Шаблони(Templates)

Нека имаме един клас IntVector, чиято функционалност е да съхранява динамично цели числа. Той си има имплементирани функции за достъп, за добавяне на елемент, за премахване на елемент и т.н. Него можем да го използваме, за да улесним съхранението на цели числа в контейнер динамично. Какво обаче се случва, ако се наложи да имаме подобен контейнер, който да съхранява дробни числа. Не е проблем, ще напишем клас DoubleVector, аналогичен на IntVector, само ще променим типа данни, който съхранява. ОК, но ако се случи, че в последствие ни бъдат нужни контейнери за булеви променливи, за char масиви или за обекти на някакъв създаден от нас клас. Тогава писането на толкова много почти еднакъв код идва малко в повечко,. И като решение на този проблем C++ предостява шаблоните.

Шаблоните в C++ са средство на езика, което ни позволява да създаваме т.нар. generic функции и класове (това е терминът на английски, няма адекватен превод на български, може би *обобщени*, но не е много интуитивен поне за мен). Това ще рече функции и класове, които могат да изпълняват сходни функционалности с различни типове данни, както примитивни, така и дефинирани от потребителя. Можем да подадем в последствие типът данни, с които искаме някой шаблонен клас или функция да работи като техен параметър. И по този начин ще можем да имаме един шаблонен клас Vector, който ще може да работи с различни типове данни.

Синтаксис:

- Декларират се с ключовата дума `template` . При нейното използване се създава шаблон, който описва какво прави някаква функция или клас, като оставя на компилатора по време на изпълнение да го направи да работи с посочен тип данни.
- Следва списък с шаблонен/нни параметър/ри (template parametr/s) ограден с `< >` (примерно `template <typename T>` или ако искаме повече шаблонни параметри `template <typename T, typename S>`), които служат като маркер, за да се обозначат местата в кода, на които ще се заменят с подадените типове данни.
- Ключовата дума `typename` може да се замени с `class`. И двете са взаимно заменяеми в тази ситуация, като появата на `class` е по-ранна, но за да не става объркване с дефинирането на класове в C++ се създава и `typename`. И двете декларират хипотетичен тип данна.
- Следва декларацията на функцията или класа.

Извикване:

- Шаблонната функция или клас се извикват, като се подават на името им и `< >` , запълнени с желаните типове данни за работа.

Примерите са надолу във файла.

Шаблонни функции

Шаблонните функции дефинират множество от операции, които могат да бъдат приложени на различни типове данни.

Пример:

```
#include <iostream>
#include <ComplexNumber.h>
using namespace std;
```

```
template <typename T>
T _max(T a, T b){
    if(a >= b){
        return a;
    }
    return b;
}

int main(){
    cout << _max<int>(5,4) << " "
         << _max<double>(5.5, 6.6) << " "
         << _max(5.5, a) << " "
         << _max<ComplexNumber>(ComplexNumber(3,5), ComplexNumber(7,6)) << "\n";
    return 0;
}
```

**Заб: При шаблонните функции е възможно пропускане на списъка с шаблонните параметри `<double>`, понеже компилатора рапознава подадените параметри. Това не е добра практика обаче.

**Заб: Може да обърнете внимание, че за да работи следния код трябва за класа ComplexNumber да са предефинирани операторите `>=` и `<<`.

Какво се случва всъщност? - По време на изпълнение на main функцията, когато компилаторът вижда, че на шаблонната функция е подаден *int*, *double* или *ComplexNumber*, той заменя нейния код и на всяко място където е означен маркерът *T* го заменя с *int*, *double* или *ComplexNumber* съответно.

Пример (шаблонна функция с два шаблонни параметъра):

```
#include <iostream>
#include <cstring>
using namespace std;

template <typename T, typename S>
void print_value(T name, S value){
    cout  << "| " << first << " | " << second << " |\n";
}

int main(){
    Person first("gosho");
    print_value<Person, int>("gosho", 50);
    print_value<const char *, double>("pesho", 43.345345);
    return 0;
}
```

Шаблонни класове

Шаблонните класове не бива да се възприемат като истински класове. Подобно на функциите, те представляват само описания, по които компилаторът създава фактическите класове. Тоест ако `Calculator` е шаблон на клас, при срещане в main функцията на `Calculator<int>` той заменя шаблонния параметър *T* на класа с подадения *int* и генерира кодът на клас `Calculator` с член-данни *int*.

Броят на шаблонните параметри на шаблонен клас е произволен. Параметрите могат да участват на произволни места в дефиницията на шаблона. Възможно някои или всички параметри на шаблона да са подразбиращи се (примерно `template <typename T = int>`).

Дефинирането на методите на шаблонен клас се осъществява по два начина:

- като inline функции - имплементацията е тривиална
- извън тялото на класа - дефиницията на всяка функция се предшества от statement-а

  `template <списъкът_с_шаблонни_параметри_на_класа>`, като обаче ако имаме предефиниран шаблонен параметър както примера по-долу не го обозначаваме преди дефиницията на метода.

Важно е да се обърне внимание:

- Дефиницията и декларацията на методите на шаблона не могат да бъдат на различни места, както досега беше с нормалните класове (.h и .cpp файлове). Най-добре е да бъдат в един хедър файл. Причината е, че чак при изпълнение на програмата се генерира кодът, който опсва шаблонът с конкретни типове и затова евентуално разделние на дефиниции от декларации на шаблон няма да се компилира.
- Използването на името на класа и scope оператора `Calculator<T>` се извършва по този начин

  ```
  template <typename T>
  T Calculator<T>::sum(){
      return a + b;
  }
  ```

- При дефиниране на конструктори името на конструктора НЕ е последвано от `<T>` , тъй като конструкторите са специални функции, те не са инстанции на класа

  ```
  template <typename T>
  Calculator<T>::Calculator(){ //при името на констуктора няма <T>
      this->a = T();
      this->b = T();
  }
  ```

- При дефиниране на конструктор по подразбиране е необходимо внимание (горният пример). Тъй като различните типове данни не могат да се приемат една и съща стойност по подразбиране е трудно да зададем стойност по подразбиране на член-данните на класа (`T a` може да стане `int a` , а може и да е обект на клас, примерно `ComplexNumber a` и не можем да дадем в дефиницията на конструктора по подразбиране `this->a = 0`, защото в случая 0 не можем да присвоим на обект от клас `ComplexNumber`). Поради тази причина, най-подходящият начин е да се извика конструктора по подразбиране на самата данна, т.е. `this->a = T()`. Така `a` винаги ще се инициализира чрез конструктора по подразбиране на съответния тип. Ако е `int` ще стане `0` (и примитивните типове данни имат конструктор по подразбиране). Ако е `ComplexNumber` ще стане `0 + 0i`. Опасното тук е, че ние трябва да сме се погрижили класовете, които сме дефинирали да имат дефиниран конструктор по подразбиране.
- Когато се използва обект от класа, примерно в оператор= връщаме референция към обект на класа `Calculator<T>&` той се декларира заедно с `<T>` , тъй като това е обект на шаблонния клас, който трябва да бъде инициализиран от компилатора

```
template <typename T>
Calculator<T>& Calculator<T>::operator=(const Calculator<T>& rhs){
    this->a = rhs.a;
    this->b = rhs.b;
    return *this;
}
```

Целият код:

```
template <typename T = int> //подразбиращ се тип
class Calculator{
private:
    T a;
    T b;
public:
    Calculator();
    Calculator(T a, T b) { //inline имплементация
        this->a = a;
        this->b = b;
    }
    Calculator<T>& operator=(const Calculator<T>& rhs); //предефинирах го само за
примера, иначе няма нужда
    T sum();
};

template <typename T>
Calculator<T>::Calculator(){
    this->a = T();
    this->b = T();
}

template <typename T>
Calculator<T>& Calculator<T>::operator=(const Calculator& rhs){
    this->a = rhs.a;
    this->b = rhs.b;
    return *this;
}

template <typename T>
T Calculator<T>::sum(){
    return a + b;
}
```

Шаблони и приятелски функции

Оставям само линк, който разглежда всички случаи на употреба на приятелски функции на шаблони. Добре написан и подробен, даже малко повече от необходимото:
https://web.mst.edu/~nmjxv3/articles/templates.html

Специализация на шаблони

Както стана ясно до момента, шаблоните се използват, за могат да се генерират класове със сходни функционалности за различни типове данни. Ако обаче има един тип данни, за който би било добре да се различава функционално от другите какво се случва? Решението е специализация на шаблони.

```
template <typename T>
class A {
public:
    A() {
        cout << "Defualt!\n";
    }
};

template <>
class A<int> {
public:
    A() {
        cout << "Int!\n";
    }
};

int main() {
    A<char> a; //"Default!"
    A<double> b; //"Default!"
    A<int> c; //"Int!"
    return 0;
}
```

Ако само деклариараме шаблон и създадем негова/и специализация/и, при употребата му с типове данни, различни от специализираните не може да се дефинира и няма да се компилира.

```
template <typename T>
class A;

template <>
class A<double> {
public:
    A() {
        cout << "Doulbe!\n";
    }
};

template <>
class A<int> {
public:
    A() {
        cout << "Int!\n";
    }
};

int main() {
    A<double> b;
    A<int> c;
    A<char> a; //Не може да се дефинира
```

```
    return 0;
}
```