Наследование и шаблоны

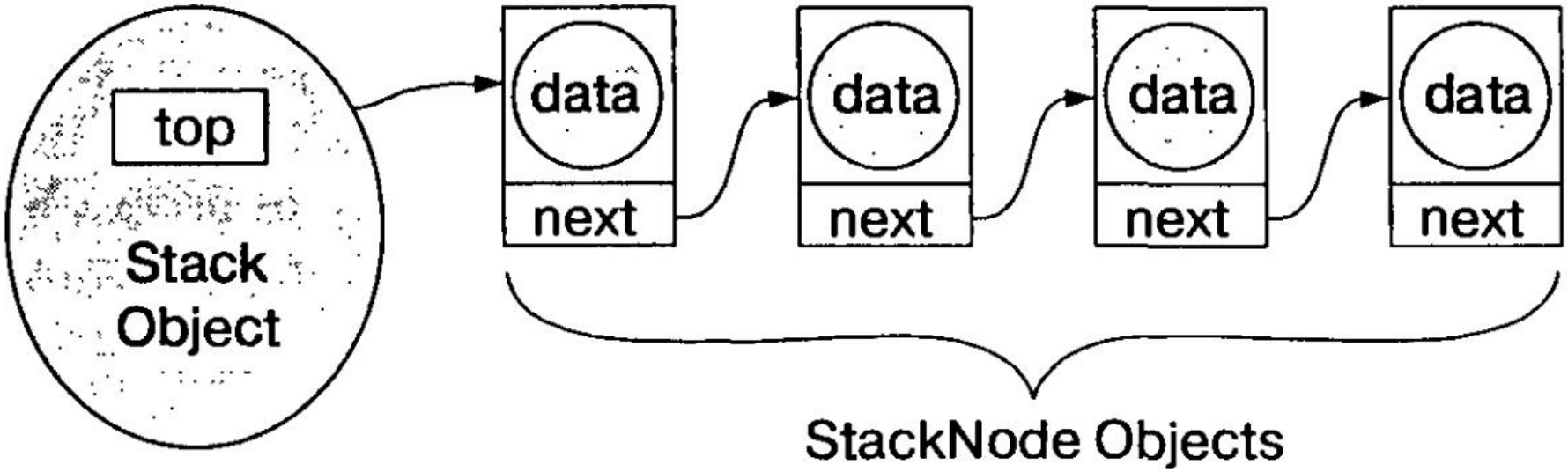
Рассмотрим две задачи проектирования:
1. Создать классы для представления стеков.

2. Создать классы для описания кошек.

(Каждая порода кошек незначительно отличается от остальных. Подобно любым объектам, «кошек» в программе можно создавать и удалять. Помимо этого о кошках можно сказать, что они только едят и спят. Однако каждая порода ест и спит присущим только ей неподражаемым способом.)

Вопрос: влияет ли тип T на поведение

```
class Stack {
public:
  Stack();
  -Stack();
  void push (const T& object) ;
  T pop();
  bool empty() const; // Пуст ли стек?
private:
  struct StackNode { // Узел связного списка.
    T data; // Данные в этом узле.
    StackNode *next; // Следующий узел в списке.
    StackNode(const T& newData, StackNode *nextNode) : data(newData), next(nextNode)
{}
    // Конструктор StackNode инициализирует оба поля.
    };
  StackNode *top; // Вершина стека.
  Stack(const Stack& rhs); // Запретить копирование
  Stacks operator=(const Stack& rhs); // и присваивание
}
```


top
Stack
Object
data
next
data
next
data
next
data
next
StackNode Objects

```
Stack::Stack(): top(0) {} // Инициализация вершины значением null,
void Stack::push(const T& object) {
    top = new StackNode (object, top); // Добавить новый узел в начале списка
}
T Stack::pop() {
   StackNode *topOfStack = top; // Запомнить верхний узел.
   top = top->next;
   T data = topOfStack->data; // Запомнить данные узла.
   delete topOfStack;
   return data;
}
Stack::~Stack(){
   while (top) {
      StackNode *toDie = top; // Получить указатель на вершину,
      top = top->next; // Перейти к следующему узлу,
      delete toDie; // Удалить предыдущую вершину.
   }
}
bool Stack::empty() const { return top ==0; }
```

```
template<class T>
class Stack {
//В точности то же
} ;
```

```
class Cat {
public:
  virtual ~Cat();
  virtual void eat () = 0; // Все кошки едят.
  virtual void sleep () = 0; // Все кошки спят.
}

class Siamese: public Cat {
public:
  void eat();
  void sleep();
} ;
class BritishShortHairedTabby: public Cat {
public:
  void eat();
  void sleep();
} ;

class Stack {  // Стек чего угодно.
public:
  virtual void push (const ??? object) = 0;
  virtual ??? pop() = 0;
                                                    …
                                                    }
```

- Шаблоны должны быть использованы для генерации семейств классов, тип объектов которых не влияет на поведение функций этих классов.

- Наследование следует использовать для создания семейств классов, тип объектов которых влияет на поведение функций создаваемых классов.

Композиция как моделирование отношения «часть» (“part of”)

Композиция (агрегирование, включение, вложение) – отношение между типами, которое возникает тогда, когда объект одного типа содержит в себе объекты других типов. Это простейший механизм для создания нового класса путем **объединения** нескольких объектов существующих классов в единое целое.

При агрегировании между классами действует «**отношение принадлежности**»

- У машины есть кузов, колеса и двигатель
- У человека есть голова, руки, ноги и тело
- У треугольника есть вершины

Вложенные объекты обычно объявляются закрытыми (private) внутри класса-агрегата.

```
class Address {...}; // адрес проживания
class PhoneNumber {...};
class Person {
public:
...
private:
  std::string name; // вложенный объект
  Address address; // то же
  PhoneNumber voiceNumber; // то же
  PhoneNumber faxNumber; // то же
};
```

template<typename T> // *неправильный* способ использования

class Set: public std::list<T> {...}; // list для определения Set

template<typename T> // правильный способ использования list

class Set { // для определения Set

public:

  bool member(const T& item) const;

  void insert(const T& item);

  void remove(const T& item);

  int cardinality() const;

private:

  std::list<T> rep; // представление множества

};

```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin(), rel.end(), item) != rep.end();
}
template<typename T>
void Set<T>::insert(const T& item)
{
  if(!member(item)) rep.push_back(item);
}
template<typename T>
void Set<t>::remove(const T& item)
{
  typename std::list<T>::iterator it =
    std::find(rep.begin(), rep.end(), item);
  if(it != rep.end()) rep.erase(it);
}
template<typename T>
int Set<T>:: cardinality() const
{
return rep.size();
}
```

Закрытое наследование

class Person {...}

class Student: private Person {...} // теперь наследование закрытое

void eat(const Person& p); // все люди могут есть

void study(const Student& s); // только студенты учатся

Person p; // p – человек (Person)

Student s; // s – студент (Student)

eat(p); // нормально, p – типа Person

eat(s); // ошибка! Student не является объектом Person

```
class Timer {
public:
    explicit Timer(int tickFrequency);
    virtual void onTick() const;
  //автоматически вызывается при каждом тике
...
};
class Widget: private Timer {
private:
  virtual void onTick() const;
  // просмотр данных об использовании Widget и т. п.
};
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
    virtual void onTick() const;
    ...
  };
  WidgetTimer timer;
  ...
};
```

Пустые базовые классы

```
class Empty {};
// не имеет данных, поэтому объекты
// не должны занимать памяти
class HoldsAnInt{ //память, по идее, нужна только для int
private:
  int x;
  Empty e; // не должен занимать память
};
// sizeof(HoldsAnInt) > sizeof(int);

class HoldsAnInt: private Empty{
private:
  int x;
}; // sizeof(HoldsAnInt) = sizeof(int);
```

Запрет генерации методов класса

```
class HomeForSale {...};
HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1); // попытка скопировать h1
  –
// не должно компилироваться!
h1 = h2; // попытка скопировать h2 –
// не должно компилироваться!
```

```
class HomeForSale {
public:
   ...
private:
   HomeForSale(const HomeForSale&); // только объявления
   HomeForSale& operator=( const HomeForSale&);
};
```

```
class Uncopyable{
protected:
  Uncopyable() {} // разрешить конструирование
  ~Uncopyable() {} // и уничтожение
          // объектов производных классов
private:
  Uncopyable(const Uncopyable&); // но предотвратить копирование
  Uncopyable& operator=(const Uncopyable&);
};
class HomeForSale : private Uncopyable {
... }; // в этом класс больше нет ни конструктора копирования, ни оператора присваивания
```

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)

Модификатор наследования

Модификатор доступа	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	нет доступа	нет доступа	нет доступа

Защищённое наследование

Защищенное наследование – наследование реализации, доступной для последующего наследования

- При защищенном наследовании открытые поля и методы родительского класса становятся защищенными полями и методами производного
- Данные методы могут использоваться классами, порожденными от производного
- Как и в случае закрытого наследования, порожденный класс должен предоставить собственный интерфейс
- Разницу между защищенным и закрытым наследованием почувствуют лишь наследники производного класса

```
class CIntArray
{
public:
	int operator[](int index)const;
	int& operator[](int index);
	int GetLength()const;
	void InsertItem(int index, int value);
};
class CIntStack : protected CIntArray
{
public:
	void Push(int element);
	int Pop()const;
	bool IsEmpty()const;
};
class CIntStackEx : public CIntStack
{
public:
	int GetNumberOfElements()const;
// использует GetLength()
};
```

Наиболее важные соответствия отношений конструкциям C++

1. **Наличие общего базового класса означает наличие общих свойств.** Если класс D1 и класс D2 объявляют класс B своим базовым классом, то D1 и D2 наследуют общие элементы данных и/или общие функции-члены B.
2. **Открытое наследование означает «есть разновидность».** Если класс D открыто наследует от класса B, то каждый объект типа D также является объектом типа B, но не наоборот.
3. **Закрытое наследование означает «реализацию посредством».** Если класс D закрыто наследует от класса B, объекты типа D достаточно просто реализуются с помощью объектов типа B; между объектами типов B и D нет концептуальной взаимосвязи.
4. **Вложение означает «содержит» или «реализуется посредством».** Если класс A содержит элементы данных типа B, то объекты типа A либо имеют компонент типа B, либо реализуются посредством объектов типа B.