Обобщенное программирование

Виды полиморфизма в C++

- Ложный статический –
  a. Ad-hoc-полиморфизм – перегрузка функций и методов
- Истинный статический – не оказывает влияния на работу функции во время выполнения, но реализуется во время компиляции
  a. Параметрический полиморфизм - шаблоны(template), auto, C-Style, (void*)
- Истинный динамический – определение типов и их поведения во время выполнения.
  a. Динамическая диспетчеризация – Полиморфизм подтипов: наследование
  b. Enum диспетчеризация – характеризуется передачей тэга (дискриминант), и юниона возможных типов.
- Специальный полиморфизм: функторы и лямбда-выражения

https://habr.com/ru/articles/718888/

Обобщенное программирование

- Парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.
- Вместо описания отдельного типа применяется описание семейства типов, имеющих общий интерфейс и *семантическое поведение* (semantic behavior).
- *Концепцией* (concept) – Набор требований, описывающий интерфейс и семантическое поведение.
- Тип *моделирует* концепцию (является моделью концепции), если он удовлетворяет её требованиям.
- Концепция является уточнением другой концепции, если она дополняет последнюю.

Требования к концепциям

- *Допустимые выражения* (valid expressions) — выражения языка программирования, которые должны успешно компилироваться для типов, моделирующих концепцию.
- *Ассоциированные типы* (associated types) — вспомогательные типы, имеющие некоторое отношение к моделирующему концепцию типу.
- *Инварианты* (invariants) — такие характеристики типов, которые должны быть постоянно верны во время исполнения. Обычно выражаются в виде предусловий и постусловий. Невыполнение предусловия влечёт непредсказуемость соответствующей операции и может привести к ошибкам.
- *Гарантии сложности* (complexity guarantees) — максимальное время выполнения допустимого выражения или максимальные требования к различным ресурсам в ходе выполнения этого выражения.

Шаблоны

Шаблоны функций

- «Write Everything Twice»(**WET**)(alt.«We enjoy typing») vs «Don’t repeat yourself» (**DRY**)

```
int _min(int a, int b){
    if( a < b){
        return a;
    }
    return b;
}

double _min(double a, double b){
    if( a < b){
        return a;
    }
    return b;
}
```

```
template<class Type>
Type _min(Type a, Type b){
    if( a < b){
        return a;
    }
    return b;
}
```

Идея

- Шаблоны позволяют создавать параметризованные классы и функции.

```
#include <iostream>
template<class Type>
Type _min(Type a, Type b) {
    if (a < b) {
        return a;
    }
    return b;
}
int main(int argc, char** argv) {
    std::cout << _min(1, 2) << std::endl;
    std::cout << _min(3.1, 1.2) << std::endl;
    std::cout << _min(5, 2.1) << std::endl; //
oops!
    return 0;
}
```

```
#include <iostream>
template<class Type>
Type _min(Type a, Type b) {
    if (a < b) {
        return a;
    }
    return b;
}
int main(int argc, char** argv) {
    std::cout << _min<double>(5, 2.1) << std::endl;
    return 0;
}
```

- Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка).

Перегрузка и специализация

```
template<class Type>
Type* _min(Type* a, Type* b){
    if(*a < *b){
        return a;
    }
    return b;
}
```

```
template<>
std::string _min(std::string a, std::string b){
    if(a.size() < b.size()){
        return a;
    }
    return b;
}
```

https://habr.com/ru/articles/436880/

Шаблоны классов и структур

```
template<typename Type>
class Interval
{
public:
      Interval(Type inStart, Type inEnd)
            : start(inStart), end(inEnd){
  }
      Type getStart() const{
    return start;
  }
      Type getEnd() const{
    return end;
  }
      Type getSize() const{
    return (end - start);
  }
```

```
 // Метод для получения интервала пересечения данного интервала с другим
      Interval<Type> intersection(const Interval<Type>& inOther) const
  {
    return Interval<Type>{
        max(start, inOther.start),
        min(end, inOther.end)
    };
  }

private:
  Type start;
  Type end;
};
```

https://habr.com/ru/articles/599801/

Шаблоны классов и структур

```
template<typename Type>
class SimpleArray
{
public:
SimpleArray(int inElementsNum)
            : elements(new Type[inElementsNum]),
num(inElementsNum)
      {
      }

      int getNum() const
      {
            return num;
      }
```

https://habr.com/ru/articles/599801/

```
Type getElement(int inIndex) const
      {
            return elements[inIndex];
      }
 /void setElement(int inIndex, Type inValue)
      {
            elements[inIndex] = inValue;
      }

      ~SimpleArray()
      {
            delete[] elements;
  }

private:
      Type* elements = nullptr;
      int num = 0;
};
```

Шаблоны классов и структур

```
int main()
{
	SimpleArray<int> simpleArray{ 4 };

	simpleArray.setElement(0, 1);
	simpleArray.setElement(1, 2);
	simpleArray.setElement(2, 3);
	simpleArray.setElement(3, 4);

	int sum = 0;
	for (int index = 0; index < simpleArray.getNum(); ++index)
		sum += simpleArray.getElement(index);

	return 0;
}
```

```
int main()
{
	SimpleArray<bool> simpleBoolArray{ 4 };

	simpleArray.setElement(0, true);
	simpleArray.setElement(1, false);
	simpleArray.setElement(2, false);
	simpleArray.setElement(3, true);

	return 0;
}
```

https://habr.com/ru/articles/599801/

Специализация SimpleArray

```
template<typename Type>
class SimpleArray
{
      //...
};

struct BitArrayAccessData
{
  int byteIndex = 0;
  int bitIndexInByte = 0;
};

template<>
class SimpleArray<bool>
{
public:
~SimpleArray(){
      delete[] elementsMemory;
  }
```

```
SimpleArray(int inElementsNum): elementsMemory(nullptr),num(inElementsNum){
      const int lastIndex = (inElementsNum - 1);
      const BitArrayAccessData lastElementAccessData =
                              getAccessData(lastIndex);
      const int neededBytesNum = lastElementAccessData.byteIndex + 1;
      elementsMemory = new unsigned char[neededBytesNum];
}

int getNum() const {return num;}

bool getElement(int inIndex) const{
      const BitArrayAccessData accessData = getAccessData(inIndex);
      const unsigned char elementMask = (1 << accessData.bitIndexInByte);
      return elementsMemory[accessData.byteIndex] & elementMask;
}
void setElement(int inIndex, bool inValue) const{
      const BitArrayAccessData accessData = getAccessData(inIndex);
      const unsigned char elementMask = (1 <<accessData.bitIndexInByte);
      elementsMemory[accessData.byteIndex] =
                  (elementsMemory[accessData.byteIndex] & ~elementMask) |
                  (inValue ? elementMask : 0);
      }
```

https://habr.com/ru/articles/599801/

Специализация SimpleArray<bool>-1

```
template<typename Type>
class SimpleArray
{
      //...
};

struct BitArrayAccessData
{
  int byteIndex = 0;
  int bitIndexInByte = 0;
};

template<>
class SimpleArray<bool>
{
public:
~SimpleArray(){
      delete[] elementsMemory;
  }
```

```
SimpleArray(int inElementsNum): elementsMemory(nullptr),num(inElementsNum){
      const int lastIndex = (inElementsNum - 1);
      const BitArrayAccessData lastElementAccessData =
                              getAccessData(lastIndex);
      const int neededBytesNum = lastElementAccessData.byteIndex + 1;
      elementsMemory = new unsigned char[neededBytesNum];
}

int getNum() const {return num;}

bool getElement(int inIndex) const{
      const BitArrayAccessData accessData = getAccessData(inIndex);
      const unsigned char elementMask = (1 << accessData.bitIndexInByte);
      return elementsMemory[accessData.byteIndex] & elementMask;
}
void setElement(int inIndex, bool inValue) const{
      const BitArrayAccessData accessData = getAccessData(inIndex);
      const unsigned char elementMask = (1 <<accessData.bitIndexInByte);
      elementsMemory[accessData.byteIndex] =
                  (elementsMemory[accessData.byteIndex] & ~elementMask) |
                  (inValue ? elementMask : 0);
      }
```

https://habr.com/ru/articles/599801/

Специализация SimpleArray<bool>-2

```
//...
private:

static BitArrayAccessData getAccessData(int inElementIndex)
  {
      BitArrayAccessData result;
      result.byteIndex = inElementIndex / 8;
      result.bitIndexInByte = inElementIndex - result.byteIndex * 8;
      return result;
  }

  unsigned char* elementsMemory = nullptr;
  int num = 0;
};
```

https://habr.com/ru/articles/599801/

Поговорить

- Параметры – константы
- Количество параметров
- Частичные специализации
- Шаблонные специализации

Функторы