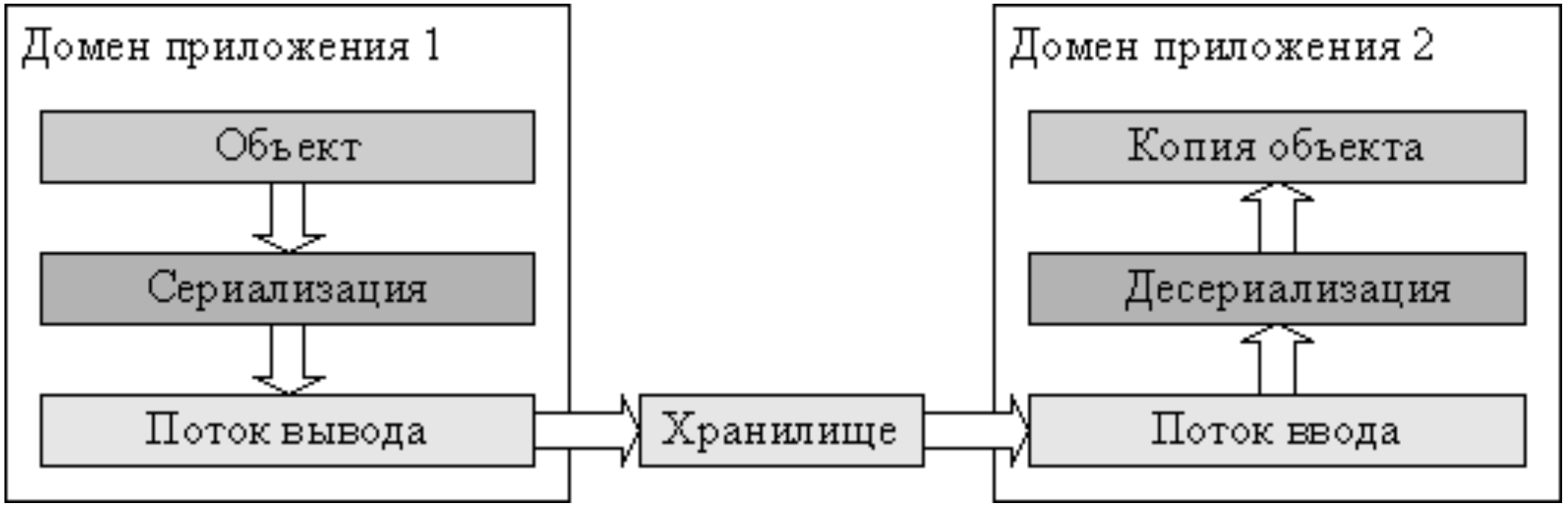
Базовые возможности C++

ЛЕКЦИЯ №3

7. сохраняемость

Сохраняемость - способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Как передать объект на другую машину?

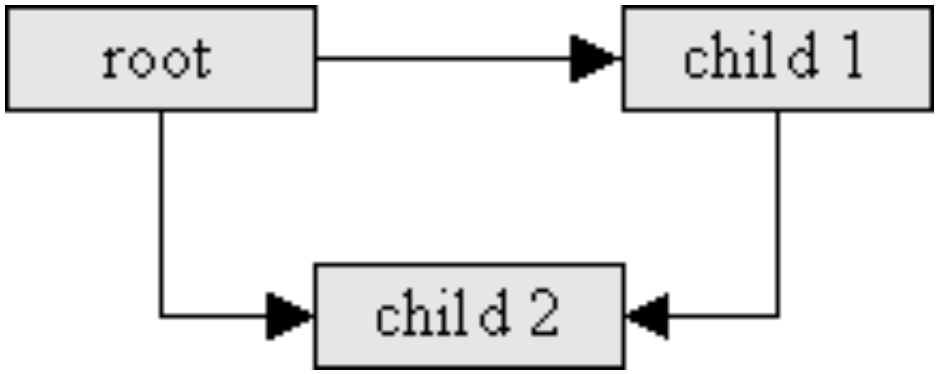


А что если объект сложный? граф объектов

Задача сериализации объекта, включающего только поля из элементарных типов значений и строк, не представляет принципиальных трудностей. Для такого объекта в ходе сериализации в поток записываются сами значения всех полей объекта. Однако в общем случае объект содержит ссылки на другие объекты, которые, в свою очередь, могут ссылаться друг на друга, образуя так называемый граф объектов (*object graph*).

```
class SampleClass
{
     SampleClass fieldA = nullptr;
     SampleClass fieldB = nullptr;
}

...
    SampleClass root = new SampleClass();
    SampleClass child1 = new SampleClass();
    SampleClass child2 = new SampleClass();
    root.fieldA = child1;
    root.fieldB = child2;
    child1.fieldA = child2;
```



Просто пронумеруем объекты почти как ссылки в C++

1. **ObjectID ?**
   В ходе сериализации объектам должны быть поставлены в соответствие некоторые идентификаторы, и в хранилище отдельно сохраняется список объектов, отмеченный идентификаторами, а при сериализации вместо ссылок записываются идентификаторы ссылаемых объектов.
2. **Виртуальный адрес!**
   В программе роль идентификатора объекта выполняет его адрес, но вместо него обычно удобнее назначить некоторые идентификаторы в процедуре сериализации для более легкого чтения человеком полученного образа.
3. **Hash-map …**
   В ходе сериализации нужно ввести список адресов уже записанных объектов, как для ведения списка идентификаторов, так и для обнаружения возможных циклов при обходе графа методом в глубину.

Пример

Example14_Serialize

```
1. class A {
2. public:
3.     A(const char* value) : next(nullptr), name(value) {};
4.     A(std::ifstream &is) {
5.         bool is_next;
6.         is >> name;
7.         is >> is_next;
8.         if (is_next) next = new A(is);}
9.     void Serialize(std::ofstream &os) {
10.         os << name << std::endl;
11.         if (next != nullptr) {
12.             os << true;
13.             next->Serialize(os);} else os << false;}
14. void SetNext(A* value) {
15.         next = value;     }
16.    virtual ~A() { delete next;     }
17. private:
18.     std::string name;
19.     A* next;
20. };
```

Полезные приемы по работе с классами

Временные объекты

Вре́менные объекты — в C++ объекты, которые компилятор создаёт автоматически по ходу вычисления выражений. Такие объекты не имеют имени и уничтожаются сразу же, как только в них исчезает потребность.

Пример:

```
string r = string("1") + "2" + "3";
```

```
string r, tmp1, tmp2, tmp3;
tmp1.ctor("1");
tmp2.ctor();
tmp2 = tmp1 + "2";
tmp3.ctor();
tmp3 = tmp2 + "3";
r.ctor(tmp3);
tmp3.dtor();
tmp2.dtor();
tmp1.dtor();
```

Модификаторы функций

Example15_Virtual

Ключевое слово virtual опционально и поэтому немного затрудняло чтение кода, заставляя вечно возвращаться в вершину иерархии наследования, чтобы посмотреть объявлен ли виртуальным тот или иной метод.

Типовые ошибки: Изменение сигнатуры метода в наследнике.

Модификатор **override** позволяет указать компилятору, что мы хотим переопределить виртуальный метод. Если мы ошиблись в описании сигнатуры метода – то компилятор выдаст нам ошибку. Этот модификатор влияет только на проверки в момент компиляции.

Модификатор final

Example28_Final

Модификатор **final**, указывающий что производный класс не должен переопределять виртуальный метод.

Работает только с модификатором virtual. Т.е. Создавать «копию» функции в классе-наследнике с помощью этой техники запретить нельзя.

Применяется, в случае если нужно запретить дальнейшее переопределение метода в дальнейших наследниках наследника (очевидно, что в родительском классе такой модификатор ставить бессмысленно).

Модификатор const
Example29_Const

const, применительно к методу (например, void foo const {}) означает, что метод не может вызывать другие методы без модификатора const и не может менять значения атрибутов объекта.

```
class A{
int a;
void fooA2() {}
void fooA () const{
      fooA2(); //ошибка
        a=7;  // ошибка
   }
```

mutable
Example33_Mutable

Ключевое слово **mutable** позволяет специфицировать атрибуты класса, которые могут меняться из **const** методов.

Иногда необходимо иметь такие атрибуты для «служебных целей», например, подсчитывать число обращений к методу и т.д.

Const

Example23_Const

```
1. int a=100;//два обычных объекта типа int
2. int b=222;
3. int *const P2=&a; //Константный указатель
4. *P2=987; //Менять значение разрешено
5. //P2=&b; //Но изменять адрес не разрешается
6. const int *P1=&a; //Указатель на константу
7. //*P1=110; //Менять значение нельзя
8. P1=&b; //Но менять адрес разрешено
9. const int *const P3=&a;//Константный указатель на константу
10. //*P3=155; //Изменять нельзя ни значение
11. //P3=&b; //Ни адрес к которому такой указатель привязан
```

Constexpr
Example31_constexpr

Позволяет создавать выражения, вычисляемые на этапе компиляции (позволяет упростить код).

Ключевое слово constexpr, добавленное в C++11, перед функцией означает, что если значения параметров возможно посчитать на этапе компиляции, то возвращаемое значение также должно посчитаться на этапе компиляции. Если значение хотя бы одного параметра будет неизвестно на этапе компиляции, то функция будет запущена в runtime (а не будет выведена ошибка компиляции).

```
constexpr int sum (int a, int b){
    return a + b;
}
void func(){
    constexpr int c = sum (5, 12); // значение переменной будет посчитано на этапе компиляции
}
```

Inheritance / наследование

Example24_MultipleInheritance

```
class temporary { /* ... */ };

class secretary : public
employee { /* ... */ };

class tsec

: public temporary, public
secretary { /* ... */ };

class consultant

: public temporary, public
manager { /* ... */ };
```

В C++ в отличии от большинства других объектно-ориентированных языков есть множественное наследование!

Очень плохо, если у двух родителей класса есть общий родитель!

Ссылка на себя

Считается, что в каждой функции-члене класса X указатель this описан неявно как

```
X *const this;
```

```
class X {
int m;
public:
int readm() { return this->m; }
};
```

Ссылка на родителя

Example25_ReferenceToParent

```
class Parent {
public:
Parent(void);
~Parent(void);
void Foo(void);
};
class Child : public Parent {
public:
Child(void);
~Child(void);
void Foo(void);
};
```

```
void Parent::Foo(void)
{
std::cout << "Parent\n";
}

void Child::Foo(void)
{
Parent::Foo();
std::cout << "Child\n";
}
```

Виртуальные деструкторы
Example26_VirtualDestructor

C++ вызывает деструктор для текущего типа и для всех его родителей.

Однако, если мы работаем с указателями то можем попасть в неприятную ситуацию, когда вызываем оператор delete у указателя, предварительно приведя его к типу родителя. В этом случае, вообще говоря, у наследников деструктор вызван не будет (например, если у родителя нет деструктора).

Однако если объявить деструктор базового класса как virtual то будут вызваны деструкторы всех классов.

Всегда объявляй деструктор как virtual!

Последовательность вызова конструкторов и деструкторов

Конструкторы вызываются начиная от родителя к наследнику.

Деструкторы вызываются начиная от наследника к родителю.

```
class A {}
class B : A {}
class C: B{}
```

Конструкторы:

`A, B, C`

Деструкторы:

`~C, ~B,~A`

Явные и неявные конструкторы

Example20_explicit

В C++ существуют явные и неявные конструкторы; преобразования, определенные с помощью конструктора со спецификатором **explicit** могут использоваться только при явном преобразовании, в то время, как другие конструкторы могут использоваться также и в неявных преобразованиях.
Например:

```
// "обычный конструктор" определяет неявное преобразование
class A { S(int); };
A a1(1);          // ok
A a2 = 1;          // ok
void Foo(A);
Foo(1);             // ok (но это может привести к неприятным

class E { explicit E(int); };    // явный конструктор
E e1(1);          // ok
E e2 = 1;          // ошибка (хотя это обычно является сюрпризом)
void f(E);
f(1);              // ошибка (защищает от сюрпризов – например,
```