ООП

Наследование

Наследование

- Механизм описание новых(производных) классов на основе уже существующего (базового, родительского)
- Свойства и функциональность родительского класса заимствуются новым классом

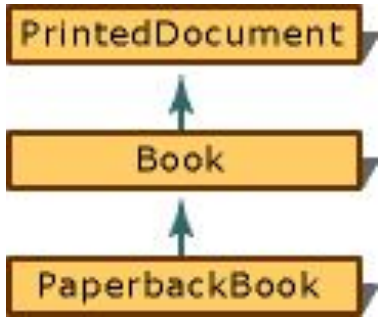
Одиночное наследование

- При одиночном наследовании у каждого производного класса не более одного базового класса

```
class PrintedDocument {};

class Book : public PrintedDocument {};

class PaperbackBook : public Book {};
```



Пример-1

```
class Document {
public:
   char *Name;   // Document name.
   void PrintNameOf();   // Print name.
};

void Document::PrintNameOf() {
   cout << Name << endl;
}

class Book : public Document {
public:
   Book( char *name, long pagecount );
private:
   long  PageCount;
};
```

```
// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
   Name = new char[ strlen( name ) + 1 ];
   strcpy_s( Name, strlen(Name), name );
   PageCount = pagecount;
};

...

Book LibraryBook( "ahkhlwauig", 944 );
LibraryBook.PrintNameOf();
```

Пример-2

```
class Document {
public:
   char *Name;           // Document name.
   void  PrintNameOf() {}  // Print name.
};
Document::Document(char *name){
   Name = new char[ strlen( name ) + 1 ];
   strcpy_s( Name, strlen(Name), name );
}
void Document::PrintNameOf() {
   cout << Name << endl;
}
class Book : public Document {
   Book( char *name, long pagecount );
   long  PageCount;
   public: void PrintNameOf();
};
```

```
Book::Book( char *name, long pagecount ) {
   Name = new char[ strlen( name ) + 1 ];
   strcpy_s( Name, strlen(Name), name );
   PageCount = pagecount;
};

void Book::PrintNameOf() {
   cout << "Name of book: ";
   Document::PrintNameOf();
}
...

Book LibraryBook( "ahkhlwauig", 944 );
LibraryBook.PrintNameOf();
// Name of book: ahkhlwauig
Document LibraryDoc( "sdfsfdas");
LibraryBook.PrintNameOf();
// sdfsfdas
```

Преобразование указателей

- ```
  struct Document {
     char *Name;
     void PrintNameOf() {}
  };

  class PaperbackBook : public Document {};

  int main() {
     Document * DocLib[10];   // Library of ten documents.
     for (int i = 0 ; i < 10 ; i++)
        DocLib[i] = new PaperbackBook;
  }
  ```
  ```

Конструкторы/деструкторы

```
class Document {
public:
   Document();
   ~Document();
};
Document::Document(char *name){
   cout<<”New Doc”;
}
Document::~Document(char *name){
   cout<<”Del Doc”;
}
class Book : public Document {
   Book();
   ~Book();
};
Book::Book() {
    cout<<”New Book”;
 };
```

```
Book::Book() {
    cout<<”Del Book”;
 };
...

{
    Book Alg;
    cout<<”\n”;
}
/*
New Doc
New Book
Del Book
Del Doc
*/
```

Виртуальные функции

- Виртуальная функция — это функция-член, которую предполагается переопределить в производных классах.
- При ссылке на объект производного класса с помощью указателя или ссылки на базовый класс можно вызвать виртуальную функцию для этого объекта и выполнить версию функции производного класса.
- Виртуальные функции являются особыми функциями, потому что при вызове объекта производного класса с помощью указателя или ссылки на него C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта.

Пример

```
class Base {
	public: virtual void who() {
		cout << *Base\n";
	}
};
class first_d: public Base {
	public: void who() {
		cout << "First derivation\n";
	}
};
class seconded: public Base {
	public: void who() {
		cout << "Second derivation\n*";
	}
};
```

```
int main()
	{
	Base base_obj;
	Base *p;
	first_d first_obj;
	second_d second_obj;
	p = &base_obj;
	p->who();
	p = &first_obj;
	p->who();
	p = &second_ob;
	p->who();
	return 0;
	}
//Base
//First derivation
//Second derivation
```

Пример-продолжение

```
class Base {
	public: virtual void who() {
		cout << *Base\n";
	}
};
class first_d: public Base {
	public: void who() {
		cout << "First derivation\n";
	}
};
class seconded: public Base {
	public: void who() {
		cout << "Second derivation\n*";
	}
};
```

```
void show_who (Base &r) {
	r.who();
}
int main()  {
	Base base_obj;
	first_d first_obj;
	second_d second_obj;
	show_who (base_ob j);
	show_who(first_obj);
	show_who(second_obj);
	return 0;
}
//Base
//First derivation
//Second derivation
```

Перекрытие методов

```
struct A{
        void f(int);
};
struct B : A{
        void f(long);
};


B b;
b.f(1);    // f(long) класса B
```

```
struct B : A
{
        using A::f;
        void f(long);
};


B b;
b.f(1);    // f(int) класса A
```

Модификаторы доступа

Public наследование	Private наследование	Protected наследование

Class A
{
private: int x;
public: int y;
protected: int z;
};
Class B
{
private: int x;
public: int y;
protected: int z;
};
Class C
{
private: int x;
public: int y;
protected: int z;
};
	Class A
	{
	private: int x;
	public: int y;
	protected: int z;
	};
	Class B
	{
	private: int x;
	public: int y;
	protected: int z;
	};
	Class C
	{
	private: int x;
	private: int y;
	private: int z;
	};
		Class A
		{
		private: int x;
		public: int y;
		protected: int z;
		};
		Class B
		{
		private: int x;
		public: int y;
		protected: int z;
		};
		Class C
		{
		private: int x;
		private: int y;
		private: int z;
		};

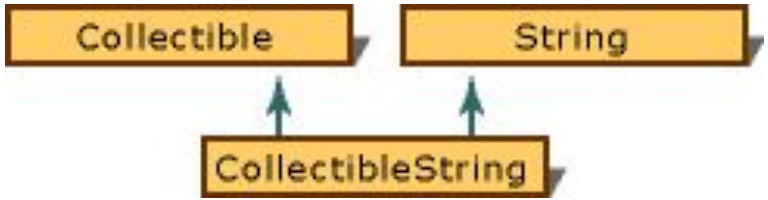
Множественное наследование

- При одиночном наследовании у каждого производного класса не более одного базового класса

```
class Collection {};

class Book {};

class CollectionOfBook : public Book, public Collection {};
```



Множественное наследование

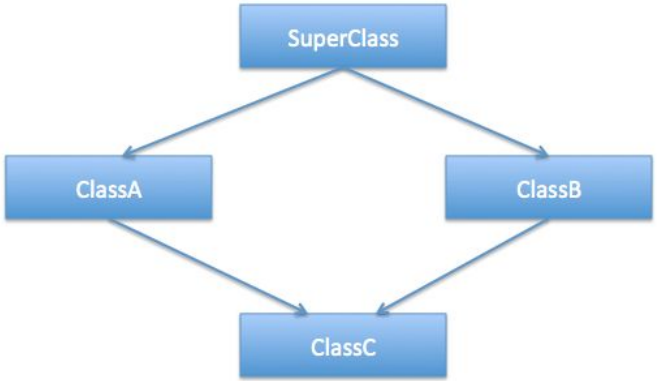
- Порядок, в котором указываются базовые классы, влияет на:
  - Порядок, в котором конструктор выполняет инициализацию. Инициализация выполняется в том порядке, в котором классы указаны в базовом-списке.
  - Порядок, в котором вызываются деструкторы для очистки. Деструкторы вызываются в порядке, обратном указанию классов в базовый-список.

```
class Collection {};
class Book {};
class CollectionOfBook : public Book, public Collection {};
```

Ромбовидное наследование

```
struct A {
    void foo()
    {
        std::cout << "A" << std::endl;
    }
};
struct B1: virtual A{
};
struct B2: virtual A{
};
struct C:B1,B2{
};

int main(){
    C c;
    c.foo();
    return 0;
}
```



SOLID

Принципы SOLID

- Single responsibility principle
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

- Принцип единственной ответственности
- Принцип открытости/закрытости
- Принцип подстановки Лисков
- Принцип разделения интерфейса
- Принцип инверсии зависимостей

SRP

```
public class Book {
    private String name;
    private String author;
    private String text;
    //constructor, getters and setters

}
```

```
public class Book {
    private String name;
    private String author;
    private String text;
    //constructor, getters and setters
    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String
replacementWord){
        return text.replaceAll(word, replacementWord);
    }
    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

SRP

```
public class BadBook {
        //...

        void printTextToConsole(){
        // our code for formatting and printing the text
        }
}
```

```
public class BookPrinter {
        // methods for outputting text
        void printTextToConsole(String text){
        //our code for formatting and printing the text
        }

        void printTextToAnotherMedium(String text){
        // code for writing to any other location..
        }
}
```

Принципы SOLID

- ~~Single responsibility principle~~
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

- ~~Принцип единственной ответственности~~
- Принцип открытости/закрытости
- Принцип подстановки Лисков
- Принцип разделения интерфейса
- Принцип инверсии зависимостей

OCP

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

```
public class SuperCoolGuitarWithFlames extends
Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Принципы SOLID

- ~~Single responsibility principle~~
- ~~Open–closed principle~~
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

- ~~Принцип единственной ответственности~~
- ~~Принцип открытости/закрытости~~
- Принцип подстановки Лисков
- Принцип разделения интерфейса
- Принцип инверсии зависимостей

LSP

```
public interface Car {
        void turnOnEngine();
        void accelerate();
}
```

```
public class MotorCar implements Car {
        private Engine engine;
                //Constructors, getters + setters
        public void turnOnEngine() {
                //turn on the engine!
                engine.on();
        }
        public void accelerate() {
                //move forward!
                engine.powerOn(1000);
        }
}
```

LSP

```
public class MotorCar implements Car {
        private Engine engine;
                //Constructors, getters + setters
        public void turnOnEngine() {
                //turn on the engine!
                engine.on();
        }
        public void accelerate() {
                //move forward!
                engine.powerOn(1000);
        }
}
```

```
public class ElectricCar implements Car {

        public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
        }

        public void accelerate() {
        //this acceleration is crazy!
        }
}
```

Принципы SOLID

- ~~Single responsibility principle~~
- ~~Open–closed principle~~
- ~~Liskov substitution principle~~
- Interface segregation principle
- Dependency inversion principle

- ~~Принцип единственной ответственности~~
- ~~Принцип открытости/закрытости~~
- ~~Принцип подстановки Лисков~~
- Принцип разделения интерфейса
- Принцип инверсии зависимостей

ISP

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
public interface BearCleaner {
    void washTheBear();
}
public interface BearFeeder {
    void feedTheBear();
}
public interface BearPetter {
    void petTheBear();
}
```

```
public class BearCarer implements BearCleaner,
BearFeeder {
    public void washTheBear() {
        //I think we missed a spot...
    }
    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

Принципы SOLID

- ~~Single responsibility principle~~
- ~~Open–closed principle~~
- ~~Liskov substitution principle~~
- ~~Interface segregation principle~~
- Dependency inversion principle

- ~~Принцип единственной ответственности~~
- ~~Принцип открытости/закрытости~~
- ~~Принцип подстановки Лисков~~
- ~~Принцип разделения интерфейса~~
- Принцип инверсии зависимостей

DIP

```
public class Windows98Machine {
    private final StandardKeyboard keyboard;
    private final Monitor monitor;
    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

```
public interface Keyboard { }
public class Windows98Machine{
    private final Keyboard keyboard;
    private final Monitor monitor;
    public Windows98Machine(Keyboard keyboard,
Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}

public class StandardKeyboard implements Keyboard { }
```

Ссылки

http://www.c-cpp.ru/books/konstruktory-i-destruktory-proizvodnyh-klassov
https://msdn.microsoft.com/ru-ru/library/a48h1tew.aspx
https://medium.freecodecamp.org/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbbec