Экзамен C++

Билет 1

В отличие от языка Java, обобщённые классы в C++ в строгом смысле отсутствуют. Вместо них в C++ используется развитый язык макроопределений, предназначенный для порождения кода во время компиляции программы. Ключевым элментом этого языка является понятие *шаблона*, который является рецептом для генерации кода класса или функции (метода).
Объявление шаблона начинается с ключевого слова `template`, за которым следует список формальных параметров шаблона:

```
template <формальные параметры>
определение класса или функции
```

Тело шаблона представляет собой определение класса или функции, тем самым позволяя объявлять *шаблон класса* и *шаблон функции.*

Формальные параметры шаблона – это идентификаторы, областью видимости которых является тело шаблона. Они могут обозначать типы, значения или, в свою очередь, другие шаблоны. При применении шаблона формальные параметры в его теле заменяются на конкретные типы, значения и шаблоны, и полученный код компилируется.
Типовые параметры шаблона обозначают типы: синтаксически имя типового параметра в списке формальных параметров шаблона предваряется ключевым словом typename (или class).
Нетиповые параметры шаблона представляют значения, а не типы. Их синтаксис похож на объявление переменных. Внутри тела шаблона имя такого параметра обозначает константу указанного типа.

Билет 2

Избежать многократного включения базового класса в производный класс позволяет виртуальное наследование.
Виртуальное наследование – это способ реализации наследования, гарантирующий, что базовый класс не будет включён ни в один из производных классов более чем в одном экземпляре.
Если класс B виртуально наследует классу A, то говорят, что A– виртуальный базовый класс для класса B.

Билет 3

Иерархия наследования – это ориентированный ациклический граф, множеством узлов которого является множество классов программы. При этом если класс Y является непосредственным базовым классом для класса X, то из узла X исходит дуга, входящая в Y.

Мы будем говорить, что класс B является **классом противоречия**, если существует такой класс A, что в иерархии наследования можно провести не менее двух непересекающихся путей из A в B.

Основной проблемой противоречивых иерархий является возможность многократного включения полей базового класса в производный класс.

Билет 4

C++ поддерживает множественное наследование. Если класс `Y` – производный от нескольких базовых классов `X1`, `X2`, ..., `Xn`, то объявление класса `Y` выглядит на C++ как

```
class Y: мод X1, мод X2, ..., мод Xn
{
    ...
};
```

Здесь «мод» – это `public`, `protected` или `private`.
При этом в конструкторе класса `Y` должны выполняться вызовы конструкторов всех базовых классов, кроме, возможно, тех из них, которые имеют конструкторы по умолчанию:

```
Y::Y(...):
    X1(...),
    X2(...),
    ...,
    Xn(...) {
    ...
}
```

43 / 122

При множественном наследовании в производном классе могут возникать два или более методов с совпадающими именами и сигнатурами. То же самое справедливо и для полей.

Для разрешения противоречий в именах наследуемых членов класса необходимо использовать квалифицированные имена.

Билет 5

Операция динамического приведения объекта obj к типу T проверяет, является ли тип T одним из типов объекта obj, и возвращает obj, если является. В противном случае операция либо возвращает нулевой указатель, либо порождает исключение.
В C++ операция динамического приведения типа может применяться к указателям и ссылкам на объекты и записывается как

```
dynamic_cast <T> (obj)
```

Так как нулевых ссылок не существует, то при неудачном приведении порождается исключение `std::bad_cast`.

Билет 6

Синтаксис одиночного наследование в C++ выглядит как

```
class ИмяПроизводногоКласса: public ИмяБазовогоКласса
{
    ...
};
```

Если конструктор базового класса не имеет параметров (является конструктором по умолчанию), его вызов добавляется компилятором C++ в конструктор производного класса автоматически.
В противном случае необходимо явно вызывать конструктор базового класса в конструкторе производного класса.
В языке C++ вызов конструктора базового класса X из конструктора производного класса Y выполняется аналогично вызовам конструкторов полей:

```
Y::Y(формальные_параметры_конструктора_Y)
    : X(фактические_параметры_конструктора_X)
{ ... }
```

В C++, как и в языке Java, для переопределения метода, унаследованного от базового класса, достаточно объявить и определить этот метод в производном классе.

Билет 7

Следует понимать, что передача объекта в качестве параметра при вызове метода автоматически влечёт создание копии объекта. Смысл копирования заключается в том, что изменение копии объекта внутри метода не приводит к изменению объекта-оригинала (семантика копирования).
Кроме этого, копирование объектов осуществляется при инициализации объявляемых переменных.

Для создания «правильных» копий объектов необходимо объявить конструктор копий, прототип которого выглядит как

```
имя_класса(const имя_класса &obj);
```

Этот конструктор будет автоматически вызываться всякий раз, когда необходимо создать копию объекта.

Проблема копирования объектов возникает также при присваивании. Поэтому, забегая вперёд, покажем на примере класса **IntArray**, как определить свою операцию присваивания:

```
class IntArray {
    ...
    IntArray& operator= (const IntArray &obj);
    ...
};

IntArray& IntArray::operator= (const IntArray &obj) {
    if (this != &obj) {
        int *new_a = new int [obj.n];
        std::copy(obj.a, obj.a + obj.n, new_a);
        delete [] a;
        n = obj.n;
        a = new_a;
    }
    return *this;
}
```


Билет 8

Если объект размещается в глобальной переменной или в статическом поле класса, то конструктор для него вызывается до передачи управления в функцию main, а деструктор – после завершения функции main.
Пример:

```
Demo d(100);

int main()
{
    cout << "main␣";
    return 0;
}
```

Вывод:

```
cons:100 main destr:100
```

Если объект класса `X` содержится в поле `x` объекта `Y`, то конструктор для этого поля вызывается из конструктора `Y` посредством следующей синтаксической конструкции:

```
Y::Y(формальные_параметры_конструктора_Y)
    : x(фактические_параметры_конструктора_X) {
    ...
}
```

Между прочим, любое поле объекта может быть инициализировано таким образом, даже если тип этого поля не является классом. Например, конструктор класса Demo может быть переписан как

```
Demo::Demo(int x): x(x) {
    cout << "cons:" << x << "␣";
}
```

Внутреннее состояние объекта может содержать указатели (ссылки) на другие объекты, созданные в его конструкторе. В языке без сборки мусора освобождение такого объекта может привести к утечке памяти.

Деструктор – это экземплярный метод, предназначенный для освобождения ресурсов, принадлежащих объекту, непосредственно перед уничтожением этого объекта.

В языке C++ деструкторы жизненно необходимы. Прототип деструктора имеет вид

```
~имя_класса();
```

Деструктор вызывается при уничтожении объекта. В частности, его автоматически вызывает оператор `delete`.

В отличие от языка Java, в C++ объекты могут размещаться в глобальных и локальных переменных, в параметрах методов, в элементах массивов и полях других объектов. Синтаксис объявления переменной, содержащей объект, выглядит как

```
имя_класса имя_переменной(параметры_конструктора);
```

Такие объявления, кстати, можно применять и для переменных, тип которых не является классом.
Примеры:

```
IntArray a(100);
Point p(10.5, 15.0);
int i(666);
const char *s("qwerty");
```

Объявление переменной, содержащей объект, вызывает экземплярный конструктор класса. Более того, при выходе из блока, где такая переменная объявлена, автоматически вызывается деструктор.

Билет 10

Рассмотрим, как в C++ принято создавать объекты классов. При этом мы на данном этапе изучения C++ ограничимся созданием объектов в динамической памяти.
Вообще, динамическое выделение памяти под значение любого типа выполняется в C++ операцией **new**, имеющей следующий синтаксис:

```
new имя_примитивного_типа
```

или

```
new имя_примитивного_типа(начальное_значение)
```

или

```
new имя_класса(фактические_параметры_конструктора)
```

Например,

```
int *x = new int;
float *y = new float(3.14);
Point *p = new Point(1.5, 2.3);
```

Операция **new** совмещает выделение памяти с инициализацией (с вызовом конструктора) и возвращает указатель на созданный и инициализированный объект.
В C++ вообще не принято использовать функцию **malloc**, потому что для создания массивов также используется специальная форма операции **new**:

```
new тип_элемента [размер]
```

Например,

```
int *a = new int [10];
Point **pa = new Point* [20];
```

Обратите внимание на то, что массив объектов можно выделять только в случае, если в классе определён конструктор по умолчанию. При этом этот конструктор будет вызван для каждого объекта в массиве.

```
Point *b = new Point [20]; // ошибка
```


Так как в C++ не предусмотрено автоматическое управление памятью (нет сборщика мусора), то все объекты, созданные операцией **new**, должны быть в какой-то момент явно освобождены.
Для этого существует оператор **delete**, имеющий следующий синтаксис:

```
delete указатель_на_объект;     // для объектов
delete [] указатель_на_массив; // для массивов
```

Например,

```
Point *p = new Point(10.0, 20.0);
delete p;
int *a = new int [20];
delete [] a;
```

Билет 11

В C++ объявление экземплярного конструктора, как и объявление обычного экземплярного метода, состоит из прототипа конструктора, записываемого в теле класса, и определения конструктора, размещённого в одном из cpp-файлов. В отличие от объявления метода, в объявлении конструктора имя конструктора совпадает с именем класса, и для конструктора не указывается тип возвращаемого значения.

```
class Point {
private:
    double x, y;
public:
    Point(double x, double y);
};

Point::Point(double x, double y) {
    this->x = x; this->y = y;
}
```

Билет 12

В языке C++ **экземплярные поля класса объявляются** также, как и поля структур в языке C. При этом объявления статических полей начинаются с модификатора static. По умолчанию поля недоступны извне класса, но доступны извне структуры или объединения. Например,

```
class Point {
public:
    int x, y; // Координаты точки
    static int count; // Общее количество точек
};
```

Особенностью языка C++ является то, что он расчитан на ту же схему компиляции, что и язык C. То есть программа на C++ состоит из набора cpp-файлов, каждый из которых компилируется в объектный файл. Чтобы связать эти объектные файлы, применяется линкер.
Так как класс в общем случае может использоваться в нескольких cpp-файлах, то в каждом файле должно быть его объявление. На практике объявление класса выносится в отдельный h-файл, который включается в нужные cpp-файлы.
Так как статическое поле – это фактически глобальная переменная, то она должна быть явно помещена в один из объектных файлов. Для этого в соответствующем cpp-файле должно быть дано её определение (имя статического поля выглядит как Имя_класса::Имя_поля).

```
int Point::count = 0;
```

Объявление метода на C++ состоит из прототипа и определения. Прототип метода помещается в тело класса, а определение – в нужный cpp-файл (тут та же история, что и со статическими полями). Прототипы статических методов объявляются с модификатором static. Пример (объявление класса с прототипом метода dist):

```
class Point {
public:
    double x, y;
    double dist();
};

double Point::dist() {
    return sqrt(x*x + y*y);
}
```

По умолчанию для вызова методов в C++ используется раннее связывание.
Чтобы включить позднее связывание, в объявлении прототипа метода надо указать модификатор `virtual`:

```
class Point
{
public:
    virtual double dist();
};
```

Прототип абстрактного метода заканчивается на «= 0»:

```
class Point
{
public:
    virtual double dist() = 0;
};
```

Класс, в котором есть абстрактный метод, автоматически становится абстрактным.

Билет 13

В языке C++ любая структура de facto является классом. Однако, классы принято объявлять с помощью конструкции

```
class Имя {
...
};
```

Для управления доступом к членам класса в C++ предусмотрены `public`-, `private`- и `protected`-секции внутри объявления класса:

```
class Имя
{
public:
    ...
private:
    ...
protected:
    ...
};
```

Секции могут быть перечислены внутри объявления класса в любом порядке и количестве. Члены класса, попадающие в `public`-секцию, видны отовсюду, члены из `private`-секции видны только из методов данного класса, а члены `protected`-секции – из методов данного класса и его классов-наследников.

Билет 14

Ссылка может быть возвращаемым значением функции. При этом надо следить, чтобы случайно не вернуть ссылку на локальную переменную.
Например,

```
char &ith(char *s, int i)
{
    return s[i];
}

int main()
{
    char s[] = "qwerty";
    ith(s, 2) = 'x';
    cout << s << endl;
    return 0;
}
```

Вывод:

```
qwxrty
```


Билет 15

Ссылка в C++ – это типизированный указатель, к которому неприменимы арифметические операции, и который не может быть нулевым. Кроме того, недопустимы ссылки на ссылки.
Для объявления ссылки используется префиксный декларатор «&»:

```
тип &имя_переменной;
```

При этом ссылки в глобальных и локальных переменных должны быть обязательно инициализированы при объявлении (забегая вперёд: ссылки в полях объектов должны быть инициализированы в конструкторе класса).
Более того, значение, полученное ссылкой при инициализации, в дальнейшем не может быть изменено.

Для того чтобы присвоить ссылке адрес некоторого значения в памяти, не нужно использовать операцию «&» для получения адреса объекта. Для доступа к значению, на которое указывает ссылка, не нужно использовать операцию разыменования «*».
Например,

```
int main()
{
    int x = 10;
    int &y = x;   // y указывает на x
    cout << y << "␣";
    y = 20;       // меняем значение x через y
    cout << x << endl;
    return 0;
}
```

Вывод:

```
10 20
```

Использование ссылок в качестве параметров функции позволяет имитировать var-параметры процедур и функций языка Pascal.
Например,

```
void swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    cout << x << "␣" << y << endl;
    return 0;
}
```

Вывод:

```
20 10
```