Основы Программирования

Лекция #5

Темы

- Модификаторы доступа
- Класс
- Полиморфизм
- Наследование
- Спецификаторы доступа
- Виртуальные функции
- Чистые виртуальные функции
- Абстрактный класс
- Финальные методы/классы
- Множественное наследование
- Виртуальный класс
- Динамическая идентификация типов

Модификаторы доступа

- Модификаторы доступа позволяют ограничивать доступ к методам и полям класса

```
struct Alpha {
        Alpha(int n): _n(n) { }
        int getN() { return _n; }
private:
        int _n;
}
// …
Alpha a(5);
cout << a.getN();
cout << a._n; // ошибка
```

Модификаторы доступа

- Модификаторы доступа позволяют ограничивать доступ к методам и полям класса
- Различают 3 модификатора:
  1. public
  2. protected
  3. private
- Для struct по умолчанию все атрибуты являются открытыми

Класс

- Ключевое слово struct можно заменить на class, тогда все поля и методы по умолчанию будут private

```
class Alpha {
        int _n;
 public:
        Alpha(int n): _n(n) { }
        int getN() { return _n; }
 }
// …
 Alpha a(5);
 cout << a.getN();
 cout << a._n; // ошибка
```

Скобочные инициализации членов данных (C++11)

```
#include <iostream>
 using namespace std;
 int counter = int();
 struct Sample {
        string msg{“Sample text”};        // форма 1
        int id = ++counter;                // форма 2
        int n{42};                         // форма 3
```

Скобочные инициализации членов данных (C++11)

```
// struct Sample
        Sample() {}
        Sample(int n) : n(n) {}
};

 int main() {
        Sample sample;
        cout << sample.id << “\t” << sample.n
                << “\t” << sample.msg; // 1, 42, “Sample text”
        return 0;
}
```

Полиморфизм (1/2)

- **Полиморфизм** (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий.
- В C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).
- В более общем смысле, концепцией полиморфизма является идея "***один интерфейс, множество методов***"

Полиморфизм: перегрузка функций (2/2)

```
double length(int x1, int y1, int x2, int y2);
double length(Point p1, Point p2);
double length(Segment s);
```

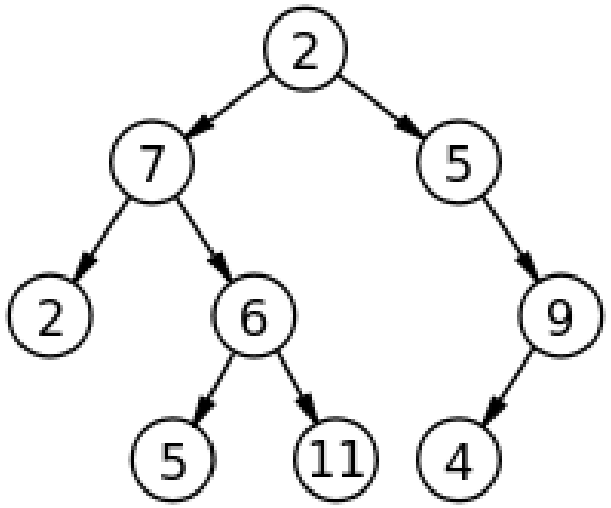
В С полиморфизма нет и приходится писать отельные функции для различных аргументов. Пример: abs(int), fabs(float), labs(long).

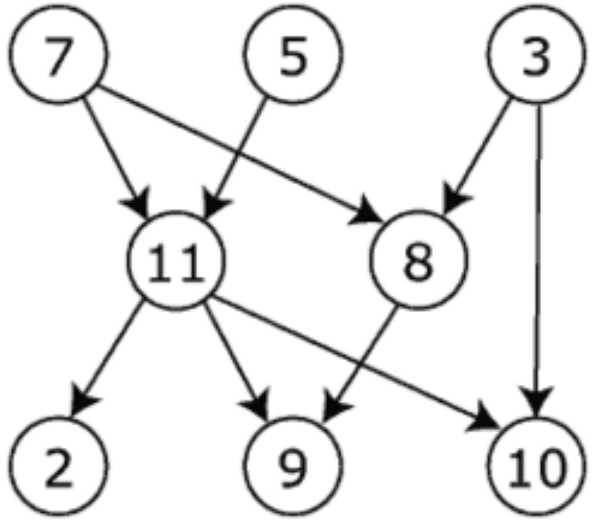
Наследование: ключевые понятия (1/2)

- **Наследование** - это механизм создания нового класса на основе уже существующего. При этом к существующему классу могут быть добавлены новые элементы (данные и функции), либо существующие функции могут быть изменены.
- 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 (base) или родительским классом, классом-предком, суперклассом (superclass)
  - Класс B является производным (derived) или дочерним классом, классом-потомком, подклассом (subclass)

Наследование: ключевые понятия (2/2)

- 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 произвольного вида.





Наследование: пример (1/2)

```
// описание производного класса
class Deposit;

// определение классов
class Account { /*… */ };
class Deposit : public Account {
        //….
};
```

Наследование: пример (2/2)

```
// описание производного класса
class Derived;

// определение классов
class BaseA { /*… */ };
class BaseB { /*… */ };
class BaseC { /*… */ };
class Derived : public BaseA, protected BaseB, private BaseC {
        //….
};
```

Защищенный 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
- Если **наличие прямого доступа** к члену класса со стороны производных классов **нежелательно**, он вводится как **закрытый** (**protected**).

Спецификаторы доступа (1/4)

При открытом (public) наследовании порожденный класс имеет доступ к наследуемым членам базового класса с видимостью **public** и **protected**. Члены базового класса с видимостью **private** – недоступны.

Спецификатор доступа	Внутри класса	В порожденном класса	Вне класса
private	+	-	-
protected	+	+	-
public	+	+	+

```
class A { };
class B : public A { };
```

Спецификаторы доступа (2/4)

При защищенном (protected) наследовании порожденный класс имеет доступ к наследуемым членам базового класса с видимостью **public** и **protected**. Члены базового класса с видимостью **private** – недоступны.

Спецификатор доступа	Внутри класса	В порожденном класса	Вне класса
private	+	-	-
protected	+	+	-
public	+	+	-

```
class A { };
class B : protected A { };
```

Спецификаторы доступа (3/4)

При закрытом (protected) наследовании порожденный класс не имеет доступ к наследуемым членам базового класса.

Спецификатор доступа	Внутри класса	В порожденном класса	Вне класса
private	+	-	-
protected	+	-	-
public	+	-	-

```
class A { };
class B : private A { };
```

Спецификаторы доступа (4/4)

- По умолчанию наследование **struct** открытое

```
struct A { };
struct B : A { };                 // равносильно
struct B : public A { };          // равносильно
```

- По умолчанию наследование **class** закрытое

```
class A { };
class B : A { };                  // равносильно
class B : private A { };          // равносильно
```

Перегрузка и перекрытие членов класса (1/5)

```
struct A {
        void f(int n) { cout << “A”; }
};
struct B : A {
        void f(long n) { cout << “B”; }
};
//…
B b;
b.f(1); // что будет выведено?
```

Перегрузка и перекрытие членов класса (2/5)

- **Члены данных** базового класса **могут перекрываться** одноименными членами данных производного класс, при этом их типы не должны обязательно совпадать. (Для доступа к члены 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 (overload), а перекрывают (override) методы базового.
- Для явного **создания объединенного множества** перегруженных методов базового и производного классов используется объявление **using**, которое вводит именованный член базового класса в область видимости производного

Перегрузка и перекрытие членов класса (3/5)

```
struct A {
        void f(int n) { cout << “A”; }
};
struct B : A {
         void f(long n) { cout << “B”; }
        using A::f;
};
//…
 B b;
 b.f(1); // “A”
```

Перегрузка и перекрытие членов класса (4/5)

```
class Account {
        public:
                void display(const char* fmt) { cout << “1”; }
                void display(const int mode = 0) { cout << “2”; }
};
class Deposit : public Account {
        public:
                void display(const std::string& fmt) { cout << “3”; }
};
// …
Deposit d;
d.display(); // что будет выведено?
```

Перегрузка и перекрытие членов класса (5/5)

```
class Account {
        public:
                void display(const char* fmt) { cout << “1”; }
                void display(const int mode = 0) { cout << “2”; }
};
class Deposit : public Account {
        public:
                void display(const std::string& fmt) { cout << “3”; }
                using Account::display
};
// …
Deposit d;
d.display();
```

Порядок вызова конструкторов производных классов (1/2)

- **Порядок вызова конструкторов** объектов, а также базовых классов **при построении объекта производного класса** не зависит от порядка из 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 производного класса.

Порядок вызова деструкторов производных классов (2/2)

- Порядок вызова деструкторов при уничтожении объекта производного класса прямо противоположен порядку вызова конструкторов и является следующим:
  - Деструктор производного класса;
  - Деструктор базового класса (или несколько)

Виртуальные функции (1/2)

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

Виртуальные функции (2/2)

- Вызов конкретной функции (или значения формальных параметров виртуальной функции) определяется
  1. На этапе компиляции
  2. Типом объекта, через который осуществляется вызов
- Отмена действия механизма виртуализации возможна и достигается статическим вызовом метода при помощи операции разрешения области видимости (::)

```
struct Alpha {
        virtual void display(); // возможен вызов Alpa::display()
};
struct Beta : Alpha {
        void display();    // возможен вызов Beta::display()
};
```

Явное перекрытие виртуальных функций (C++ 11)

```
struct Alpha {
        virtual void foo();
        void bar();
};
struct Beta : Alpha {
        // спецификатора override гарантирует, что функция
        // (а) является виртуальной и (б) перекрывает виртуальную
        // функцию базового класса с идентичной ей сигнатурой
        void foo() override;                // допустимо
        // void foo() const override;       // недопустимо
        // void bar() override;              // недопустимо
};
```

Чистые виртуальные функции

- Производный класс может **наследовать** реализацию виртуального метода из базового класса или **перекрывать** его собственной реализацией, при этом прототипы обеих реализаций обязаны совпадать.
- Класс в котором виртуальный метод описывается впервые, должен определять его тело либо декларировать метод как не имеющую собственной реализации **чистую виртуальную функцию**.

virtual void display() = 0;

Абстрактные классы

- Класс, который определяет или наследует хотя бы одну чистую виртуальную функцию, является **абстрактным**.
- **Экземпляры** абстрактных классов **создать нельзя**. Абстрактный класс может реализовываться только как подобъект производного, неабстрактного класса.
- Чистые виртуальные функции **могут иметь** тело, определенное строго вне тела класса. Обращение к телу чистой виртуально функции может производиться только статически (при помощи операции разрешения области видимости), но не динамически (при помощи механизма виртуализации).

Чистые виртуальные функции и абстрактные классы: пример

```
struct Alpha {
        virtual void display() = 0;
};
void Alpha::display() { /*…*/ }
struct Beta : Alpha {
        void display() {
                // статический вызов чистой виртуальной функции
                Alpha::display();
        }
};
```

Деструктор абстрактных классов

- Деструктор класса может быть чистой виртуальной функцией, однако и в этом случае он **должен** (а не может) иметь тело (определяемое вне тела класса).
- Полиморфное обращение к чистой виртуальной функции из деструктора и из конструктора класса **запрещено** (так как влечен неопределенное поведение).

Деструктор абстрактных классов

```
 struct Alpha {
        virtual ~Alpha() = 0;
};
 Alpha::~Alpha() {}
 struct Beta : Alpha {};
 // Alpha alpha;      // недопустимо, абстрактный класс
 Beta beta;
```

Финальные методы и финальные классы C++ 11

```
 struct Alpha {
        virtual void foo() final;
        // void bar() final;            // недопустимо
};
struct Beta final : Alpha {
        // void foo();                  // недопустимо
};
/*
 struct Gamma : Beta { };               //недопустимо
*/
```

Множественное наследование

- **Множественное наследование** в ООП – это наследование от двух и более базовых классов, возможно, с различным уровнем доступа (нет никаких ограничений на количество).
- При множественном наследовании **конструкторы** базовых классов вызываются **в порядке перечисления имен классов** в списке базовых классов. Порядок вызова деструкторов ему прямо противоположен.

```
class Derived : public BaseA, protected BaseB, private BaseC {
        //….
};
```

Виртуальное наследование

- При множественном наследовании возможна ситуация неоднократного включения подобъекта одного и того же базового класса в состав производного. Связанные с нею проблемы и неоднозначности снимает виртуальное наследование.
- Суть – включение в состав класса единственного разделяемого подобъекта (виртуального базового класса).

Виртуальное наследование

```
struct A { };
struct B1 : A { };
struct B2 : A { };
struct C : B1, B2 { }; // проблема. Класс A представлен дважды
```

Виртуальное наследование

```
struct A { };
struct B1 : virtual A { };
struct B2 : virtual A { };
struct C : B1, B2 { };
```

Полиморфизм классов

- В том случае, если базовый и производный классы имеют общий открытый интерфейс, говорят, что производный класс представляет собой **подкласс** базового

- 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.

Автоматически генерируемый конструктор по умолчанию

- В случае отсутствия в классе явных конструкторов любого типа **компилятор самостоятельно неявно определяет** конструктор по умолчанию как встраиваемый (inline) открытый (public) метод данного класса
- По умолчанию генерируется компилятором и работает точно так же, как явно определенный конструктор с **пустым телом** и **пустым списком инициализации**.
- В случае участия класса в иерархии наследования автоматически генерируемый конструктор по умолчанию вызывает конструкторы по умолчанию базовых классов и своих членов, не являющихся статическими.

Тривиальный конструктор по умолчанию

- Конструктор по умолчанию является травильным, если одновременно соблюдаются все следующие условия:
  1. Конструктор определен неявно или определен как default
  2. Класс не имеет виртуальных методов
  3. Класс не имеет виртуальных базовых классов
  4. Каждый непосредственный предок класса имеет тривиальный конструктор по умолчанию
  5. Каждый нестатический член класса имеет тривиальный конструктор
- Все это также справедливо для конструкторов копирования/переноса и деструкторов класса

Принуждение и подавление генерации конструкторов и деструкторов (C++ 11)

- Автоматическая генерация конструктора по умолчанию, конструктора копирования/переноса, деструктора компилятором может быть подавлена программистом, так и, наоборот, форсирована

```
class Sample {
public:
        Sample() = delete;          // запрет актогенерации
        Sample(int n, int m);
        ~Sample() = default;        //принудительная автогенерация
};
```

Динамическая идентификация типов времени выполнения

- Динамическая идентификация типов времени выполнения позволяет программе узнать реальный производный тип объекта, адресуемого по ссылке или по указателю на базовый класс. Реализована 2 операциями:
  - **dynamic_cast** – поддерживает преобразование типов времени выполнения
  - операция **typeid** идентифицирует реальный тип выражения

Операция dynamic_cast

- Встроенная унарная операция **dynamic_cast** языка C++ позволяет **безопасно трансформировать указатель** на базовый класс в указатель на производный класс
  - При невозможности возвращается нулевой указатель

```
Alpha* alpha = new Beta;
if (Beta* beta = dynamic_cast<Beta*>(alpha)) {
        // успешно
} else {
        // неуспешно
}
```

Операция dynamic_cast (ссылки)

- При невозможности ссылочного преобразования генерируется bad_cast исключение

```
#include <typeinfo> // для std::bad_cast
 void foo(Alpha& alpha) {
 try {
        Beta& beta = dynamic_cast<Beta&>(alpha);
        // ... Дальнейшая обработка
 } catch (std::bad_cast) {
        // обработка исключения
 }
```

Операция typeid

- Встроенная унарная операция typeid позволяет установить фактический тип любых объектов

```
#include <typeinfo>
 Alpha* alpha = new Alpha;

 if (typeid(alpha) == typeid(Alpha*)) { /*…*/ }
 if (typeid(*alpha) == typeid(Alpha)) { /*…*/ }
```