C++
Лекция 5
Перегрузка операторов
Шаблоны

- `IntSet` и `SortedIntSet`
- Перегрузка операторов
- Шаблоны
- Принцип подстановки Лисков

Перегрузка операторов

- В философии C++ пользовательские типы данных должны иметь такие же возможности как и встроенные.
- То есть можно использовать *операторы* (такие как +, -, [ ], и т. д.) для собственных типов данных!
- Для этого необходимо рассказать компилятору что мы хотим получить от оператора.
  - Данный механизм называется ***перегрузкой операторов***.

Перегрузка функций

- Несколько функций с одним именем, но разными **сигнатурами**.
  - Сигнатура включает в себя имя и входные параметры.

Что видим мы

```
class Person {
public:
  void greet() {...}
  void greet(int x) {...}
  void greet(string x) {...}
  void greet(int x,
             string x) {...}
  bool greet() {...}
};
```

Что видит компилятор

```
class Person {
public:
  void greet() {...}
  void greet_int(int x) {...}
  void greet_string(string x)
{...}
  void greet_int_string(int x,
                        string x)
{...}
  bool greet() {...}
};
```

Ошибка, дублирование.

Перегрузка операторов

- Представьте оператор как функцию.

```
x + y                    operator+(x,y)
```

- Перегрузка работает точно так же как с функциями.

Что видим мы

```
int main() {
  int x = 3;
  int y = 3;
  IntSet s1;
  IntSet s2;

  int z = x + y;
  IntSet s3 = s1 + s2;

}
```

Что видит компилятор (часть 1)

```
int main() {
  int z = operator+(x, y);
  IntSet s3 = operator+(s1, s2);
}
```

Что видит компилятор (часть 2)

```
int main() {
  int z = operator+_int_int(x, y);
  IntSet s3 =
    operator+_IntSet_IntSet(s1, s2);
}
```

Перегрузка операторов

- Пример перегрузки оператора вывода:

```
class Card {
...
};

std::ostream & operator<<(std::ostream &os,
                          const Card &card) {
  // Implementation here
}
```

operator<< означает перегрузку оператора вывода.

- Пример использования в коде:

```
Card c = _____;
cout << c << endl;
```

Оператор вывода

- Некоторые операторы используют функции не являющиеся членами класса (**non-member** функции)
- Если реализовать оператор << для наших классов описывающих множества, то можно использовать cout для вывода объектов множества.
  - Для реализации перегрузки оператора используется функция с именем `operator<<`, функция не является членом класса.

Оператор вывода

```
class IntSet {
  ...
};

std::ostream &operator<<(
  std::ostream &os,
  const IntSet &s);
```

IntSet.h

```
ostream &operator<<(
  ostream &os,
  const IntSet &s) {

  s.print(os);
  return os;
}
```

IntSet.cpp

Передаем объект, так как функция не член класса

```
int main() {
  ...
  cout << set << endl;
}
```

Оператор индексирования

- Для перегрузки некоторых операторов используются функции являющиеся методами класса.
- Реализуем перегрузку оператора [ ] для проверки элемента.
  - Функция `operator[]` - является методом класса.

Оператор индексирования

```
class IntSet {
public:
  ...
  bool operator[](int v)
    const;
};
```

IntSet.h

```
bool IntSet::operator[](int v) const {
  return contains(v);
}
```

IntSet.cpp

```
int main() {
  ...
  set.insert(32);

  cout << "Contains 32? " << set[32] << endl;
}
```

- Идея: Скопипастить класс `IntSet`, затем изменить `int` на `char` везде.

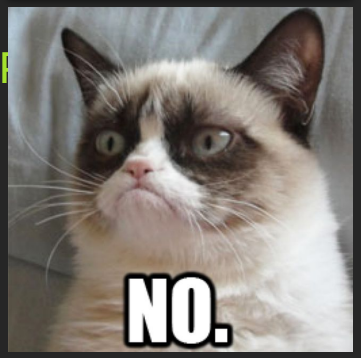
```
class IntSet {
public:
  ...
  void insert(int v);
  void remove(int v);
  bool contains(int v) const;
  int size() const;
  ...

private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
class CharSet {
public:
  ...
  void insert(char v);
  void remove(char v);
  bool contains(char v) const;
  int size() const;
  ...

private:
  char elts[ELTS_CA
  int elts_size;
  ...
};
```

Хороший подход?



1 Очень плохая идея.

- Идея получше: написать код таким образом чтобы компилятор сам копипастил код за нас.

```
class IntSet {
public:
  ...
  void insert(int v);
  void remove(int v);
  bool contains(int v) const;
  int size() const;
  ...

private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
template <typename T>
class UnsortedSet {
public:
  ...
  void insert(T v);
  void remove(T v);
  bool contains(T v) const;
  int size() const;
  ...

private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

Шаблоны

- Шаблоны позволяют создавать многократно используемый код.
- Используя шаблоны, можно создавать *обобщенные* функции или классы.
- Тип данных задается как **параметр**, поэтому не нужно явным образом создавать реализации для каждого типа данных.

Обобщенная функция - функция перегружающая сама себя.

```
template <typename T>
class UnsortedSet {
  ... // T используется в коде
};
```

T может быть любым типом данных

Использование шаблонов

- Компилятор вместо **T** нужные тип данных и генерирует для него код.

T=int

T=char

T=Duck

```
int main() {
  UnsortedSet<int> is;
  is.insert(3);
  is.insert(7);
  is.insert(8);
  cout << is; // { 3, 7, 8 }

  UnsortedSet<char> cs;
  cs.insert('a');
  cs.insert('e');
  cs.insert('i');
  cout << is; // { a, e, i }

  UnsortedSet<Duck> ds;
}
```

Использование шаблонов

UnsortedSet.h

```
template <typename T>
class UnsortedSet {
public:
  ...
  void insert(T v);
  void remove(T v);
private:
  T elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

The compiler instantiates the template as needed according to how it is used in the code.

```
#include "UnsortedSet.h"
int main() {
  UnsortedSet<int> is;
  UnsortedSet<Duck> ds;
}
```

```
class UnsortedSet<int> {
public:
  ...
  void insert(int v);
  void remove(int v);
private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

```
class UnsortedSet<Duck> {
public:
  ...
  void insert(Duck v);
  void remove(Duck v);
private:
  Duck elts[ELTS_CAPACITY];
  int elts_size;
  ...
};
```

Шаблонные функции

- A **function template** can be instantiated to make versions to work with different types of inputs.

```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {
  int i = max(3, 10);
  double d = max(3.14, 3.33);

  Card c1(Card::RANK_ACE, Card::SUIT_CLUBS);
  Card c2(Card::RANK_TEN, Card::SUIT_HEARTS);
  Card best_card = max(c1, c2);
}
```

The compiler is able to **deduce** which version of `max` we want in each case from the argument types.

Компиляция шаблонов

- Параметр шаблона потенциально может быть любым типом данных, но...

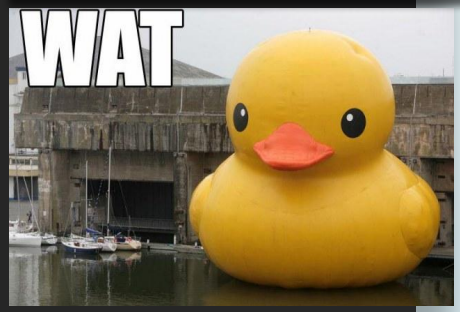
```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {

  Duck d1("Donald");
  Duck d2("Scrooge");
  Duck best_duck = max(d1, d2);
  cout << best_duck.getName() << " wins!" << endl;

}
```

Не скомпилируется.



Компиляция шаблонов

1. Создание версии под данных тип.
2. Проверка что компиляция пройдет успешно.

реализация для T = Duck

```
T max(Duck val1, Duck val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}

int main() {

  Duck d1("Donald");
  Duck d2("Scrooge");
  Duck best_duck = max(d1, d2);
  cout << best_duck.getName() << " wins!" << endl;
}
```

Ошибка: оператор > не определен для типа Duck, Duck.

Компиляция шаблонов

- Инстанцирование шаблонов происходит на этапе компиляции, до этапа линковки.
- Это значит определения всех функций должно находится в одном сегменте компиляции.
- Поэтому нельзя разделить объявление и реализацию в .h и .cpp.

library.h

```
template <typename T>
T max(T val1, T val2);
```

library.cpp

```
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}
```

main.cpp

```
#include "library.h"
int main() {
  int i = max(3, 10);
}
```

Ошибка

```
g++ main.cpp library.cpp
(Ошибка линковки)
```

Компиляция шаблонов

- Объявление и реализация должны быть в `.h` файле.

library.h

```
template <typename T>
T max(T val1, T val2);
...
template <typename T>
T max(T val1, T val2) {
  if (val1 > val2) { return val1; }
  else { return val2; }
}
```

Объявления в начале файла, реализация в конце.

main.cpp

```
#include "library.h"
int main() {
  int i = max(3, 10);
}
```

OK.

```
g++ main.cpp
(Этап линковки)
```

Шаблоны - методы класса

```
template <typename T>
class UnsortedSet {
public:
  void insert(T v);
...
};

template <typename T>
void UnsortedSet<T>::insert(T v) {
  assert(size() < ELTS_CAPACITY);
  if (contains(v)) {
    return;
  }
  elts[elts_size] = v;
  ++elts_size;
}
...
```

Говорим о том, что функция член класса UnsortedSet<T>.

Exercise: fillFromArray

- Написать функцию, которая заполнит UnorderedSet<T> элементами типа **T** из массива.

```
template <typename T>
void fillFromArray(          set,          arr, int n) {

  // CODE

}

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Какие здесь параметры?

Solution: `fillFromArray`

- Написать функцию, которая заполнит `UnorderedSet<T>` элементами типа **T** из массива.

```
template <typename T>
void fillFromArray(UnsortedSet<T> &set, const T *arr, int n) {
  for (int i = 0; i < n; ++i) {
    set.insert(arr[i]);
  }
}

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Полиморфизм Статический vs. Динамический

- Полиморфизм - способность принимать множество форм.
- Шаблоны обеспечивают **параметрический полиморфизм**.
  - Параметр шаблона T может быть любым типом `int`, `char`, etc.[1]
- **Полиморфизм подтипов**.
  - Указатель на базовый класс может указывать на любой класс наследник

(использование позднего связывания.)

Статический полиморфизм (на этапе компиляции).

Динамический полиморфизм (во время выполнения).

Возвращаясь к множествам

- Если захотим вместо UnsortedSet использовать SortedSet то код придется изменять во многих местах.

```
template <typename T>
void fillFromArray(UnsortedSet<T> &set, const T *arr,
                   int n);

int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Псевдонимы типов

- Для объявления псевдонимов используется ключевое слово using

```
template <typename T>
using Set = UnsortedSet<T>;

template <typename T>
void fillFromArray(Set<T> &set, const T *arr,
                   int n);

int main() {
  Set<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fillFromArray(set1, arr1, 4);

  Set<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fillFromArray(set2, arr2, 3);
}
```

Теперь изменения придется делать только в одном месте

Подтипы vs. Производные типы

- Не всегда наследуемые (производные) типы данных являются подтипами
  - Подтипы должны иметь отношение “род - вид” к базовому классу
- Что такое отношение “род - вид”?
  - Принцип подстановки Барбары Лисков

Принцип подстановки Лисков

- Если S подтип T…
  - Любое свойство T является свойством S.
  - Любой код зависящий от интерфейса T, можно подставить подтип S и это не вызовет никаких проблем.

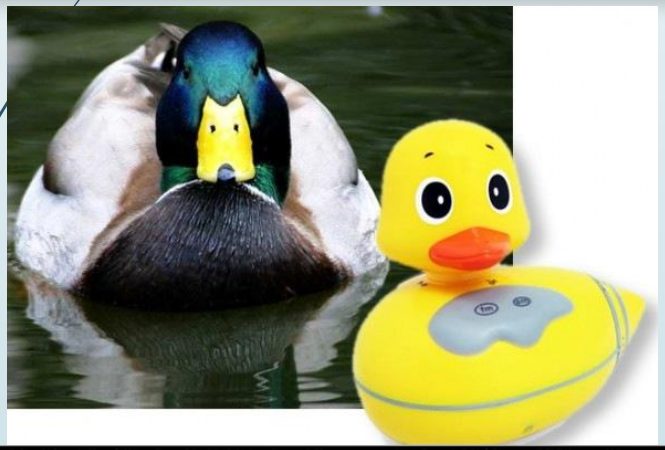
Barbara Liskov, MIT

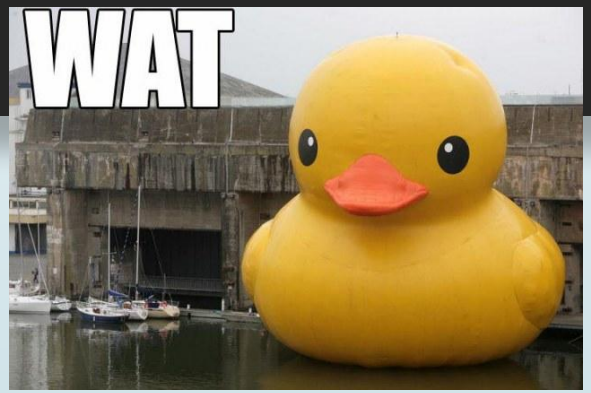
Наследующий класс должен дополнять, а не замещать поведение базового класса

И снова утки

```
class Duck : Bird {
  ...
  // Результат:  вывести “ква”
  virtual void talk() {
    cout << "ква" << endl;
  }
};
```

```
class ToyDuck : public Duck {
  ...
  // Результат:  вывести “ква” если
  //           batteryLevel >= 10
  virtual void talk() {
    if (batteryLevel >= 10) {
      cout << "quack" << endl;
      --batteryLevel;
    }
  }
};
```





LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction