Подготовительная программа по программированию на C/C++

Занятие №4

Валентина Глазкова

Специальные вопросы инкапсуляции и наследования

- Неустойчивые и изменчивые члены классов
- Классы-объединения
- Праводопустимые выражения в C++11
- Инициализация, копирование, преобразование и уничтожение объектов

Неустойчивые объекты (1 / 2)

Неустойчивые, или **асинхронно изменяемые** (volatile), объекты могут изменяться незаметно для компилятора.

Пример: переменная, обновляемая значением системных часов (например, в обработчике события, сигнала).

Целью определения объекта как неустойчивого является информирование компилятора о том, что тот не может определить, каким образом может изменяться значение данного объекта.

Спецификатор volatile сообщает компилятору о том, что при работе с данным объектом **не следует выполнять оптимизацию кода**.

Неустойчивые объекты (2 / 2)

Допустимы неустойчивые объекты скалярных и составных типов, указатели на неустойчивые объекты, неустойчивые массивы:

- в неустойчивом массиве неустойчивым считается каждый элемент;
- в неустойчивом экземпляре (объекте) класса неустойчивым считается каждый член данных. **Объекты классов неустойчивы целиком.**

Например:

```
volatile unsigned long timer;// неустойчивый скаляр
volatile short ports[size]; // неустойчивый массив
// указатель на неустойчивый объект класса
volatile Timer *tmr;
```

Для преобразования неустойчивого типа в устойчивый используется const_cast.

Неустойчивые методы класса

Методы класса могут объявляться как **неустойчивые**.

Неустойчивые методы класса являются **единственной категорией методов**, которые (наряду с конструкторами и деструкторами) могут вызываться применительно к неустойчивым объектам класса (объектам, значение которых изменяется способом, не обнаруживаемым компилятором).

Например:

```
class Timer
{           /* … */
            void getCurTime() volatile;
            /* … */
};
```

Изменчивые члены данных

Атрибуты класса, допускающие модификацию при любом использовании объекта, должны определяться как **изменчивые**.

Изменчивые атрибуты **не являются константными**, даже будучи членами константного объекта, что позволяет модифицировать их значения, в том числе константными методами.

Например:

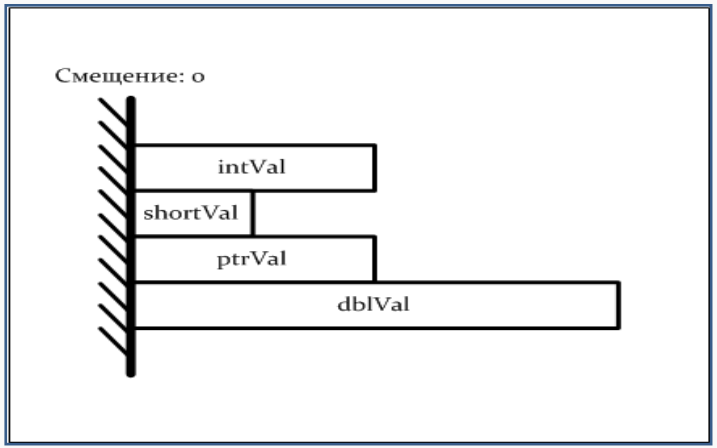
```
class Book
{       /* ... */
        mutable int _currentPage;
        /* ... */
        void locate(const int &value) const
        { /* ... */ _currentPage = value; /* ... */ }
};
```

Классы-объединения

Объединение в C++ — специальная категория класса, в котором члены данных физически располагаются, начиная с одного машинного адреса.

Размер класса-объединения определяется размерами его атрибутов и в целом **равен максимальному** среди них.

В любой момент времени значение может быть присвоено **только одному атрибуту**.



Объединение может включать как члены-данные, так и члены-функции

Классы-объединения

Все члены объединения **открыты** по умолчанию.

Объединение **не может наследовать** какие-либо другие классы.

Объединение **не может** использоваться **в качестве базового класса**.

Объединение **не может** иметь **виртуальные члены-функции**.

Никакие **статические** переменные **не могут** быть членами объединения.

Никакой объект **не может** быть членом объединения, если этот объект имеет **конструктор или деструктор**.

Классы-объединения: пример

```
union DataChunk
{
        int     intVal;
        short   shortVal;
        char*   ptrVal;
        double  dblVal;
};
DataChunk dc;
DataChunk *pdc = &dc;
dc.intVal = 0xFFAA;
pdc->shortVal = 077;
```

Безымянные объединения

```
// имя типа объединения
// может быть опущено
class IOPort
{       /* … */
        union
        {
                int         intVal;
                short       shortVal;
                char*       ptrVal;
                double dblVal;
        } _value;
} port;
port._value.intVal = 0xABCD;
```

Анонимные объединения

Объединение без имени, за которым не следует определение объекта, называется **анонимным**. К его членам можно обращаться непосредственно из той области видимости, в которой оно определено.

Анонимные объединения позволяют устранить один уровень доступа, у них не может быть каких бы то ни было методов.

Анонимные объединения: пример

```
class IOPort
{
        /* … */
        union {
                int     intVal;
                short   shortVal;
                char*   ptrVal;
                double dblVal;
        };
} port;
port.ptrVal = NULL;
```

Битовые поля в определении классов

Для хранения заданного числа двоичных разрядов может быть определен член класса, называемый **битовым полем**. Его тип должен быть знаковым или беззнаковым целым.

Определенные друг за другом битовые поля по возможности «упаковываются» компилятором. Например:

```
class IOPort
{
        unsigned int _ioMode  : 2;
        unsigned int _enabled : 1;
        /* … */
};
```

К битовому полю запрещено применять оператор взятия адреса.

Битовые поля не могут быть статическими членами класса.

Семантика переноса (C++11)

Введение в C++11 семантики переноса (англ. move semantics) обогащает язык возможностями более **тонкого и эффективного управления памятью данных**, устраняющего копирование объектов там, где оно нецелесообразно.

Технически семантика переноса реализуется при помощи **ссылок на праводопустимые выражения** (англ. rvalue reference) и **конструкторов переноса**.

Объявление *rvalue*-ссылок : type &&.

Семантика переноса (C++11)

В C++11 правила разрешения перегрузки позволяют использовать разные перегруженные функции для неконстантных временных объектов, обозначаемых посредством rvalues, и для всех остальных объектов

Конструкторы переноса не создают точную копию своего параметра, а перенастраивают параметр так, чтобы права владения соответствующей областью памяти были переданы вновь создаваемому объекту («заимствованы» последним).

Аналогично работают **операции присваивания с переносом**.

Конструктор переноса: пример (1 / 2)

```
class Alpha {
public:
            Alpha();
            // к о н с т р у к т о р  к о п и р о в а н и я
            Alpha(const Alpha &a);
            // к о н с т р у к т о р  п е р е н о с а
            Alpha(Alpha &&a);
            ~Alpha();
private:
            size_t sz;
            double *d;
};

Alpha::Alpha() : sz(0), d(0) { }

Alpha::~Alpha() {
            delete [] d;
}
```

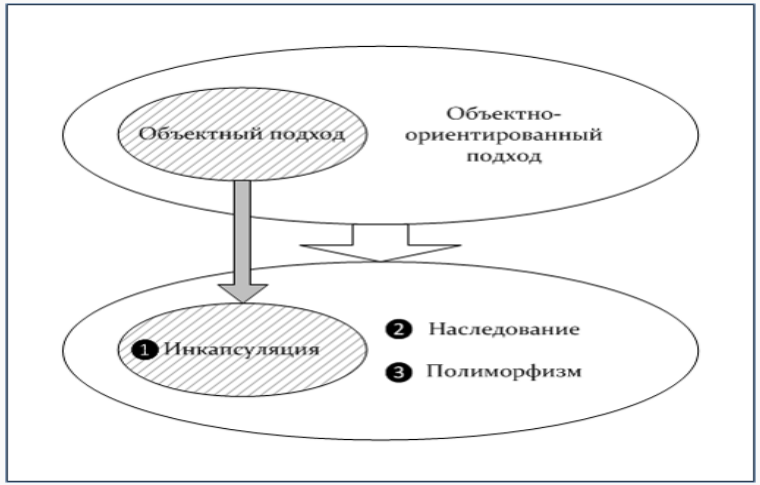
Конструктор переноса: пример (2 / 2)

```
// к о н с т р у к т о р  к о п и р о в а н и я
Alpha::Alpha(const Alpha &a) : sz(a.sz)
{
        d = new double[sz];
        /* … */
        for(size_t i = 0; i < sz; i++)
                d[i] = a.d[i];
}

// к о н с т р у к т о р  п е р е н о с а
Alpha::Alpha(Alpha &&a) : sz(a.sz)
{
        d  = a.d;
        // п е р е н а с т р о й к а  п а р а м е т р а
        a.d  = nullptr; // C++11
        a.sz = 0;
}
```

Объектный или объектно-ориентированный подход?

Объектно-ориентированное программирование расширяет объектный подход и вводит отношения «тип – подтип», прибегая для этого к механизмам **наследования** и **полиморфизма**.



Наследование: ключевые понятия

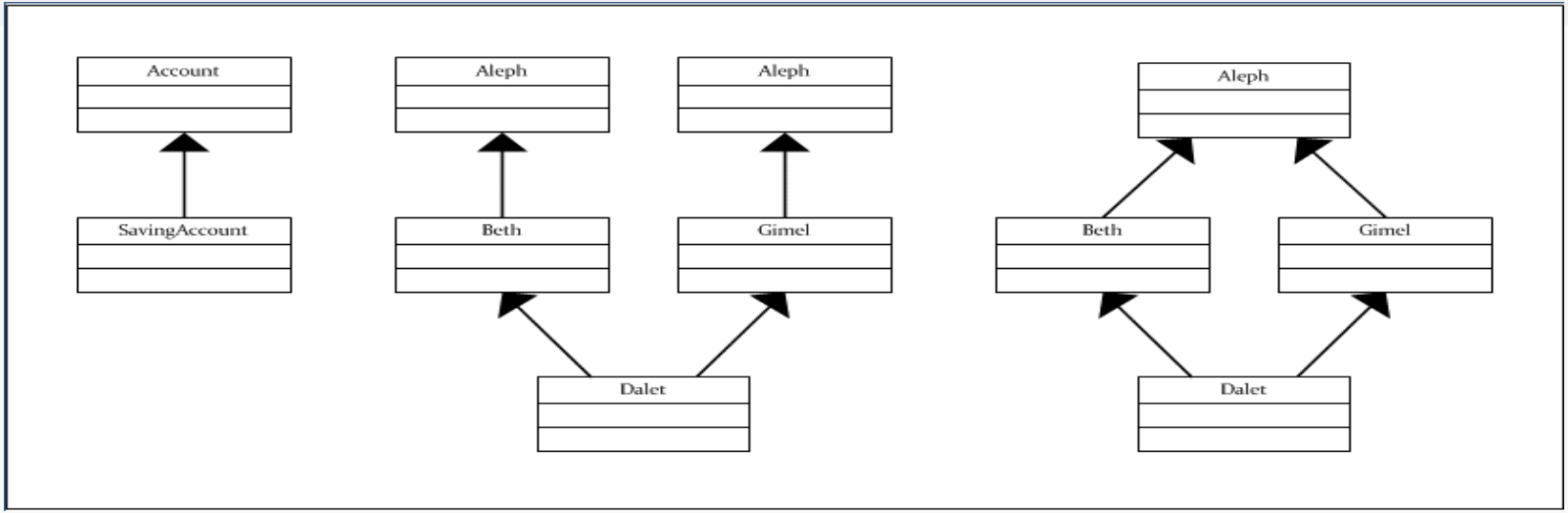
Наследование содействует **повторному использованию** атрибутов и методов класса, а значит, делает процесс разработки ПО более эффективным. Возникающие между классами A и B отношения наследования позволяют, например, говорить, что:

- класс A является **базовым** (**родительским**) классом, классом-**предком**, **надклассом** (англ. superclass);
- класс B является **производным** (**дочерним**) классом, классом-**потомком**, **подклассом** (англ. subclass).

Отношения наследования связывают классы в **иерархию наследования**, вид которой зависит от числа базовых классов у каждого производного:

- при **одиночном наследовании** иерархия имеет вид дерева;
- при **множественном наследовании** — вид направленного ациклического графа (НАГ) произвольного вида.

Наследование «в картинках»



Одиночное наследование (слева), множественное наследование (в центре), виртуальное множественное наследование (справа)

Полиморфизм подклассов

Отношение между классом и подклассом, позволяющее указателю или ссылке на базовый класс без вмешательства программиста адресовать объект производного класса, возникает в C++ благодаря поддержке **полиморфизма**.

Полиморфизм позволяет предложить такую реализацию ядра объектно-ориентированного приложения, которая не будет зависеть от конкретных используемых подклассов.

Раннее и позднее связывание

В рамках классического объектного подхода, — а равно и процедурного программирования, — адрес вызываемой функции (метода класса) определяется на этапе компиляции (сборки). Такой порядок связывания вызова функции и ее адреса получил название **раннего** (**статического**).

Позднее (**динамическое**) связывание состоит в нахождении (разрешении) нужной функции во время исполнения кода. При этом работа по разрешению типов перекладывается с программиста на компилятор.

В языке C++ динамическое связывание поддерживается механизмом **виртуальных** методов класса, для работы с которыми компиляторы строят **таблицы виртуальных методов** (англ. VMT, virtual method table).

Базовые и производные классы

ОО-проектирование допускает существование классов, которые могут выполнять чисто технические функции, моделировать абстрактные сущности и отличаться функциональной неполнотой:

- не подлежащий реализации в виде экземпляров (объектов) базовый класс может оставаться **абстрактным**. В противовес абстрактным базовым классам классы, предполагающие создание экземпляров, именуют **конкретными**;
- (абстрактные) базовые классы **специфицируют открытые интерфейсы** иерархий и **содержат общие** для всех подклассов атрибуты и методы (или их прототипы).

Множество подклассов любого базового класса ограничено иерархией наследования, но потенциально бесконечно (ввиду отсутствия пределов по расширению иерархии вглубь и вширь).

Определение наследования

Определение отношения наследования имеет вид (для одиночного наследования):

```
// заголовок класса
class  <имя производного класса> :
                    <уровень доступа> <имя базового класса>
// тело класса
{                   /* ... */                       };
```

где <уровень доступа> — ключевое слово public, private или protected, а <имя базового класса> — имя ранее определенного (не описанного!) класса.

Производный класс расширяет функциональность базового класса

В зависимости от уровня доступа к членам базового класса говорят об **открытом**, **закрытом** или **защищенном** наследовании.

Определение наследования: пример

```
// описание производного класса
//(не включает список базовых классов!)
class Deposit;
/* … */
// определение базового класса
class Account
{
        /* … */
};
// определение производного класса
class Deposit : public Account
{
        /* … */
};
```

Уровень доступа при наследовании

Квалификатор доступа члена базового класса	Область видимости
private	Доступны внутри данного класса и из дружественных функций
protected	Также доступны в производных классах
public	Доступны всюду

Квалификатор доступа члена базового класса	Уровень доступа при наследовании		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	---	--	--

Уровень доступа: пример

```
class Base {
  int m_Foo;
protected:
  void SetFoo(int in_Foo) { m_Foo = in_Foo; }
public:
  int GetFoo() const { return m_Foo; }
};
class Derived : public Base {
public:
  void IncFoo() {
    m_Foo++;  // Cannot access private member
    SetFoo(GetFoo() + 1); // OK
  }
};
```

```
void main() {
  Derived d;
  d.m_Foo = 0; // Cannot access private member
  d.SetFoo(0); // Cannot access protected member
  d.IncFoo();  // OK
}
```

Защищенные и закрытые члены класса

Атрибуты и методы базового класса, как правило, должны быть **непосредственно доступны для производных классов** и непосредственно недоступны для прочих компонентов программы. В этом случае они помещаются в секцию protected, в результате чего защищенные члены данных и методы базового класса:

- доступны производному классу (прямому потомку);
- недоступны классам вне рассматриваемой иерархии, глобальным функциям и вызывающей программе.

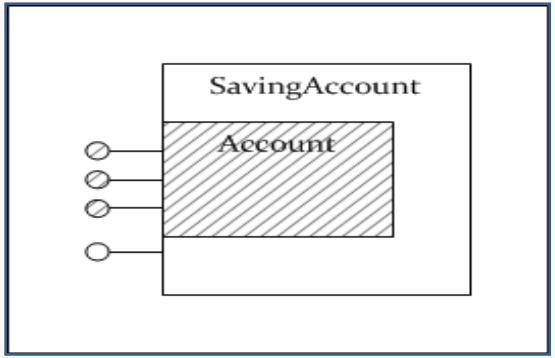
Если **наличие прямого доступа** к члену класса со стороны производных классов **нежелательно**, он вводится как **закрытый**.

Закрытые члены класса не наследуются потомками. Для доступа к ним класс-потомок должен быть объявлен в классе-предке как дружественный. **Отношения дружественности не наследуются.**

Производный объект «в разрезе»

Согласно объектной модели языка C++ экземпляр (объект) производного класса состоит из **подобъектов**, соответствующих каждому из его базовых классов,
а также части, объединяющей нестатические члены самого класса.

Заметим, что объект производного класса **имеет** непосредственный **доступ только** к защищенным и открытым членам **входящего в него подобъекта**



Перегрузка и перекрытие членов класса

Члены данных базового класса **могут перекрываться** одноименными членами данных производного класса, при этом их типы не должны обязательно совпадать. (Для доступа к члену базового класса его имя должно быть квалифицировано.)

Методы базового и производного классов **не образуют множество перегруженных функций**. В этом случае методы производного класса не перегружают (англ. overload), а перекрывают (англ. override) методы базового.

Для явного создания объединенного множества перегруженных функций базового и производного классов используется объявление using, которое вводит именованный член базового класса в область видимости производного.

Перегрузка и перекрытие членов класса: пример 1

В данном примере в область видимости производного класса попадают все одноименные методы базового класса, а не только некоторые из них.

```
class Account
{ /* … */
          void display(const char *fmt);
          void display(const int mode = 0);
};
class Deposit : public Account
{
          void display(const string &fmt);
          using Account::display;
          /* … */
};
```

Перегрузка и перекрытие членов класса: пример 2

```
int x;
void f(int a, int b) { x = a + b; }
class B {
  int x;
public:
  void f() { x = 2; }
};
class D : public B {
public:
  void f(int a) { ::x = a; }
  void g() {
    f();     // Error
    D::f(1);
    f(5, 1); // Error
    x = 2;   // Error
  }
};
```

```
void main()
{
  D d;
  d.f(); // Error
  f(5);  // Error
  f('+', 1);
}
```

Статические члены данных и наследование

При наличии в базовом классе статического члена данных объекты производного класса ссылаются на его единственный разделяемый статический атрибут.

Никакие другие экземпляры данного атрибута при наследовании от содержащего его класса не создаются.

Порядок вызова конструкторов производных классов (1 / 2)

Порядок вызова конструкторов объектов-членов, а также базовых классов **при построении объекта производного класса** не зависит от порядка их перечисления в списке инициализации конструктора производного класса и является следующим:

- конструктор базового класса (если таковых несколько, конструкторы вызываются в порядке перечисления имен классов в списке базовых классов);
- конструктор объекта-члена (если таковых несколько, конструкторы вызываются в порядке объявления членов данных в определении класса);
- конструктор производного класса.

Порядок вызова конструкторов производных классов (2 / 2)

Конструктор производного класса может вызывать (в списке инициализации) только конструкторы классов, непосредственно являющихся базовыми для данного (прямых предков)

Примечание: правильно спроектированный конструктор производного класса не должен инициализировать атрибуты базового класса напрямую (путем присваивания значений).

Список инициализации при наследовании: пример

```
class Alpha {
public:
        // Alpha();
        Alpha(int i); /* … */
};
class Beta : public Alpha {
public:
        Beta() : _s("dictum factum") { }
        // Beta() : Alpha(), _s("dictum factum") { }
        Beta(int i, string s) : Alpha(i), _s(s) { }
protected:
        string _s; /* … */
};
```

Порядок вызова деструкторов производных классов

Порядок вызова деструкторов при уничтожении объекта производного класса прямо противоположен порядку вызова конструкторов и является следующим:

- деструктор производного класса;
- деструктор объекта-члена (или нескольких);
- деструктор базового класса (или нескольких).

Взаимная противоположность порядка вызова конструкторов и деструкторов является **строгой гарантией** языка C++.

Порядок вызова конструкторов и деструкторов: пример

```
class B {
public:
  B() { cout << "B::B();" << endl; }
  B(const B&) {
    cout << "B::B(const B&);" << endl;
  }
  ~B() { cout << "B::~B();" << endl; }
};
class D : public B {
public:
  D() { cout << "D::D();" << endl; }
  D(const D& d) : B(d) {
    cout << "D::D(const D&);" << endl;
  }
  ~D() { cout << "D::~D();" << endl; }
};
```

```
D f(D& x, D& y) {
  return x;
}
void main() {
  D d;
  d = f(d, d);
}
```

```
B::B();
D::D();
B::B(const B&);
D::D(const D&);
D::~D();
B::~B();
D::~D();
B::~B();
```

Виртуальные функции (1 / 4)

Методы, результат разрешения вызова которых зависит от «реального» (динамического) типа объекта, доступного по указателю или ссылке, называются **виртуальными** и при определении в базовом классе снабжаются спецификатором virtual.

Примечание: в этом контексте тип непосредственно определяемого экземпляра, ссылки или указателя на объект называется статическим. Для самого объекта любого типа (автоматической переменной) статический и динамический тип совпадают.

По умолчанию объектная модель C++ работает с **невиртуальными** методами. Механизм виртуальных функций работает только в случае **косвенной адресации** (по указателю или ссылке).

Виртуальные функции (2 / 4)

Значения **формальных параметров** виртуальных функций определяются (а) на этапе компиляции (б) типом объекта, через который осуществляется вызов.

Отмена действия механизма **виртуализации** возможна и достигается статическим вызовом метода при помощи операции разрешения области видимости (::).

```
class Alpha {
        /* … */
        virtual void display();
};
class Beta : public Alpha {
        void display();
}
```

Виртуальные функции (3/ 4)

```
class X { public:
  virtual void g(int x) { h(); cout << "X::g() "; }
  void h() { t(); cout << "X::h() "; }
  virtual void t() { cout << "X::t() "; }
};
class Z: public X { public:
  void g(int y) { h(); cout << "Z::g() "; }
  virtual void h() { t(1); cout << "Z::h() "; }
  virtual void t(int k) { cout << "Z::t() "; }
};
void main() {
  X a; Z b; X* p = &b;
  p->g(2); // Z::t() Z::g() Z::g()
  p->h();  // X::t() X::h()
  p->t(5); // Error: X::t doesn't take one argument
}
```

Не имеет значения, т.к. у класса нет наследников

Переопределения не происходит, т.к. у функции другой прототип

Виртуальные функции (3/4)

```
class A {
public:
  virtual void Foo(int k = 0, int i = 1)
    { cout << "A::Foo(" << i << ")" << endl; }
};
class B : public A {
public:
  void Foo(int k, int i = 0)
    { cout << "B::Foo(" << i << ")" << endl; }
};
void main() {
  B b;
  A* pa = &b; pa->Foo(5); // B::Foo(5, 1)
  B* pb = &b; pb->Foo(); // Error
}
```

Это объявление функции скрывает версию функции без параметров

Значения функции по умолчанию фиксируются на этапе компиляции

Виртуальные деструкторы

```
class A {
public:
  A() { cout << "A::A()" << endl; }
  virtual ~A() { cout << "A::~A()" << endl; }
};
class B : public A {
public:
  B() { cout << "B::B()" << endl; }
  ~B() { cout << "B::~B()" << endl; }
};
void main()
{
  A* pa = new B;
  delete pa;
}
```

```
A::A()
B::B()
B::~B()
A::~A()
```

Чистые виртуальные функции и абстрактные классы (1 / 3)

Класс, где виртуальный метод объявляется впервые, должен определять его тело либо декларировать метод как не имеющую собственной реализации **чистую виртуальную функцию**.

Производный класс может **наследовать** реализацию виртуального метода из базового класса или **перекрывать** его собственной реализацией, при этом прототипы обеих реализаций обязаны совпадать.

Например: `virtual void display() = 0;`

Чистые виртуальные функции и абстрактные классы (2 / 3)

Класс, который определяет или наследует хотя бы одну чистую виртуальную функцию, является абстрактным.

Экземпляры абстрактных классов создавать нельзя. Абстрактный класс может реализовываться только как подобъект производного, неабстрактного класса.

Чистые виртуальные функции и абстрактные классы (3 / 3)

```
class Object {
public:
  virtual const char* ToString() = 0;
};
class PointSet : public Object {
public:
  const char* ToString() { return "PointSet"; }
};
void main()
{
  Object obj; // Error - cannot instantiate abstract class
  PointSet set;
  Object* pobj = &set;
  cout << pobj->ToString(); // PointSet
}
```

Множественное наследование (1 / 2)

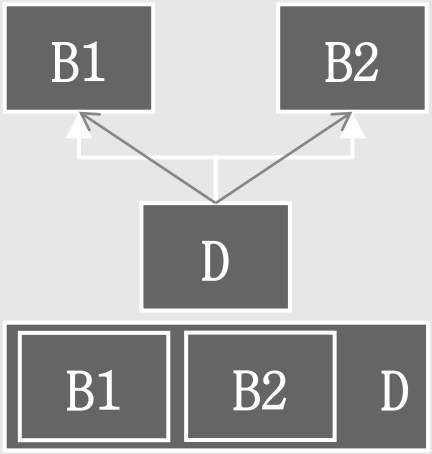
Множественное наследование в ООП — это наследование от двух и более базовых классов, возможно, с различным уровнем доступа. Язык не накладывает ограничений на количество базовых классов.

При множественном наследовании **конструкторы** базовых классов вызываются **в порядке перечисления имен классов** в списке базовых классов. Порядок вызова **деструкторов** ему прямо противоположен.

Унаследованные от разных базовых классов методы не образуют множество перегруженных функций, а потому разрешаются только по имени, без учета их сигнатур.

Множественное наследование (2/2)

```
class B1 {
  int a;
public:
  B1(int x) { a = x; }
};
class B2 {
  int b;
public:
  B2(int x) { b = x; }
};
class D : public B1, public B2 {
  int c;
public:
  D(int x, int y, int z) :
    B2(x), B1(y)
  { c = z; }
};
```



Конструкторы базовых классов вызываются в порядке перечисления имен классов в списке базовых классов, а не в списке инициализации

Виртуальное наследование (1 / 2)

При множественном наследовании возможна ситуация неоднократного включения (дублирования) подобъекта одного и того же базового класса в состав производного. Связанные с нею проблемы и неоднозначности снимает **виртуальное наследование**.

Суть виртуального наследования — включение в состав класса **единственного разделяемого подобъекта** базового класса (виртуального базового класса).

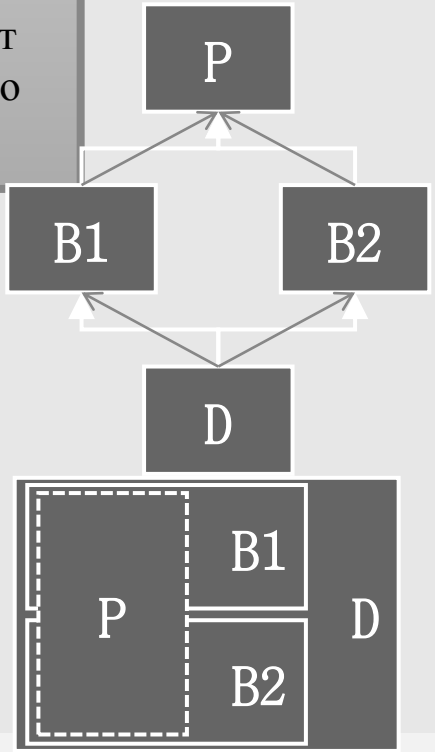
Виртуальное наследование не характеризует базовый класс, а лишь описывает его отношение к производному.

Использование виртуального наследования должно быть взвешенным проектным решением конкретных проблем объектно-ориентированного проектирования.

Виртуальное наследование (2 / 2)

```
struct P { int p; };
struct B1 : virtual P {};
struct B2 : virtual P {};
struct D : B1, B2 {};
void main()
{
  D d;
  d.p; // OK
  d.B1::p; // OK
  d.B2::p; // OK
  static_cast<B1>(d).p; // OK
  static_cast<B2>(d).p; // OK
}
```

Ключевое слово `virtual` должно быть объявлено во всех наследованиях от класса, порождающего ромбовидность



Конструкторы объектов при виртуальном наследовании

Виртуальные базовые классы конструируются перед невиртуальными независимо от их расположения
в иерархии наследования.

В промежуточных производных классах прямые вызовы конструкторов виртуальных базовых классов автоматически подавляются.

Динамическая идентификация типов времени выполнения (RTTI)

Динамическая идентификация типов времени выполнения (англ. Real-Time Type Identification) обеспечивает **специальную поддержку полиморфизма** и **позволяет** программе **узнать реальный производный тип объекта**, адресуемого по ссылке или по указателю на базовый класс.

Поддержка RTTI в C++ реализована двумя операциями:

- операция dynamic_cast поддерживает преобразование типов времени выполнения;
- операция typeid идентифицирует реальный тип выражения.

Операции RTTI — это события времени выполнения для классов с виртуальными функциями и события времени компиляции для остальных типов.

Исследование RTTI-информации полезно, в частности, при решении задач системного программирования.

Операция dynamic_cast (1 / 3)

Встроенная унарная операция dynamic_cast языка C++ позволяет:

- **безопасно трансформировать указатель** на базовый класс в указатель на производный класс (с возвратом нулевого указателя при невозможности выполнения трансформации);
- **преобразовывать леводопустимые значения**, ссылающиеся на базовый класс, в ссылки на производный класс (с возбуждением исключения bad_cast при ошибке).

Единственным операндом dynamic_cast должен являться тип класса, в котором имеется хотя бы один виртуальный метод.

Операция dynamic_cast (2 / 3)

Например (для указателей):

- пусть классы Alpha и Beta образуют полиморфную иерархию, в которой класс Beta открыто наследуется от класса Alpha, тогда:

```
Alpha *al  = new Beta;
if(Beta *bt = dynamic_cast<Beta*>(al))
{
        /* у с п е ш н о */
}
else
{
        /* н е у с п е ш н о */
}
```

Операция dynamic_cast (3/3)

Например (для ссылок):

- пусть классы Alpha и Beta образуют полиморфную иерархию, в которой класс Beta открыто наследуется от класса Alpha, тогда:

```
#include <typeinfo> // для std::bad_cast
void foo(Alpha &al)
{       /* … */
        try {
                Beta &bt = dynamic_cast<Beta&>(al))
        }
        catch(std::bad_cast) {  /* … */  }
}
```

Операция typeid (1 / 2)

Встроенная унарная операция typeid позволяет **установить фактический тип выражения-операнда** и может использоваться с выражениями и именами любых типов (включая выражения встроенных типов и константы).

Операция typeid имеет тип (возвращает значение типа) type_info и требует подключения заголовочного файла <typeinfo>.

Реализация класса type_info зависит от компилятора, но в общем и целом позволяет получить результат в виде C-строки (const char*), присваивать объекты type_info друг другу (operator =), а также сравнивать их на равенство и неравенство (operator ==, operator !=).

Операция typeid (2 / 2)

Например:

- пусть классы Alpha и Beta образуют полиморфную иерархию, в которой класс Beta открыто наследует классу Alpha, тогда:

```
#include <typeinfo> // для type_info

Alpha *al = new Alpha;
Beta   *bt = new Beta;

if(typeid(al)  == typeid(Alpha*)) /* … */
if(typeid(*al) == typeid(Alpha))  /* … */
```

Вопросы производительности

- Глубина цепочки наследования не увеличивает затраты времени и не ограничивает доступ к унаследованным членам базовых классов.
- Вызов виртуальной функции в большинстве случаев не менее эффективен, чем косвенный вызов функции по указателю на нее.
- При использовании встроенных конструкторов глубина иерархии наследования почти не влияет на производительность.
- Отмена действия механизма виртуализации, как правило, необходима по соображениям повышения эффективности.

Валентина Глазкова

Спасибо за внимание!