Программирование на C++

Лекция 7

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2026

Шаблоны функций vs перегруженные функции

Перегруженные функции обычно используются для выполнения **похожих по семантике**, но **разных по синтаксису операций**.

Если же для **каждого типа данных** должны выполняться **идентичные операции**, то более компактным и удобным решением является использование **шаблонов функций**.

Шаблоны функций и шаблонные функции

Шаблоны дают возможность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (перегруженных), называемых **шаблонными функциями**.

Шаблоны функций и шаблонные функции — это не одно и то же!

Шаблонная функция — это «реализация функции по шаблону».

Макросы и шаблоны функций

В языке C некоторым аналогом простейших шаблонов являются макросы, определяемые директивой препроцессора #define.

Однако при использовании макросов компилятор не выполняет проверку соответствия типов, из-за чего нередко возникают серьёзные побочные эффекты.

Шаблоны же позволяют полностью контролировать соответствие типов.

Директива #define

принимает две формы:

- определение констант;
- определение макросов.

Определение констант

#define nameToken value

```
#include <iostream>

#define TEXT "Марс" // определение константы

int main() {
    std::cout <<  TEXT;
    return 0;
}
```

Определение параметризованных макросов

```
#define nameMacros(arg1, arg2, ... ) expression
```

Макросы

```
#define MIN(a,b)  a < b ? a : b          //Может привести к ошибке

…
int a=3;
int b=5;
cout<< MIN(a,b)+5;
```

результат = ?

Макросы

```
#define MIN(a,b)  a < b ? a : b          //Может привести к ошибке

…
int a=5;
int b=3;
cout<< MIN(a,b)+5;
```

```
cout<< a < b ? a : b+5;
```

результат = 8

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))
```

Пример. Реализовать и использовать шаблоны функции для вывода элементов массива и нахождения суммы

```
template <typename T>                    //можно template <class T>
void printArray(const T* array, int count) {
  for (int i=0; i<count; ++i)
    cout << array[i] << " ";
  cout << endl;
}
```

Каждый параметр типа из описания шаблона функции должен появиться в списке параметров функции, по крайней мере, один раз.

```
template <typename T>                    //можно template <class T>
T sumArray(const T* array, int count) {
  T sum=0;
  for (int i=0; i<count; ++i)
    sum+=array[i];
  return sum;
}
```

```
int main() {
  const int aCount = 5, bCount =7, cCount = 6;
  int a[aCount]={1,2,3,4,5};
  double b[bCount]={1.1,2.2,3.3,4.4,5.5};
  char c[cCount]="Hello";
  printArray(a,aCount);
  printArray(b,bCount);
  printArray(c,cCount);
  cout<<endl<<sumArray(a,aCount)<<endl;
  cout<<endl<<sumArray(b,bCount)<<endl;
  //cout<<endl<<sumArray(c,cCount)<<endl;
  return 0;
}
```

Когда компилятор обнаруживает в тексте программы вызов функции printArray, он заменяет T во всей области определения шаблона на тип первого параметра функции printArray и создает шаблонную функцию вывода массива указанного типа данных.

После этого вновь созданная функция компилируется.

Процесс конструирования шаблонной функции называется инстанцированием (конкретизацией) шаблона.

Где размещать шаблоны?

Шаблон должен быть описан либо в том же файле (.cpp), где и используется, либо в заголовочном файле (.h), который подключается к этому файлу (.cpp).

Пример. Реализовать функции нахождения максимума из двух параметров следующих типов данных: int, double, char*, date.

```
template <typename T>
T max(const T a, const T b) {
  return a>b ? a : b;
}
```

```
cout << max(3, 2) << endl;
cout << max(3.5, 20.4) << endl;
cout<<max(2,1.5) << endl;
```

Специализация шаблона

Иногда **общее определение**, предоставляемое шаблоном, для некоторых типов вообще **не работает**, а для некоторых работает неэффективно.

В этом случае необходимо предоставить **специализацию** (специализированное определение для конкретизации шаблона) или использовать **перегруженную функцию** с нужным списком параметров.

```
template <>  //специализация шаблона
const char* max(const char* a, const char* b) {
  if (strcmp(a, b)>0)
    return a;
  else
    return b;
}
```

```
struct date {
  int day, month, year;
};

date max(const date &a, const date &b){      //перегруженная функция
  if (a.year>b.year)
    return a;
  else if (a.year<b.year)
    return b;
  else if (a.month>b.month)
    return a;
  else if (a.month<b.month)
    return b;
  else if (a.day>b.day)
    return a;
  else
    return b;
}
```

```
int main() {
  cout << max(3, 2) << endl;
  cout << max(3.5, 20.4) << endl;
  cout << max("Hello", "By") << endl;
  date a = { 27, 03, 2015 }, b = { 31, 01, 2008 }, c;
  c = max(a, b);
  cout << c.day << '.' << c.month << '.' << c.year << endl;
  return 0;
}
```

Порядок определения, какой экземпляр функции соответствует данному вызову, следующий:

- Сначала компилятор ищет функцию или конкретизацию шаблона, которая точно соответствует по своему имени и типам параметров вызываемой функции.

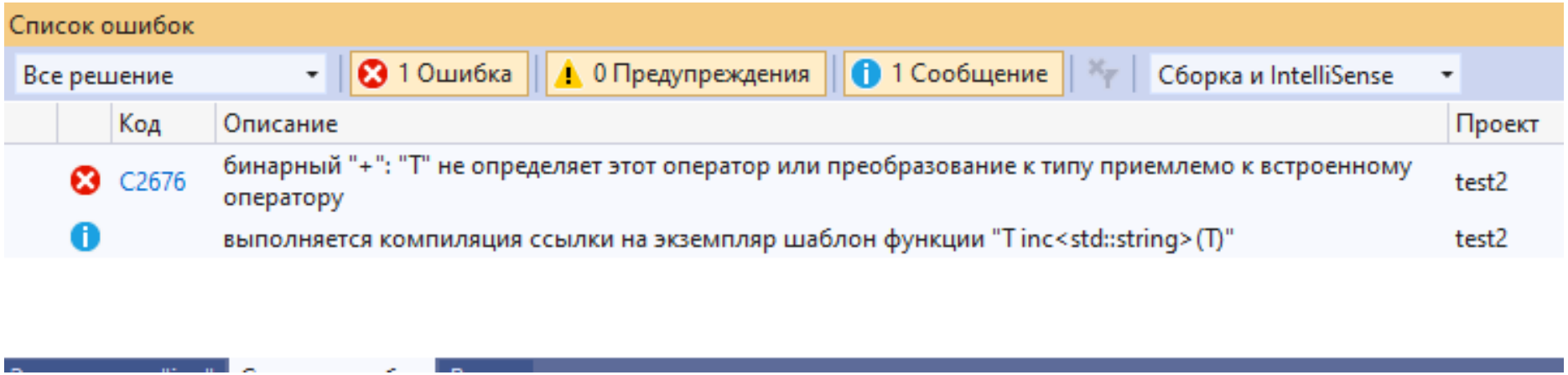
- Если на этом этапе компилятор терпит неудачу, то он ищет шаблон функции, с помощью которого он может сгенерировать шаблонную функцию с точным соответствием типов параметров и имени функции. Если такой шаблон обнаруживается, то компилятор генерирует и использует соответствующую шаблонную функцию. При этом автоматическое преобразование типов не производится.

- И только в случае неудачи в качестве последней попытки компилятор последовательно выполняет процесс подбора перегруженной функции с учетом автоматического преобразования типов.

Ошибка компиляции

```
template<typename T>
T inc(T t) {  return t + 1; }

int main() {
    string s = "Hello!";
    cout<<inc(s);
}
```



```
template <typename T>                    //можно template <class T>
void printArray(const T* array, int count) {
  for (int i=0; i<count; ++i)
    cout << array[i] << " ";
  cout << endl;
}
```

```
int main() {
  const int aCount = 5, bCount =7, cCount = 6;
  int a[aCount]={1,2,3,4,5};
  double b[bCount]={1.1,2.2,3.3,4.4,5.5};
  char c[cCount]="Hello";
  printArray<int*>(a,aCount);
  printArray(b,bCount);
  printArray(c,cCount);
  cout<<endl<<sumArray(a,aCount)<<endl;
  cout<<endl<<sumArray(b,bCount)<<endl;
  //cout<<endl<<sumArray(c,cCount)<<endl;
  return 0;
}
```