стр.	содержание исправления
101	Внизу страницы (добавить круглые скобки): **Примеры**: (a + b) ∗ c → (a + b) c ∗ → a b + c ∗. a ∗ b + c → (a b ∗) + c → a b ∗ c +. Формально подобное преобразование можно выполнять по *алгоритму Дейкстры*:

133	Над нижним заголовком убрать else:
	Представим обобщенную форму для моделирования цикла с предусловием, написанную на языке F#:
	~~else~~
	`let rec loopWhile condition bodyLoop =`
	`  if condition() then`
	`  bodyLoop()`
	`  loopWhile condition bodyLoop`
	`  else`
	`  ( );;`
	Циклы с постусловием

377	В 4-м абзаце сверху исправить в заголовке процедуры:
	Пары вхождений x, y и z
	`procedure swap (var x, y : T);`
	`  var z : T;`
	`  begin`
	`  z := x; x := y; y := z`
	`  end;`

379	В нижнем абзаце строки, выделенные синим, подвинуть вправо на одну позицию по образцу:
	`Program U;`
	`  Procedure X;`
	`  Procedure Y;`
	`  …`
	`  end Y;`
	`  end X;`
	`  Procedure A;`
	`  Procedure B;`
	`  …`
	`  end B;`
	`  end A;`
	…

382	Исправить (добавить) отступы во фрагменте по образцу → f (3) → f (2) → f (1) → f (0) ← f (0) = 1 ← f (1) = 1 ← f (2) = 2 ← f (3) = 6
392	В нижнем абзаце строку, выделенную синим, подвинуть влево на одну позицию по образцу: program M; procedure P; var x, y, z; procedure Q; procedure R; begin (* R *) … z := P; … end R; begin (* Q *) … y := R; … end Q; begin (* P *) … x := Q; … end P; Begin (* M *) … P; … End M.

Удаление и добавление точки с запятой (по образцу):

```
# include <iostream.h> //** подключение станд. библиотеки ввода-вывода
class Stack  {
      private:    //**скрытые элементы
          int *stack_ptr;
          int   max_len;
          int   tos;
       public:     //** эти элементы видимы клиенту
          Stack ( )   { //** конструктор
                    stack_ptr = new int [500];
                    max_len = 499;
                    tos = -1
          }
          ~ Stack ( )  {delete [ ]  stack_ptr;};  //** деструктор
          void push (int number)  {
                    if (tos  = =  max_len)
                       cout << “Error in push — stack is full \n”;
                    else stack_ptr [++tos] = number;
          }
          void pop ( )  {
                    if  (tos  = =  -1)
                       cout << “Error in pop — stack is empty \n”;
                    else  tos --;
          }
          int top ( )  {return (stack_ptr [tos]);}
          int empty ( )  {return (tos  = =  -1);}
};
```

Удаление и добавление точки с запятой (по образцу):

```
class Stack  {
      private:    //**скрытые элементы
          T *stack_ptr;
          int   max_len;
          int   tos;
       public:     //** эти элементы видимы клиенту
          Stack ( )   { //** конструктор для 500 элементов в стеке
                    stack_ptr = new T [500];
                    max_len = 499;
                    tos = -1
          }
          Stack (int size) { //** конструктор для переменного количества элементов в стеке
                    stack_ptr = new T [size];
                    max_len = size - 1;
                    tos = -1;
          }

          ~ Stack ( )  {delete [ ]  stack_ptr;};  //** деструктор
          void push (T number)  {
                    if (tos  = =  max_len)
                       cout << “Error in push — stack is full \n”;
                    else stack_ptr [++tos] = number;
          }
          void pop ( )  {
                    if  (tos  = =  -1)
                       cout << “Error in pop — stack is empty \n”;
                    else  tos --;
          }
          int top ( )  {return (stack_ptr [tos]);}
          int empty ( )  {return (tos  = =  -1);}

};
```

455

Удаление точки с запятой (по образцу):

Интерфейсы и абстрактные классы

Некоторые объектно-ориентированные языки, такие как Java, Ada, поддерживают концепцию, называемую интерфейсом. *Интерфейс* определяет протокол определенного поведения, но не обеспечивает его реализацию. Представим пример интерфейса, описывающего объекты, которые можно «*читать из*» и «*записывать в*» поток ввода-вывода:

```
public interface Transation {
    void writeOut (Stream s);
    void readFrom (Stream s);

};
```

Подобно классу, интерфейс определяет новый тип. Это значит, что переменные можно объявлять на основе имени интерфейса:

```
Transation buffer;
```

Класс может объявить, что он реализует протокол, определенный интерфейсом. Экземпляры класса могут назначаться переменным интерфейсного типа:

```
public class Memorization implements Transation {
    void writeOut (Stream s) {
        // ...

    };
    void readFrom (Stream s) {
        // ...

    };

};
buffer = new Memorization ();
```

465

Удаление точки с запятой (по образцу) в середине страницы:

Псевдопеременную непременно нужно указывать, если метод хочет передать себя как параметр в другую функцию, что иллюстрирует следующий фрагмент на языке Java:

```
class ExitButton extends Button implements Action {
    public ExitButton () {
        ...
        // установить себя в качестве обработчика событий кнопки
        addAction(this);
    }
    ...

};
```

Добавление точки с запятой (по образцу):

Единичное наследование

Форма записи класса-наследника имеет вид:
<класс_наследник> : <уровень_доступа> <базов_класс>

{<объявления элементов-данных, элементов-функций>};

При уровне доступа public сохраняется уровень видимости элементов базового класса. При уровне доступа private все элементы производного класса считаются приватными, то есть невидимыми для клиентов. Кроме того, приватные производные классы закрывают доступ ко всем элементам классов-предков со стороны всех классов-потомков.

Положим, что существует родительский класс:

```
class Parent_class  {
  private:
      int a;
      float x;
  protected: // разрешается доступ потомкам и друзьям
      int b;
      float y;
  public:
      int c;
      float z;
};
```

Будем считать, что у этого класса есть публичный и приватный наследники:

```
class Child_1: public Parent_class    {…};
class Child_2: private Parent_class  {…};
```

При приватном наследовании в экземплярах класса-наследника нет элементов, которые видят клиенты. Для восстановления видимости элемента применяется его повторное экспортирование. Оно отменяет скрытность элемента. Например:

```
class Child_3: private Parent_class {
      Parent_class ::  c;
      . . .
};
```

Удаление точки с запятой (по образцу):

Теперь в экземплярах Child_3 доступен элемент c (как если бы наследование было публичным). Двойное двоеточие — это операция разрешения области видимости. Она задает класс, где определен элемент.

Рассмотрим класс:

```
class Linked_list {
  private:
     class Element {
        public:
           Element *link;
           int content;
     };
     Element *head;
  public:

     Linked_list ( ) {head = 0};
     void insert_at_head (int);
     void insert_at_tail (int);
     int delete_at_head ( );
     int empty ( );
};
```

Вложенный в него класс Element определяет элемент связанного списка, включающий два компонента — поле целого типа и указатель на элемент. Этот класс размещен в приватной секции, что скрывает его от других классов. Элементы-данные в Element публичны, то есть видимы в объемлющем классе Linked_list. Если бы они были приватными, то для обеспечения видимости элементов-данных пришлось бы объявить класс Element другом класса Linked_list. Заметим, что у вложенных классов нет никаких специальных прав доступа к элементам объемлющего класса. Только статические элементы-данные объемлющего класса видимы методам вложенного класса.

Объемлющий класс Linked_list имеет один элемент-данные, это указатель на заголовок списка. Он содержит конструктор, обнуляющий значение указателя.

Четыре элемента-функции обеспечивают вставку элементов в список (с головы или хвоста), удаление элементов (с головы), проверку пустоты списка.

Определим классы стек и очередь, которые станут публичными потомками-наследниками класса связанный список:

```
class Stack : public Linked_list {
  public:

     Stack ( ) { };
     void push (int value) {
        Linked_list :: insert_at_head (int value);

     };
     int pop ( ) {
        return Linked_list :: delete_at_head ( );

     };
};
class Queue : public Linked_list {
  public:

     queue ( ) { };
     void enqueue (int value) {
        Linked_list :: insert_at_tail (int value);

     };
     int dequeue ( ) {
```

Удаление точки с запятой (по образцу):

```
        return Linked_list :: delete_at_head ( );
    };
};
```

Заметим, что объекты как подкласса Stack, так и подкласса Queue имеют доступ к функции empty ( ) из родительского класса Linked_list (так как это публичное наследование). Новые методы подклассов push(), pop(), enqueue(), dequeue() на самом деле лишь вызывают соответствующие методы родительского класса, которые и выполняют необходимую работу.

Оба подкласса содержат по конструктору (который ничего не делает). При создании объекта неявно вызывается конструктор подкласса. Далее вызывается конструктор родительского класса (в нашем примере именно он и выполняет необходимую инициализацию).

Однако здесь возникает серьезная проблема. Поскольку наследование публичное, объект класса Stack имеет доступ к операции insert_at_tail ( ) родительского класса, а это разрушает целостность стека. Аналогично объекту класса Queue доступна операция insert_at_head ( ). Эти нежелательные вызовы вполне возможны, поскольку и класс Stack, и класс Queue являются подтипами типа Linked_list, а значит их экземпляры (в соответствии с принципом подстановки) могут использоваться вместо экземпляров Linked_list.

Если задуматься, то следует сказать правду: функциональное сходство стека и очереди со связанным списком весьма сомнительного свойства и не выдерживает теста «*является*», поскольку эти дети слабо похожи на своего родителя. Следовательно, нужно отказаться от родства и применить «запретное» наследование для конструирования, которое в языке C++ поддерживается механизмом приватного наследования. Заметим, что приватным подклассам придется скопировать метод empty(), поскольку он будет скрыт от их экземпляров. В итоге получим следующие подклассы:

```
class PrivStack : private Linked_list {
  public:
    PrivStack ( ) { };
    void push (int value) {
        Linked_list :: insert_at_head (int value);
    };
    int pop ( ) {
        return single_linked_list :: delete_at_head ( );
    };
    Linked_list :: empty ( );
};
class PrivQueue : private Linked_list {
  public:
    PrivQueue ( ) { };
    void enqueue (int value) {
        Linked_list :: insert_at_tail (int value);
    };
    int dequeue ( ) {
        return Linked_list :: delete_at_head ( );
    };
    Linked_list :: empty ( );
};
```

Обратим внимание, что потребовался повторный экспорт операции empty ( ), поскольку она стала скрытой.

483	Второй абзац сверху. Добавление точки с запятой (по образцу):
	class Форма {
	public:
	virtual void display() = 0; // чисто виртуальная функция
	. . .
	};
	class Круг : public Форма {
	public:
	virtual void display() {…}
	. . .
	};
	class Прямоугольник : public Форма {
	public:
	virtual void display() {…}
	. . .
	};
	class Квадрат : public Прямоугольник {
	public:
	virtual void display() {…}
	. . .
	};

510	Вверху (добавить закрывающую фигурную скобку):
	public void operation1(<parameters>) {
	Проверить авторизацию // Проверка безопасности
	Блокировка для обеспечения сохранности потока // Контроль распараллеливания
	Начало транзакции // Управление транзакцией
	Регистрация начала операции // Трассировка
	Выполнение самой операции
	Регистрация завершения операции // Трассировка
	Завершение или откат транзакции // Управление транзакцией
	Разблокировка для обеспечения сохранности потока // Контроль распараллеливания
	}
	... Другие операции, реализующие множество понятий
	}

511	добавить закрывающую фигурную скобку:
	public class SomeBusinessLogic {
	Основные элементы данных
	public void operation1(<parameters>) {
	Выполнение самой операции
	}
	... Другие операции
	}

512	добавить открывающую и закрывающую фигурные скобки, а также три точки:
	Лучшим решением этой проблемы является применение аспекта:
	`aspect проверкаПодлинности {`
	`before: call (public void обновить*(..)) // это срез`
	`{`
	`// этот совет вплетается в процесс выполнения`
	`int попытка = 0;`
	`string userPassword = Password.Get(попытка);`
	`while (попытка < 3 && userPassword != thisUser.password()){`
	`// разрешаются 3 попытки ввода пароля`
	`попытка = попытка + 1;`
	`userPassword = Password.Get(попытка);`
	`}`
	`if (userPassword != thisUser.password()) then`
	`// при неправильном пароле выход из системы`
	`System.Logout(thisUser.uid);`
	`public void включить() {`
	`...`
	`}`
	`}`
	`} // проверкаПодлинности`

527	Второй абзац сверху (поставить точку в конце абзаца): Правило, определяющее, какой обработчик должен обрабатывать конкретное исключение, формулируется в терминах *динамической цепи активаций* подпрограмм, которая ведет к подпрограмме, породившей исключение. Если исключение X порождено в подпрограмме A, то оно обрабатывается обработчиком, определенным в этой подпрограмме A (если таковой имеется). Если обработчика нет, то A завершает свое выполнение. Если A была вызвана из подпрограммы B, то исключение распространяется на B. Если в B нет обработчика для исключения X, то B также завершается, а исключение распространяется на подпрограмму C, вызвавшую B, и так далее. Если ни в одной подпрограмме, а также в главной программе не найдется обработчика для этого исключения, то вся программа останавливается и вызывается стандартный обработчик, определенный в данном языке программирования.