Варианты наследования

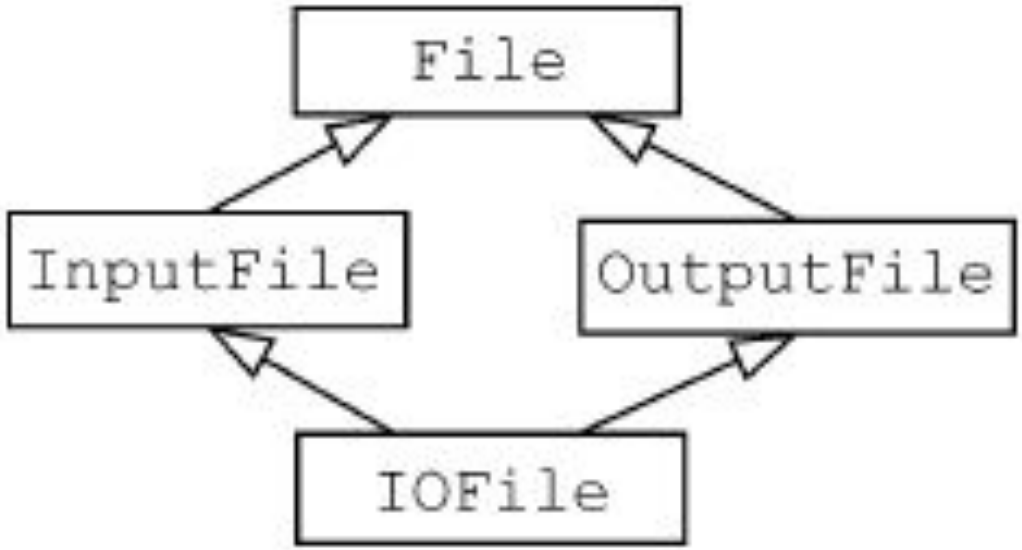
- По типу наследования
  - Публичное (открытое) наследование
  - Приватное (закрытое) наследование
  - Защищенное наследование
- По количеству базовых классов
  - Одиночное наследование (один базовый класс)
  - **Множественное наследование (два и более базовых классов)**

Проблема неоднозначности

```
class BorrowableItem { // нечто, что можно позаимствовать из библиотеки
public:
  void checkOut();
...
};
class ElectronicGadget {
private:
  bool checkOut() const; // выполняет самотестирование, возвращает
   // признак успешности теста
...
};
class MP3Player: public BorrowableItem, public ElectronicGadget {...}
/* здесь множественное наследование (в некоторых библиотеках реализована
функциональность, необходимая для MP3-плееров) */

MP3Player mp;
mp.checkout(); // неоднозначность! какой checkOut?
mp.ElectronicGadget::checkOut() // ошибка!  Попытка вызвать закрытый метод
mp.BorrowableItem::checkOut(); // вот какая checkOut  нужна!
```

Проблема ромба

```
class File {...};
class InputFile: public File {...};
class OutputFile: public File {...};
class IOFile:  public InputFile, public OutputFile {...};
/*данные базового класса дублируются в
    объекте подкласса столько раз, сколько
    имеетс
```



Виртуальное наследование

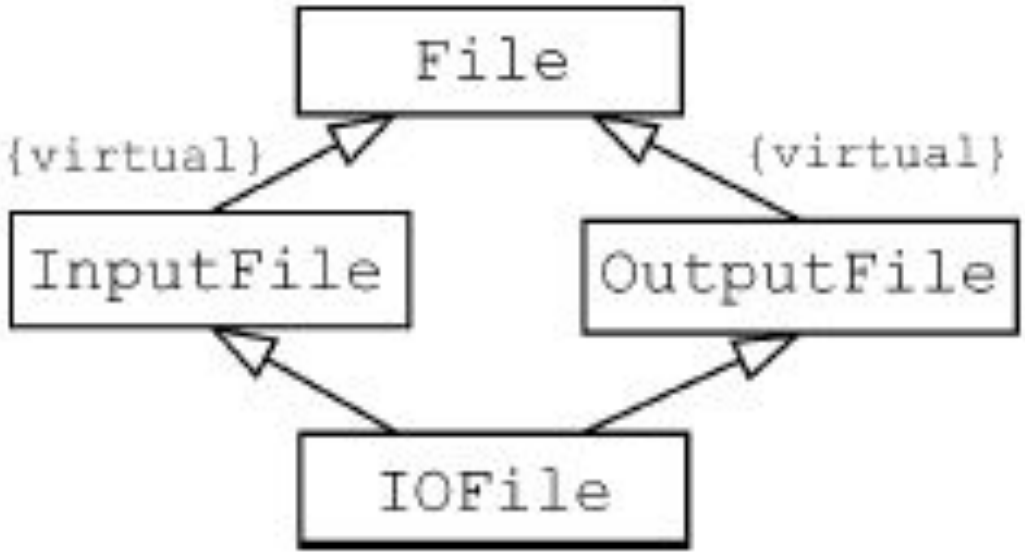
class File {...}; // **виртуальный базовый класс**

class InputFile: **virtual** public File {...};

class OutputFile: **virtua**l public File {...};

class IOFile: public InputFile, public OutputFile {...};

/*все непосредственные потомки используют виртуальное наследование */



В стандартной библиотеке C++ есть похожая иерархия, только классы в ней являются шаблонными и называются basic_ios, basic_istream, basic_ ostream и basic_iostream.

Виртуальное наследование требует затрат.

Размер объектов классов обычно оказывается больше, а доступ к полям виртуальных базовых классов медленнее.

Если избежать виртуальных базовых классов не удаётся, старайтесь не размещать в них данных. (Аналог: интерфейсы Java и. NET)

Семантика инициализации

Ответственность за инициализацию виртуального базового класса ложится на *самый дальний производный класс* в иерархии. Отсюда следует, что:

(1) классы, наследующие виртуальному базовому и требующие инициализации, должны знать обо всех своих виртуальных базовых классах, независимо от того, как далеко они от них находятся в иерархии

(2) когда в иерархию добавляется новый производный класс, он должен принять на себя ответственность за инициализацию виртуальных предков (как прямых, так и непрямых).

/* конструктор IOFile должен вызывать конструкторы

Пример интерфейсного класса

```
class IPerson {
public:
  virtual ~IPerson();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};  /*Пользователи IPerson должны программировать в терминах указателей и ссылок на IPerson, поскольку создавать объекты абстрактных классов запрещено.*/

Person * makePerson(DatabaseID personIdentifier);  // функция-фабрика
      //для создания объекта Person по уникальному идентификатору из базы данных
DatabaseID askUserForDatabaseID();
  // функция для запроса идентификатора у пользователя

DatabaseID id(askUserForDatabaseID());
Person * pp = makePerson(id);  // создать объект, поддерживающий интерфейс IPerson
... // манипулировать *pp при помощи методов IPerson
Delete pp; // удаляем объект, когда он больше не нужен
```

Предположим, что старый, ориентированный только на базы данных класс PersonInfo предоставляет почти все необходимое для реализации конкретных классов – наследников IPerson:

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseID pid)
  virtual ~PersonInfo();
  virtual const char *theName() const;
  virtual const char *theBirthDate() const;
  ...
private:
  virtual const char *valeDelimOpen() const;
  virtual const char *valeDelimClose() const;
    // позволяет производным классам задать
    // открывающие и закрывающие разделители
  ...
};
```

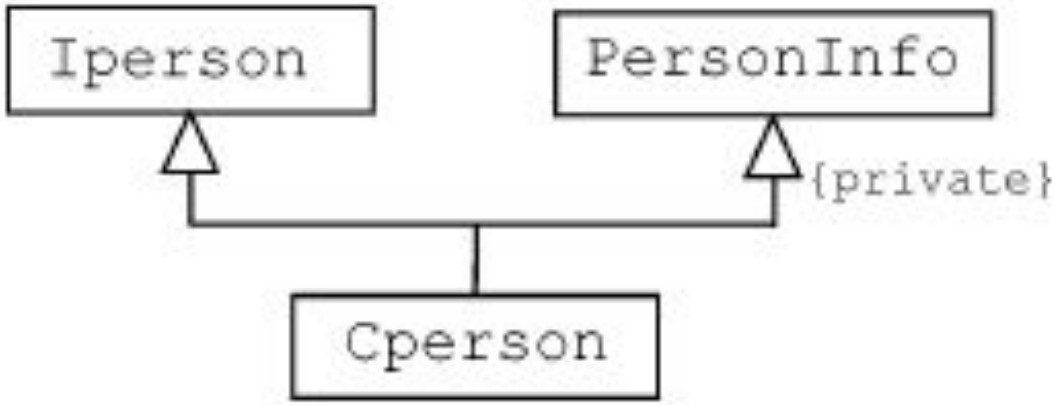
По умолчанию открывающим и закрывающим разделителями служат квадратные скобки, поэтому значение поля «Homer» будет отформатировано так: [Homer]

```
const char *PersonInfo::valueDelimOpen() const{
   return “[“; // открывающий разделитель по умолчанию
}
const char *PersonInfo::valueDelimClose() const{
   return “]“; // закрывающий разделитель по умолчанию
}
const char * PersonInfo::theName() const
{
  // резервирование буфера для возвращаемого значения; поскольку он
  // статический, автоматически инициализируется нулями
  static char value[Max_Formatted_Field_Value_Length];
     // скопировать открывающий разделитель
  std::strcpy(value, valueDelimOpen());
     // добавить к строке value значение из поля name объекта
     //(будьте осторожны – избегайте переполнения буфера!)
  std::strcpy(value, valueDelimClose()); // скопировать закрывающий разделитель
  return value;
}
```

PersonInfo упрощает реализацию некоторых функций CPerson.

Стало быть, речь идет об отношении «реализован посредством».

```
class CPerson: public IPerson, private PersonInfo {
  // используется множественное наследование
  // реализации функций-членов из интерфейса IPerson
public:
  explicit CPerson(DatabaseID pid): PersonInfo(pid){}
  virtual std::string name() const
{ return PersonInfo::theName();}
  virtual std::string birthDate() const
    { return PersonInfo::theBirthDate();}
private:
    // переопределения унаследованных виртуальных
    // функций, возвращающих строки-разделители
  const char * valeDelimOpen() const { return “”;}
  const char * valeDelimClose() const { return “”;}
};
```



• Множественное наследование сложнее одиночного. Оно может привести к неоднозначности и необходимости применять виртуальное наследование.

• Цена виртуального наследования – дополнительные затраты памяти, снижение быстродействия и усложнение операций инициализации и присваивания. На практике его разумно применять, когда виртуальные базовые классы не содержат данных.

• Множественное наследование вполне законно. Один из сценариев включает комбинацию открытого наследования интерфейсного класса и закрытого наследования класса, помогающего в

```
class CartoonCharacter {  // персонаж мультфильма
public:
    virtual void dance() {}  // по умолчанию – ничего не делать
    virtual void sing() {}
};
class Grasshopper: public CartoonCharacter {{  //кузнечик
public:
   virtual void dance () ; // Определение в другом месте,
   virtual void sing() ; // Определение в другом месте.
};
class Cricket: public CartoonCharacter //сверчок
public:
   virtual void dance () ;
   virtual void sing() ;
};
```

Множественное наследование?

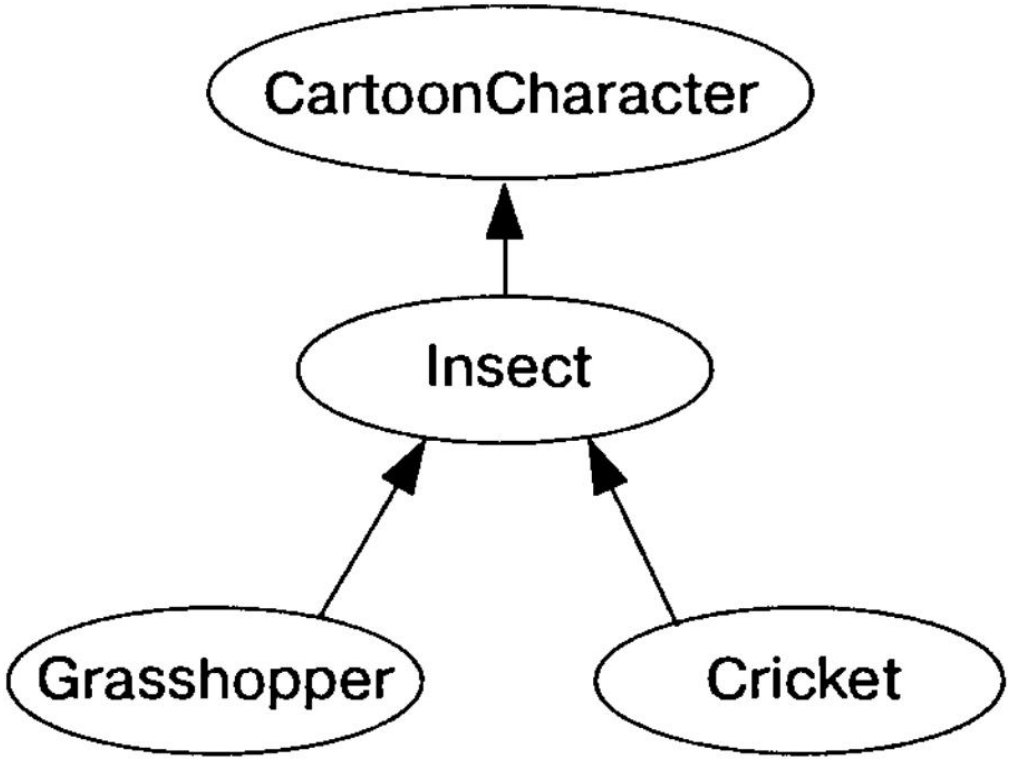
```
class Cricket : public CartoonCharacter,
        private Grasshopper {
public:
   virtual void dance() ;
  virtual void sin() ;
};

class Grasshopper: public CartoonCharacter {
public:
  virtual void dance() ;
  virtual void sing();
protected:
  virtual void danceCustomizationl();
  virtual void danceCustomization2();
  virtual void singCustomization() ;
};
```

```
void Grasshopper::dance()
    {
     выполнить общие танцевальные действия
     danceCustomizationl();
    выполнить другие общие танцевальные действия
     danceCustomization2();
     завершить общие танцевальные действия
    }
```

```
class Cricket:public CartoonCharacter,
        private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }
protected:
  virtual void danceCustomizationl();
  virtual void danceCustomization2();
  virtual void singCustomization() ;
};
```

```
class Insect: public CartoonCharacter {
public:       // Общий код для кузнечиков
  virtual void dance () ; //и сверчков
  virtual void sing();
protected:
  virtual void danceCustomization1 () = 0;
  virtual void danceCustomization2 () = 0;
  virtual void singCustomization() = 0;
};
class Grasshopper: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
  virtual void singCustomization() ;
};
class Cricket: public Insect {
protected:
  virtual void danceCustomization1()
  virtual void danceCustomization2();
  virtual void singCustomization();
} ;
```

class CartoonCharacter { ... };



Обращение к именам в шаблонных базовых классах

```
class CompanyA{
public:
  void sendClearText(const std::string& msg);
  void sendEncryptedText(const std::string& msg);
  ...
};

class CompanyB{
public:
  void sendClearText(const std::string& msg);
  void sendEncryptedText(const std::string& msg);
 ...
};
... // классы для других компаний

class MsgInfo {...}; // класс, содержащий информацию,
// используемую для создания сообщения
```

```
template<typename Company>
class MsgSender{
public:   ... // конструктор, деструктор и т. п.
  void sendClear(const MsgInfo& info){
    std::string msg;
    создать msg из info
    Company c;
    c.sendClearText(msg);
  }
  void sendSecret(const MsgInfo& info) {...}
  // аналогично sendClear, но вызывает c.sendEncrypted
};

template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:   ...
  void sendClearMsg(const MsgInfo& info){
    записать в протокол перед отправкой;
    sendClear(info); // вызвать функцию из базового класса
                     // этот код не будет компилироваться!
    записать в протокол после отправки;
  }
```

...

```
class CompanyZ {   // этот класс не представляет функции sendCleartext
public:   …
   void sendEncrypted(const std::string& msg);
};
```

Общий шаблон MsgSender не подходит для CompanyZ, потому что в нем определена функция sendClear, которая для объектов класса CompanyZ не имеет смысла. Чтобы решить эту проблему, мы можем создать специализированную версию MsgSender для CompanyZ:

```
template <> // полная специализация MsgSender
class MsgSender <CompanyZ> {
        //отличается от общего шаблона только отсутствием функции sendCleartext
public:  …
  void sendSecret(const MsgInfo& info){...}
};
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public: ...
  void sendClearMsg(const MsgInfo& info){
     записать в протокол перед отправкой;
     sendClear(info); // если Company == CompanyZ,  то этой функции не существует
     записать в протокол после отправки;
  }
};
```

Варианты решения

1. Можно предварить обращения к функциям из базового класса указателем this:

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info){
  записать в протокол перед отправкой;
  this->sendClear(info); // порядок! Предполагается, что
     // sendClear будет унаследована
  записать в протокол после отправки;
  }
};
```

2. Можно воспользоваться using-объявлением (делает скрытые имена из базового класса видимыми в производном классе).

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;
    // сообщает компилятору о том, что
    // sendClear есть в базовом классе
  void sendClearMsg(const MsgInfo& info){
  ...
  sendClear(info); // нормально, предполагается, что
  ...            // sendClear будет унаследована
  }
};
```

3. Явно указать, что вызываемая функция находится в базовом классе (недостаток: если вызываемая функция виртуальна, то явная квалификация отключает динамическое связывание).

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
pubilc:
  void sendClearMsg(const MsgInfo& info){
  ...
  MsgSender<Company>::sendClear(info);
      // нормально, предполагается, что
  ...  // sendClear будет унаследована
  }
...
};
```

С точки зрения видимости имен, все три подхода эквивалентны: они обещают компилятору, что любая специализация шаблона базового класса будет поддерживать интерфейс, предоставленный общим шаблоном.

Но если данное обещание не будет выполнено, истина всплывет позже.

```
LoggingMsgSender<CompanyZ> zMsgSender;
MsgInfo msgData;
... // поместить info в msgData
zMsgSender.sendClearMsg(msgData); // ошибка! не скомпилируется
```

Разбухание кода в результате применения шаблонов

```
template<typename T, std::size_t n> // шаблон матрицы размерностью n x n,
class SquareMatrix {                // состоящей из объектов типа T;
public:
  ...
  void invert(); // обращение матрицы на месте
};
```

В результате конкретизации шаблона может возникнуть дублирование:

```
SquareMatrix<double, 5> sm1;
...
sm1.invert(); // вызов SquareMatrix<double, 5>::invert()
SquareMatrix<double, 10> sm2;
...
sm2.invert(); // вызов SquareMatrix<double, 10>::invert()
```

```
template<typename T> // базовый класс, не зависящий
class SquareMatrixBase { // от размерности матрицы
protected:
void invert(std::size_t matrixSize);
     // обратить матрицу заданной размерности
   ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
   using SquareMatrixBase<T>::invert;
   // чтобы избежать сокрытия базовой версии invert;
public:
   ...
   void invert() {this->invert(n);} // встроенный вызов параметризованной
                                    // версии invert из базового класса
};
```

```
template<typename T>
class SquareMatrixBase {
protected:  ...
  SquareMatrixBase(std::size_t n, T pMem) :size(n), pData(pMem){}
    // и указатель на данные матрицы сохраняет размерность
  void setData(T *ptr) {pData = ptr;} // присвоить значение pData
private:
  std::size_t size; // размерность матрицы
  T *pData; // указатель на данные матрицы
};
```

Это позволяет производным классам решать, как выделять память.

```
template<typename T, size_t size>
class SquareMatrix: private SquareMatrixBase {
public:  ...
  SquareMatrix() : SquareMatrixBase<t>(n, data) {}
  // передать базовому классу размерность матрицы и указатель на данные
private:
  T data(n*n); 
```

например, прямо в объекте SquareMatrix

- Шаблоны генерируют множество классов и функций, поэтому любой встречающийся в шаблоне код, который не зависит от параметров шаблона, приводит к разбуханию кода.
- Разбухания из-за параметров шаблонов, не являющихся типами, часто можно избежать, заменив параметры шаблонов параметрами функций или данными-членами класса.
- Разбухание из-за параметров-типов можно ограничить, обеспечив общие реализации для случаев, когда шаблон конкретизируется типами с одинаковым двоичным представлением

(например, указательные типы list<int*>, list<const int*>, list<SquareMatrix<long,3>*> и т.п.)