Тест с отчетом 5

1) Какие методы компилятор может автоматически сгенерировать для данного класса?

class A { };

(**1**) конструктор по умолчанию,
конструктор копирования,
оператор operator=(const A&),
деструктор

(**2**) конструктор копирования
оператор operator=(const A&)
деструктор

(**3**) конструктор по умолчанию
конструктор копирования
деструктор

(**4**) конструктор по умолчанию
деструктор

2) Чему будет равно значение переменной X после выполнения фрагмента кода:

```
int arr[] = { 1 , 2 };
 int X = (arr[1] - arr[0]) [arr];
```

Варианты: (1) 0
(2) ошибка на этапе компиляции
(3) 2
(4) 1

arr[1] == 1[arr]

Численные алгоритмы

Численные алгоритмы выполняют разнообразную обработку числовых элементов.

Название	*Описание*
accumulate()	Объединяет все значения элементов (вычисляет сумму, произведение и т. д.)
inner_product()	Объединяет все элементы двух интервалов
adjacent_difference()	Объединяет каждый элемент с его предшественником
partial_sum()	Объединяет каждый элемент со всеми предшественниками

Алгоритм **accumulate()** по умолчанию вычисляет сумму элементов. Если элементами являются строки, то вычисляется их конкатенация. А если переключиться с оператора **+** на оператор *, алгоритм вычислит про изведение элементов.

Алгоритмы **accumulate()** и **inner_product()** вычисляют и возвращают сводное значение без модификации интервалов. Другие алгоритмы записывают свои результаты в приемный интервал, количество элементов в котором соответствует количеству элементов в исходном интервале.

Алгоритмы – пример (1)

// Содержание вспомогательного файла **algo.h**

```
#ifndef _ALGO_H
#define _ALGO_H

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <functional>
#include <numeric>
```

Алгоритмы – пример (2)

```
/* Prn_Elems()
*  вывод необязательной строки C, за которой выводятся
*  - все злементы коллекции col, разделенные пробелами.
*/

template <class T>
inline void Prn_Elems (const T& col, const char* optcstr="")
{
  typename T::const_iterator pos;
  std::cout << optcstr;
  for (pos=col.begin(); pos!=col.end(); ++pos) {
    std::cout << *pos << " ";
  }
  std::cout <<std::endl;
}
```

Алгоритмы – пример (3)

```
/* Insert_Elems (collection, first. last)
* - заполнение коллекции значениями от first до last
* - ВНИМАНИЕ: интервал НЕ ЯВЛЯЕТСЯ полуоткрытым
*/

template <class T>
inline void Insert_Elems (T& col, int first, int last)
{
  for (int i=first; i<=last; ++i) {
    col.insert (col.end( ), i );
  }

}
#endif   // _ALGO_H – конец заголовочного файла algo.h
```

Алгоритмы – пример (4)

```
#include "algo.h"
using namespace std;
void main()
{     vector<int> col;
  Insert_Elems(col,1,9);
  Prn_Elems(col);
  cout << "sum: "                                  // Вычисление суммы элементов
  << accumulate (col.begin(), col.end(),    // Интервал
              0) << endl;                            // Начальное значение
  cout << "sum: "                          // Вычисление суммы элементов с вычетом 100
  << accumulate (col.begin(), col.end(),  // Интервал
              -100) << endl;                      // Начальное значение

  cout << "product: "                             // Вычисление произведения элементов
  << accumulate (col.begin(), col.end(),  // Интервал
              1,                                      // Начальное значение
              multiplies<int>()) << endl;     // Операция
}
```

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
```

Распределители памяти

В STL используются специальные объекты, занимающиеся выделением и освобождением памяти. Такие объекты называются распределителями (allocators). Распределитель представляет собой абстракцию, которая преобразует потребность в памяти в физическую операцию ее выделения. Параллельное использование разных объектов-распределителей позволяет задействовать в программе несколько разных моделей памяти.

template < class Type**> class allocator**

Здесь Type – тип хранящихся в контейнере данных**.**

Конструкторы:

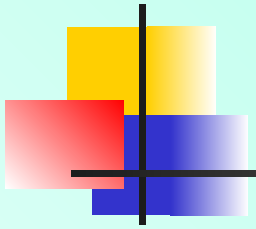
allocator( ); – конструктор по умолчанию

allocator (**const allocator**<Type>& _Right);

– конструктор копирования

template <**class** Other > **allocator**(**const allocator**<Other>& _Right);

– обобщенный конструктор копирования



Распределители памяти

Этот распределитель используется по умолчанию во всех тех ситуациях, когда распределитель может передаваться в качестве аргумента. Он использует стандартные средства выделения и освобождения памяти, то есть операторы **new и delete,** но в спецификации ничего не говорится о том, когда и как эти операторы должны вызываться. Таким образом, конкретная реализация распределителя по умолчанию может, например, организовать внутреннее кэширование выделяемой памяти.

В большинстве программ используется распределитель по умолчанию. Иногда библиотеки предоставляют специализированные распределители, которые просто передаются в виде аргументов. Необходимость в самостоятельном программировании распределителей возникает очень редко, на практике обычно достаточно распределителя по умолчанию.

Распределители памяти – члены класса

Основные операции распределителей:

Операция	*Описание*
allocate(num)	Выделение памяти для **num** элементов
construct(p)	Инициализация элемента, на который ссылается указатель **p**
destroy(p)	Уничтожение элемента, на который ссылается указатель **p**
deallocate(p, num)	Освобождение памяти для **num** элементов, на которую ссылается указатель **p**

Основные переопределяемые типы данных:

const_pointer	Тип, обеспечивающий постоянный указатель
const_reference	Тип, обеспечивающий постоянную ссылку
difference_type	Тип знакового целого, который представляет собой разницу между значениями пары указателей на объекты, управляемые распределителем
pointer	Тип, обеспечивающий указатель
reference	Тип, обеспечивающий ссылку
size_type	Тип беззнакового целого, связанного с размерами управляемых объектов
value_type	Тип данных, обеспечиваемых распределителем

Инициализирующий итератор

Класс **raw_storage_iterator** предназначен для перебора неинициализированной памяти и ее инициализации.

Итератор **raw_storage_iterator** может использовать любые алгоритмы для инициализации памяти значениями, полученными в результате выполнения алгоритма.

Например, следующая команда инициализирует память, на которую ссылается указатель **elems**, значениями из интервала [x.begin(), x.end());

```
copy (x.begin(), x.end(),                  // Источник
raw_storage_iterator<T*,T>(elems));        // Приемник
```

В первом аргументе шаблона (в данном случае T*) должен передаваться итератор вывода для типа элементов. Второй аргумент шаблона (в данном случае T) определяет тип элементов.

«Умные указатели» – тип auto_ptr

Стандартная библиотека C++ предоставляет класс **auto_ptr** как своего рода «умный указатель», помогающий предотвратить утечки ресурсов при исключениях.

*Знакомство с классом **auto_ptr:*** Функции часто работают по следующей схеме.

1. Получение ресурсов.
2. Выполнение операций.
3. Освобождение полученных ресурсов.

Если полученные ресурсы связаны с локальными объектами, они автоматически освобождаются при выходе из функции в результате вызова деструкторов локальных объектов. Но если ресурсы не связаны с объектом, они должны освобождаться явно. Как правило, такие операции с ресурсами выполняются с применением указателей.

Характерный пример работы с ресурсами через указатели - создание и уничтожение объектов операторами **new** и **delete**:

```
void f()
{   classA* ptr = new ClassA; // Создание объекта
                    // Выполнение операций
    delete ptr;               // Уничтожение объекта
}
```

«Умные указатели» – тип auto_ptr

При использовании подобных схем нередко возникают проблемы. Первая и наиболее очевидная проблема заключается в том, что программист может забыть об удалении объекта (особенно если внутри функции присутствуют команды **return**).

Но существует и другая, менее очевидная опасность: во время работы функции может произойти исключение, что приведет к немедленному выходу из функции без выполнения оператора **delete**, находящегося в конце тела функции. В результате возникает утечка памяти или, в общем случае, - ресурсов. Для ее предотвращения обычно приходится перехватывать все возможные исключения. Пример:

```
void f()  {
    ClassA* ptr new ClassA;   // Создание объекта
    try {   .  .  .  }          // Работа с объектом
    catch ( ... ) {                  // Для произвольного исключения:
        delete ptr:                  // - освободить ресурс
        throw:                  // - перезапустить исключение
    }
    delete ptr:                  // Нормальное освобождение ресурса
}
```

«Умные указатели» – тип auto_ptr

Освобождение объекта при возникновении исключений приводит к усложнению программы и появлению лишнего кода. Если по этой схеме обрабатывается не один, а два объекта или имеется несколько секций **catch**, программа становится еще запутаннее. В подобных решениях - сложных и чреватых ошибками – проявляется плохой стиль программирования.

В данной ситуации нужен умный указатель, освобождающий данные, на которые он ссылается, при уничтожении самого указателя. Более того, поскольку такой указатель является локальной переменной, он будет уничтожаться автоматически при выходе из функции независимо от причины выхода – нормального завершения или исключения. Класс **auto_ptr** проектировался именно для этих целей.

Указатель **auto_ptr** является владельцем объекта, на который он ссылается. В результате уничтожение указателя автоматически приводит к уничтожению объекта. Для работы **auto_ptr** необходимо, чтобы управляемый объект имел только одного владельца.

«Умные указатели» – тип auto_ptr – как использовать?

```
#include <memory>  // Заголовочный файл для auto_ptr
void f()
{
  std::auto ptr<ClassA> ptr(new ClassA);  // Создание и инициализация auto ptr
   . . .                                  // Работа с указателем
}
```

Команда **delete** и секция **catch** стали лишними. Интерфейс указателя **auto_ptr** почти не отличается от интерфейса обычного указателя; оператор * производит разыменование объекта, на который ссылается указатель, а оператор -> предоставляет доступ к членам класса или структуры. Математические операции с указателями (такие, как ++) для **auto_ptr** не определены. Впрочем, это скорее достоинство, нежели недостаток, потому что вычисления с указателями слишком часто приводят к неприятностям.

Важно: тип **auto_ptr** не позволяет инициализировать объект обычным указателем в конструкции присваивания. Инициализация **auto_ptr** должна производиться напрямую по значению:

```
std::auto ptr<ClassA> ptr1(new ClassA);       // ОК
std::auto ptr<ClassA> ptr2 = new ClassA;    // ОШИБКА
```

Передача прав владения в **auto_ptr**

Два и более экземпляра **auto_ptr** не должны одновременно быть владельцами одного объекта. К сожалению, в программе такая ситуация не исключена (например, если два экземпляра **auto_ptr** инициализируются одним и тем же объектом). Программист обязан позаботиться о том, чтобы этого не случилось.

Возникает вопрос: как работают копирующий конструктор и оператор присваивания типа **auto_ptr**? В обычном варианте эти операции копируют данные из одного объекта **auto_ptr** в другой, но в нашем случае это создает ситуацию, при которой один объект принадлежит сразу двум экземплярам **auto_ptr**. Проблема решается просто, но у этого решения есть одно важное следствие; копирующий конструктор и оператор присваивания передают «право владения» тем объектом, на который ссылается **auto_ptr**.

Рассмотрим следующий пример использования копирующего конструктора:

```
        // Инициализация auto ptr новым объектом
std::auto_ptr<ClassA> ptr1(new ClassA);
        // Копирование auto ptr
        // - право владения объектом передается от ptr1 к ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

Передача прав владения в **auto_ptr**

Смена владельца не является простым копированием. Во всех случаях передачи права владения предыдущий владелец (ptr1 в нашем примере) перестает им быть. В результате после передачи права владения предыдущий владелец содержит null-указатель. Подобное поведение серьезно противоречит общим принципам инициализации и присваивания в языках программирования.

Копирующий конструктор модифицирует объект, используемый для инициализации нового объекта, а оператор присваивания модифицирует правую часть операции присваивания. Программист должен сам следить за тем, чтобы программа не пыталась разыменовать экземпляр **auto_ptr**, переставший владеть объектом и содержащий null-указатель.

Новое значение, присваиваемое **auto_ptr**, также должно относиться к типу **auto_ptr**. Присваивание обычных указателей не допускается:

```
std: :auto ptr<ClassA> ptr;                 // Создание auto ptr

ptr = new ClassA;                           // ОШИБКА
ptr = std::auto_ptr<ClassA>(new ClassA);    // ОК. удаление старого объекта
                                            // и получение нового.
```

Boost libraries

Библиотека BOOST C++ - это собрание множества независимых библиотек, созданных независимыми разработчиками и тщательно проверенными на различных платформах. Можно считать, что BOOST C++ - это расширение стандартной библиотеки C++. Многие вещи, предлагавшиеся для нового стандарта C++ и отвергнутые комитетом по стандартизации, осели в BOOST. Использование компонентов BOOST C++ в своих разработках экономит значительные усилия - особенно если планируется портировать проект на различные платформы.

Полная (и актуальная) документация по всем аспектам библиотеки BOOST C++ находится на сайте **www.boost.org**. Там же Вы можете найти все ее исходные тексты - скачивайте архив, распаковывайте и используйте. Практическая полезность библиотеки BOOST гарантируется.

С составом библиотеки можно ознакомиться здесь:

http://www.solarix.ru/for_developers/cpp/boost/boost-library-list.shtml

Boost libraries – работа с памятью

pool - управление памятью в виде пулов

smart_ptr - "Умные" указатели.

scoped_ptr - не копируемые "автоматические" указатели. Подобно auto_ptr из стандартной библиотеки, но operator=() не реализован.

scoped_array - ни auto_ptr, ни scoped_ptr нельзя использовать для массивов. scoped_array - это адаптация scoped_ptr для массивов

shared_ptr - указатели с подсчетом ссылок

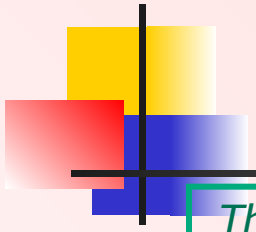
shared_array - реализация shared_ptr для массивов

weak_ptr - реализация "ослабленного" shared_ptr. Необходим для для разрыва циклических связей. Скажем в main() определен shared_ptr на объект A в котором явно или не явно присутствует shared_ptr на себя самого. Счетчик ссылок будет равен 2. При удалении первого shared_ptr счетчик ссылок на объект A будет уменьшен до 1 и объект никогда не удалится! Для решения этой проблемы и был создан week_ptr.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004)
Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.



Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принцип разнообразия : не доверяйте никаким претензиям на знание «единственно правильного пути».

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принцип разнообразия : не доверяйте никаким претензиям на знание «единственно правильного пути».

Принцип экономии*: время программиста дорого, пусть лучше работает ма*шина.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принцип разнообразия : не доверяйте никаким претензиям на знание «единственно правильного пути».

Принцип экономии: *время программиста дорого, пусть лучше работает ма*шина.

Принцип расширяемости *:* проектируйте с прицелом на будущее, потому что оно настанет раньше, чем вы ожидаете.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принцип разнообразия : не доверяйте никаким претензиям на знание «единственно правильного пути».

Принцип экономии: *время программиста дорого, пусть лучше работает ма*шина.

Принцип расширяемости *:* проектируйте с прицелом на будущее, потому что оно настанет раньше, чем вы ожидаете.

Принцип генерации *:* избегайте кодирования вручную; пишите программы, порождающие другие программы, когда это имеет смысл.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип ясности : ясность лучше изощренности.

Принцип композиции : проектируйте компоненты так, чтобы их можно было связать между собой.

Принцип разнообразия : не доверяйте никаким претензиям на знание «единственно правильного пути».

*Принцип экономии: время программиста дорого, пусть лучше работает ма*шина.

Принцип расширяемости : проектируйте с прицелом на будущее, потому что оно настанет раньше, чем вы ожидаете.

Принцип генерации : избегайте кодирования вручную; пишите программы, порождающие другие программы, когда это имеет смысл.

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принцип модульности: пишите простые части, объединяемые с помощью четко сформулированных интерфейсов.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принцип модульности: пишите простые части, объединяемые с помощью четко сформулированных интерфейсов.

Принцип наибольшего удивления : если уж приходится завершать программу с ошибкой, делайте это как можно более шумно и чем скорее, тем лучше.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принцип модульности: пишите простые части, объединяемые с помощью четко сформулированных интерфейсов.

Принцип наибольшего удивления : если уж приходится завершать программу с ошибкой, делайте это как можно более шумно и чем скорее, тем лучше.

Принцип оптимизации : пусть сначала заработает, оптимизировать будем потом.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принцип модульности: пишите простые части, объединяемые с помощью четко сформулированных интерфейсов.

Принцип наибольшего удивления : если уж приходится завершать программу с ошибкой, делайте это как можно более шумно и чем скорее, тем лучше.

Принцип оптимизации : пусть сначала заработает, оптимизировать будем потом.

Принцип скаредности : пишите большие компоненты только тогда, когда убедительно продемонстрировано, что ничего другого не остается.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип наименьшего удивления : проектируя интерфейс, стремитесь к интуитивной очевидности.

Принцип модульности: пишите простые части, объединяемые с помощью четко сформулированных интерфейсов.

Принцип наибольшего удивления : если уж приходится завершать программу с ошибкой, делайте это как можно более шумно и чем скорее, тем лучше.

Принцип оптимизации : пусть сначала заработает, оптимизировать будем потом.

Принцип скаредности : пишите большие компоненты только тогда, когда убедительно продемонстрировано, что ничего другого не остается.

Принцип надежности : надежность – дитя прозрачности и простоты.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип надежности : надежность – дитя прозрачности и простоты.

Принцип разделения : отделяйте политику от механизма, а интерфейс – от реализации.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип надежности : надежность – дитя прозрачности и простоты.

Принцип разделения : отделяйте политику от механизма, а интерфейс – от реализации.

Принцип простоты : проектируйте так, чтобы было просто пользоваться; раскрывайте сложность внутреннего устройства лишь тогда, когда без этого не обойтись.

Принципы программирования

The Art of UNIX Programming (AddisonWesley, 2004) Эрик Раймонд (Eric Raymond)

Принцип надежности : надежность – дитя прозрачности и простоты.

Принцип разделения : отделяйте политику от механизма, а интерфейс – от реализации.

Принцип простоты : проектируйте так, чтобы было просто пользоваться; раскрывайте сложность внутреннего устройства лишь тогда, когда без этого не обойтись.

Принцип прозрачности : проектируйте так, чтобы упростить изучение или отладку кода.

Принципы программирования

Уилсон М. Расширение библиотеки STL для C++. Наборы и итераторы: – М.: ДМК Пресс, СПб, БХВ-Петербург, 2008.

Понятность определяется тем, насколько просто разобраться в компоненте до такой степени, чтобы можно было им воспользоваться.

Понятность – это, прежде всего, мера интуитивной очевидности интерфейса компонента – его формы, непротиворечивости, ортогональности, соглашений об именовании, отношений между параметрами, имен методов, идиоматичности и т. д. Сюда же относится документация, наличие учебных руководств, примеров и всего, что помогает потенциальному пользователю прийти к пониманию.

Понятный интерфейс легко использовать правильно, и трудно – неправильно.

Прозрачность определяется тем, насколько просто разобраться в компоненте до такой степени, чтобы можно было его модифицировать.

Прозрачность в большей степени относится к организации кода: размещению файлов, расстановке скобок, именам локальных переменных, полезным комментариям и т.д., хотя в этом же ряду стоит документация, описывающая реализацию, если таковая существует.