooking);
    cout << "Хрустит на зубах: "; out(crunchy);
    cout << "Съедобное: "; out(eating);
    cout << "\n";
}

void Orange::show()
{
    cout << name << " апельсин - это: " << "\n";
    cout << "Однолетнее растение: "; out(annual);
    cout << "Многолетнее растение: "; out(perennial);
    cout << "Дерево: "; out(tree);
    cout << "Тропическое: "; out(tropical);
    cout << "Цвет: " << c[clr] << "\n";
    cout << "Годится для приготовления сока: "; out(juice);
    cout << "Кислый: "; out(sour);
    cout << "Съедобный: "; out(eating);
    cout << "\n";
}

void out(enum yn x)
{
    if (x==no) cout << "нет\n";
    else cout << "да\n";
}

int main()
{
    Apple a1, a2;
    Orange o1, o2;

    a1.seta("Красная прелесть", red, no, yes, yes);
    a2.seta("Джонатан", red, yes, no, yes);

    o1.seto("Пуп", orange, no, no, yes);
    o2.seto("Валенсия", orange, yes, yes, no);

    a1.show();
    a2.show();

    o1.show();
    o2.show();

    return 0;
}
```

Как можно заметить, базовый класс **fruit** определяет несколько свойств, характерных для фруктов любого типа. (Конечно, чтобы 'сократить пример и таким образом приспособить его для книги, класс **fruit** отчасти упрощен.) Например, все фрукты растут на однолетних или многолетних растениях. Все фрукты растут на деревьях или на растениях другого типа, таких как лоза или куст. Все фрукты имеют цвет и название. Затем такой базовый класс наследуется классами **Apple и Orange.** Каждый из этих классов обеспечивает объект информацией, характерной для фруктов конкретного типа.

Этот пример иллюстрирует основной смысл наследования. Создаваемый здесь базовый класс определяет основные черты, связанные со *всеми* фруктами.

Производным классам предоставляется возможность обеспечения тех Характеристик, которые являются характерными в каждом *конкретном* случае.

Эта программа раскрывает другой важный аспект наследования; производные классы не "владеют" базовым классом безраздельно. Он может наследоваться любым количеством классов.

2.4. Указатели на объекты

До сих пор вы осуществляли доступ к членам объекта с помощью оператора точка (.). Если вы работаете с объектом, то это правильно. Однако доступ к члену объекта можно получить также и через указатель на этот объект. В этом случае обычно применяется оператор стрелка (->). (Аналогичным способом оператор стрелка (">) используется при работе с указателем на структуру.)

Вы объявляете указатель на объект точно так же, как и указатель на переменную любого другого типа. Задайте имя класса этого объекта, а затем имя переменной со звездочкой перед ним. Для получения адреса объекта перед ним необходим оператор ***&,*** точно так же, как это делается для получения адреса переменной другого типа.

Как и для любого другого, указателя, если вы инкрементируете указатель на объект, он будет указывать на следующий объект такого же типа.

Примеры

1. Простой пример использования указателя на объект:

```
#include <iostream>
using namespace std;

class myclass {
   int a;
public:
   myclass(int x); // конструктор
   int get();
};

myclass::myclass(int x)
{
   a = x;
}

int myclass::get()
{
   return a;
}

int main()
{
   myclass ob(120); // создание объекта
   myclass *p; // создание указателя на объект

   p = &ob; // передача адреса ob в p

   cout << "Значение, получаемое через объект:" << ob.get();
```

```
    cout << "\n";

    cout << "Значение, получаемое через указатель:" << p->get();

    return 0;
}
```

Отметьте, как объявление

```
myclass *p;
```

создает указатель на объект класса **myclass.** Это важно для понимания следующего положения: создание указателя на объект *не* создает объекта, оно создает только указатель на него.
Для передачи адреса объекта **ob** в переменную **p,** использовалась инструкция:

```
p = &ob;
```

И последнее, в программе показано, как можно получить доступ к членам объекта с помощью указателя на этот объект.

Мы вернемся к обсуждению указателей на объекты в главе 4, когда у вас уже будет более полное представление о C++.

2.5. Классы, структуры и объединения

Как вы уже заметили, синтаксически класс похож на структуру. Вас, возможно, удивило то, что класс и структура имеют фактически одинаковые свойства. В C++ определение структуры расширили таким образом, что туда, как и в определение класса, удалось включить функции-члены, в том числе конструкторы и деструкторы. Таким образом, единственным отличием между структурой и классом является то, что члены класса, по умолчанию, являются закрытыми, а члены структуры ~ открытыми. Здесь показан расширенный синтаксис описания структуры:

```
struct имя_типа {
        // открытые функции и данные — члены класса
private:
        // закрытие функции и данные — члены класса
} список объектов
```

Таким образом, в соответствии с формальным синтаксисом C++ как структура, так и класс создают новые *типы* данных. Обратите внимание на введение нового ключевого слова. Им является слово **private,** которое сообщает компилятору, что следующие за ним члены класса являются закрытыми.

В том, что структуры и классы обладают фактически одинаковыми свойствами, имеется кажущаяся избыточность. Те, кто только знакомится с C++, часто удивляются этому дублированию. При таком взгляде на проблему уже не кажутся необычными рассуждения о том, что ключевое слово **class** совершенно лишнее.

Объяснение этому может быть дано в "строгой" и "мягкой" формах. "Строгий" довод состоит в том, что необходимо поддерживать линию на совместимость с C. Стиль задания структур C совершенно допустим и для программ C++. Поскольку в C все члены структур

по умолчанию открыты, это положение также поддерживается и в C++. Кроме этого, поскольку класс синтаксически отличается от структуры, определение класса открыто Для развития в направлении, которое в конечном итоге может привести к несовместимости со взятым из C определением структуры. Если эти два пути разойдутся, то направление, связанное с C++, уже не будет избыточным.

"Мягким" доводом в пользу наличия двух сходных конструкций стало отсутствие какого-либо ущерба от расширения определения структуры в C++ таким образом, что в нее стало возможным включение функций-членов.

Хотя структуры имеют схожие с классами возможности, большинство программистов ограничивают использование структур взятыми из C формами и не применяют их для задания функций-членов. Для задания объекта, содержащего данные и код, эти программисты обычно указывают ключевое слово **class.** Однако все это относится к стилистике и является предметом вашего собственного выбора. (Далее в книге за исключением текущего раздела с помощью ключевого **слова, struct** задаются объекты, которые не имеют функций-членов.)

Если вы нашли интересной связь между классами и структурами, вас заинтересует и следующее необычное свойство C++: объединения и классы в этом языке столь же близки! В C++ объединение также представляет собой тип класса, в котором функции и данные могут содержаться в качестве его членов. Объединение похоже на структуру тем, что в нем по умолчанию все члены открыты до тех пор, пока не указан спецификатор **private.** Главное же в том, что в C++ все данные, которые являются членами объединения, находятся в одной и той же области памяти (точно так же, как и в C). Объединения могут содержать конструкторы и деструкторы. Объединения C++ совместимы с объединениями C.

Если в отношениях между структурами и классами существует, на первый взгляд, некоторая избыточность, то об объединениях этого сказать нельзя. В объектно-ориентированном языке важна поддержка инкапсуляции. Поэтому способность объединений связывать воедино программу и данные позволяет создавать такие типы классов, в которых все данные находятся в общей области памяти. Это именно то, чего нельзя сделать с помощью классов.

Применительно к C++ имеется несколько ограничений, накладываемых на использование объединений. Во-первых, они не могут наследовать какой бы то ни было класс и не могут быть базовым классом для любого другого класса. Объединения не могут иметь статических членов. Они также не должны содержать объектов с конструктором или деструктором, хотя сами по себе объединения *могут* иметь конструкторы и деструкторы.
В C++ имеется особый тип объединения — это *анонимное объединение (anonymous union).* Анонимное объединение не имеет имени типа и следовательно нельзя объявить переменную такого типа. Вместо этого анонимное объединение просто сообщает компилятору, что все его члены будут находиться в одной и той же области памяти. Во всех остальных отношениях члены объединения действуют и обрабатываются как самые обычные переменные. То есть, доступ к членам анонимного объединения осуществляется непосредственно, без использования оператора точка (.)- Например, рассмотрим следующий фрагмент:

```
union { // анонимное объединение
        int i;
char ch[4];
};
// непосредственный доступ к переменным i и ch
```

```
i = 10;
ch[0] = 'X';
```

Обратите внимание на то, что, поскольку переменные **i и ch не** являются частью какого бы то ни было объекта, доступ к ним осуществляется непосредственно. Тем не менее, они находятся в одной и той же области памяти.
Смысл анонимных объединений в том и состоит, что они обеспечивают простой способ сообщить компилятору о необходимости разместить одну или несколько переменных в одной и той же области памяти. Исключая эту особенность, члены анонимного объединения больше ничем не отличаются от других переменных.

Все те ограничения, которые накладываются на использование обычных объединений, применимы и к анонимным объединениям. Кроме этого к ним добавлено еще несколько. Глобальное анонимное объединение должно быть объявлено как статическое. Анонимное объединение не может содержать закрытых членов. Имена членов анонимного объединения не должны конфликтовать с другими идентификаторами той же области видимости.

Примеры

1. Ниже представлена короткая программа, в которой для создания класса используется ключевое слово **struct:**

```
#include <iostream>
#include <cstring>
using namespace std;

// использование структуры для определения типа класса
struct st_type {
   st_type(double b, char *n);
   void show();
private:
   double balance;
   char name[40];
};

st_type::st_type(double b, char *n)
{
   balance = b;
   strcpy(name, n);
}

void st_type::show()
{
   cout << "Имя:" << name;
   cout << ":$" << balance;
   if(balance < 0.0) cout << "***";
   cout << "\n";
}

int main()
{
   st_type     acc1(100.12, "Johnson");
   st_type     acc2(-12.34, "Hedricks");

   acc1.show();
   acc2.show();
   return 0;
```

```
}
```

Отметьте, что, как уже говорилось, члены структуры по умолчанию являются открытыми. Для объявления закрытых членов необходимо использовать
ключевое слово **private.**

Кроме этого отметьте существенное отличие между структурами С и структурами С++. В С++ имя тега становится также и законченным именем типа данных, которое можно использовать для объявления объектов. В С, чтобы имя тега стало законченным именем типа данных, перед ним надо указывать ключевое слово **struct.**

Ниже представлена только что рассмотренная программа, но вместо структуры здесь используется класс:

```
#include <iostream>
#include <cstring>
using namespace std;

class sl_type {
   double balance;
   char name[40];
public:
   sl_type(double b, char *n);
   void show();
};

sl_type::sl_type(double b, char *n)
{
   balance = b;
   strcpy(name, n);
}

void sl_type::show()
{
   cout << "Имя:" << name;
   cout << ":$" << balance;
   if(balance < 0.0) cout << "***";
   cout << "\n";
}

int main()
{
   sl_type    acc1(100.12, "Johnson");
   sl_type    acc2(-12.34, "Hedricks");

   acc1.show();
   acc2.show();

   return 0;
}
```

2. Пример использования объединения для побайтного вывода значения типа **double** в двоичном представлении:

```
#include <iostream>
using namespace std;

union bits {
   bits(double n);
```

```
    void show_bits();
    double d;
    unsigned char c[sizeof (double)];
};

bits::bits(double n)
{
    d = n;
}

void bits::show_bits()
{
    int i, j;

    for( j = sizeof (double) - 1; j >= 0; j --) {
        cout << "Двоичное представление байта" << j << ":";
        for( i = 128; i; i >>= 1)
            if(i & c[ j ]) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}

int main()
{
    bits ob(1991.829);

    ob.show_bits();

    return 0;
}
```

Результат работы программы:

```
Двоичное представление байта 7:  01000000
Двоичное представление байта 6:  10011111
Двоичное представление байта 5:  00011111
Двоичное представление байта 4:  01010000
Двоичное представление байта  3: 11100101
Двоичное представление байта 2:  01111111
Двоичное представление байта 1:  01000001
Двоичное представление байта  0: 10001001
```

3. Структуры и объединения могут иметь конструкторы и деструкторы. В следующем примере класс **strtype** переделан в структуру. В структуре имеются конструктор и деструктор.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

struct strtype {
    strtype(char *ptr);
    ~strtype();
    void show();
private:
    char *p;
    int len;
};
```

```
strtype::strtype(char *ptr)
{
   len = strlen(ptr);
   p = (char *) malloc(len + 1);
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }
   strcpy(p, ptr);
}

strtype::~strtype()
{
   cout << "Освобождение p\n";
   free(p);
}

void strtype::show()
{
   cout << p << "- длина: " << len;
   cout << "\n";
}

int main()
{
   strtype s1("Это проверка"), s2("Мне нравится C++");

   s1.show();
   s2.show();

   return 0;
}
```

4. В следующей программе для побайтного вывода на экран значения типа **double** используется анонимное объединение. (Предполагается, что длина значения типа **double** равна восьми байтам.)

```
// Использование анонимного объединения
#include <iostream>
using namespace std;

int main()
{
   union {
      unsigned char bytes[8];
      double value;
   };
   int i;
   value = 859345.324;

   // побайтный вывод значения типа double
   for(i=0; i<8; i++)
      cout << (int) bytes[i] << " ";

   return 0;
}
```

Обратите внимание, что доступ к переменным **value** и **bytes** осуществляется так, как если бы они были не частью объединения, а обычными переменными. Несмотря на то, что эти переменные объявлены как часть анонимного объединения, их имена находятся в той же области видимости, что и другие объявленные здесь локальные переменные. Именно по этой причине член анонимного объединения не может иметь то же имя, что и любая

переменная в данной области видимости.

2.6. Встраиваемые функции

Перед тем как продолжить исследование классов необходимо краткое отступление. В C++ можно задать функцию, которая на самом деле не вызывается, а ее тело встраивается в программу в месте ее вызова. Она действует почти так же, как макроопределение с параметрами в C. Преимуществом встраиваемых *(in-line)* функций является то, что они не связаны с механизмом вызова функций и возврата ими своего значения. Это значит, что встраиваемые функции могут выполняться гораздо быстрее обычных. (Запомните, что выполнение машинных команд, которые генерируют вызов функции и возвращение функцией своего значения, занимает определенное время. Если функция имеет параметры, то ее вызов занимает еще большее время.)

Недостатком встраиваемых функций является то, что если они слишком большие и вызываются слишком часто, объем ваших программ сильно возрастает. Из-за этого применение встраиваемых функций обычно ограничивается короткими функциями.
Для объявления встраиваемой функции просто впишите спецификатор inline перед определением функции. Например, в этой короткой программе покзано, как объявить встраиваемую функцию:

```
// Пример встраиваемой функции
 #include <iostream>
using namespace std;
     inline int even(int x) {
          return ! (x%2);
    }
int main()
{
      if (even(10)) cout « "10 является четным \n';
      if (even(ll)) cout <<; "11 является четным\n'
      return 0;
}
```

В этом примере функция **even(),** которая возвращает истину при четном аргументе, объявлена встраиваемой. Это означает, что строка

```
if (even(10)) cout « "10 является четным\n";
```

функционально идентична строке

```
if (! (10%2)) cout « "10 является четным\n";
```

Этот пример указывает также на другую важную особенность использования встраиваемой функции: она должна быть задана до ее первого вызова. Если это не так, компилятор не будет знать, какой именно код предполагается встроить в программу с помощью встраиваемой функции. Поэтому функция **even()** была определена перед функцией **main().**

В пользу использования встраиваемых функций вместо макроопределений с параметрами имеется два довода. Во-первых, они обеспечивают более стройный способ встраивания в программу коротких фрагментов кода. Например, при создании макроса с параметрами легко забыть, что для гарантии правильности встраивания в каждом случае часто требуются круглые внешние скобки. Встраиваемая функция исключает эту проблему.

Во-вторых, компилятор гораздо лучше работает со встраиваемой функцией, чем с макрорасширением. Как правило, программисты C++ для многократных вызовов коротких функций вместо макросов с параметрами практически всегда используют встраиваемые функции.

Здесь важно понимать, что спецификатор **inline** для компилятора является запросом, а не командой. Если, по разным причинам, компилятор не в состоянии выполнить запрос, функция будет компилироваться, как обычная функция, а запрос **inline** будет проигнорирован.

В зависимости от типа вашего компилятора возможны некоторые ограничения на использование встраиваемых функций. Например, некоторые компиляторы не воспринимают функцию как встраиваемую, если функция является рекурсивной или если она содержит либо статическую **(static)** переменную, либо любую инструкцию выполнения цикла, либо инструкцию **switch**, либо инструкцию **goto.** Вам необходимо просмотреть руководство по вашему компилятору, чтобы точно определить ограничения на использование встраиваемых функций.

Примеры

1. Любая функция может стать встраиваемой, включая функции — члены классов. Например, функция **divisible()** для ускорения ее выполнения сделана встраиваемой. (Функция возвращает истину, если ее первый аргумент без остатка может делиться на второй.)

```
// Демонстрация встраиваемой функции-члена
#include <iostream>
using namespace std;

class samp {
   int i, j;
public:
   samp(int a, int b);
   int divisible(); // встраивание происходит в этом определении
};

samp::samp(int a, int b)
{
   i = a;
   j = b;
}

/* Возврат 1, если i без остатка делится на j. Тело этой функции-члена
встраивается в программу
*/
inline int samp::divisible()
{
   return !(i%j);
}

int main()
{
   samp ob1(10, 2), ob2(10, 3);

   // это истина
   if(ob1.divisible()) cout << "10 делится на 2\n";
```

```
  // это ложь
  if(ob2.divisible()) cout << "10 делится на 3\n";

  return 0;
}
```

2. Допускается перегружать встраиваемую функцию. Например, эта программа перегружает **min()** тремя способами. В каждом случае функция также объявляется встраиваемой.

```
#include <iostream>
using namespace std;

// Перегрузка функции min() тремя способами

// int
inline int min(int a, int b)
{
  return a < b ? a : b;
}

// long
inline long min(long a, long b)
{
  return a < b ? a : b;
}

// double
inline double min(double a, double b)
{
  return a < b ? a : b;
}

int main()
{
  cout << min(-10, 10) << "\n";
  cout << min(-10.01, 100.002) << "\n";
  cout << min(-10L, 12L) << "\n";

  return 0;
}
```

2.7. Встраиваемые функции в объявлении класса

Если определение функции-члена достаточно короткое, его можно включить в объявление класса. Поступив таким образом, мы заставляем, если это возможно, функцию стать встраиваемой. Если функция задается внутри объявления класса, ключевое слово **inline** не требуется. (Однако использование его в такой ситуации не является ошибкой.) Например, как показано ниже, функция **divisible()** из предыдущего раздела может быть по умолчанию сделана встраиваемой:

```
#include <iostream>
using namespace std;
class samp {
        int i, j;
public:
        samp(int a, int b) ;
/* Функция divisible(), которая здесь определяется, по умолчанию становится встраиваемой.*/
int divisible() { return !'(i%j); } };
```

```
samp: :samp(int a, int b)
{
        i = a;
        j = b;
}

int main()
{
        samp obi (10, 2), ob2(10, 3);

        // это истина
        if(obl.divisible()) cout << "10 делится на 2\n";

        // это ложь
        if(ob2.divisible()) cout << "10 делится на 3\n";

        return 0;
}
```

Как видите, код функции **divisible()** находится внутри объявления класса **samp**. Отметьте, что никакого другого определения функции **divisible()** не нужно, это даже запрещено. Определение функции **divisible()** внутри класса samp автоматически заставляет ее стать встраиваемой функцией.

Если функция, заданная внутри объявления класса, не может стать встраиваемой функцией (поскольку были нарушены ограничения), она, обычно, преобразуется в обычную функцию.

Отметьте, как именно функция **divisible()** задается внутри класса **samp**, особенно само тело функции. Оно целиком расположено на одной строке. Такой формат для программ C++ является совершенно обычным, если:

функция объявляется внутри объявления класса. Такое объявление становится более компактным. Однако класс **samp** мог бы быть описан и так:

```
class samp {
        int i, j;
public:
        samp(int a/ int b) ;
/* Функция divisible()/ которая здесь определяется, по умолчанию становится встраиваемой.*/

        int divisible()
        {
        return !(i%j) ;
        }
};
```

Здесь в определении функции **divisible()** используется более или менее стандартный стиль отступов. С точки зрения компилятора, компактный и стандартный стили не отличаются. Однако в программах C++ при задании коротких функций внутри определения класса обычно используется компактный стиль.

На применение таких встраиваемых функций накладываются те же ограничения, что и на применение обычных встраиваемых функций.

Примеры

1. Вероятно наиболее традиционным использованием встраиваемых функций, определяемых внутри класса, является определение конструктора и деструктора. Например, класс **samp** может быть определен более эффективно:

```
#include <iostream>
using namespace std;

class samp {
   int i, j;
public:
   // встраиваемый конструктор
   samp(int a, int b) { i = a; j = b; }
   int divisible() { return !(i%j); }
};
```

2. Иногда короткие функции могут включаться в объявление класса даже тогда, когда преимущества встраивания мало что дают или вовсе не проявляются. Рассмотрим следующее объявление класса:

```
class myclass {
   int i;
public:
   myclass(int n) { i = n; }
   void show() { cout << i; }
};
```

Здесь функция **show()** по умолчанию становится встраиваемой. Однако, как вы, наверное, знаете, операции ввода/вывода, по сравнению с операциями процессор/память, являются настолько медленными, что какой бы то ни было эффект от устранения вызова функции практически отсутствует. Однако в программах на C++, как правило, можно встретить такие короткие функции внутри класса. Делается это просто для удобства, поскольку никакого вреда не приносит.