Введение в шаблоны.

inline (Шаг в сторону, ну почти)

- изначально означало, что эта функция должна быть оптимизированна и ее тело вставленно в код без вызова, но сейчас компилятор сам решает (может вставлять и и не вставлять почти любые функции).
- (начиная с C++17) Встроенная функция или переменная с внешней связьюимеет следующие дополнительные свойства:
    - В программе может быть более одного определения встроенной функции или переменной , при условии, что каждое определение появляется в отдельной единице трансляции и все определения идентичны. Например, встроенная функция или встроенная переменная могут быть определены в заголовочном файле, который включен в несколько исходных файлов.
    - Он должен быть объявлен встроенным в каждую единицу перевода.
    - Он имеет один и тот же адрес в каждой единице перевода.
- Функция (функции-члены и дружественные функции), определенная полностью внутри определения класса/структуры/union , , не являющейся членом, неявно является inline функцией

Шаблоны

- Шаблоны обеспечивают статический полиморфизм (на этапе компиляции). Зависят только от тех свойств, что используют.
- Параметры шаблонов – типы и константы времени компиляции.
- Также быстры, как и специальный код для соответствующих типов – статическое связывание не ограничивает оптимизатор.
- Не требует связи типов между собой – например, одной иерархии.
- Во время компоновки идентичные инстанцирования, созданные различными единицами перевода, объединяются.

Примеры (шаблонны функциЙ)

Шаблон функции определяет семейство функций.

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

template<class SomeName>
void sort(SomeName begin, SomeName end) {
  /*...*/
}

template<typename To, typename From>
To convert(From f){ /*...*/ }
```

Примеры

```
template<class T, T::type n = 0>
class X{
  public :
  T var_;
  /*...*/
}

struct S{
    using type = int;
};

using T1 = X<S, int, int>; // Ошибка: слишком много аргументов
using T2 = X<>;            // ошибка: нет аргумента по умолчанию для первого шаблона
using T3 = X<1>;           // ошибка: значение 1 не соответствует параметру типа
using T4 = X<int>;         // ошибка: ошибка замены для второго параметра шаблона
using T5 = X<S>;           // OK
```

Параметры шаблонов

Типы как параметры шаблонв:

```
//1)Тип как параметры шаблонв без значения по умолчанию.
template<class T>
class My_vector { /* ... */ };

//2)Тип как параметры шаблонв  со значением по умолчанию.
template<class T = void>
struct My_op_functor { /* ... */ };

//3)Пакет параметров шаблона типа (могут быть разными).
template<typename... Ts>
class My_tuple { /* ... */ };

//4)(начиная с c++ 20)Параметр шаблона ограниченного типа без значения по умолчанию.
template<My_concept T>
class My_constrained_vector { /* ... */ };
// ...(5,6) тоже самое для концептов
```

Параметры шаблонов

параметры шаблона, не являющийся типом;

- Константные выражения времени компиляции (те которые известны во время компиляции)
- Переменные или функции, которые могут быть использованы в нескольких файлах программы

```
template <int &i>
int foo(){ return i; };

int g_a = 100;

void bar(){
    std::cout << foo<g_a>() << std::endl; // Можно
    int a =100;
    // std::cout << foo<a>() << std::endl; // Ошибка
}
```

Параметры шаблонов (примеры)

```
template<class T, size_t N = 100>
struct small_array {
    small_array& operator=(small_array& other){
        /* ... */
        return *this;
    }
    T& operator[](size_t index){
        /* ... */
        return data[index];
    }
    /* ... */
    private:
    T data[N];
};

int main(){
  auto arr = new  small_array<int, 20>();
  std::cout<< sizeof(small_array<int, 20>); // будет 80
  delete arr;
  return 0;
}
```

Параметры шаблонов (примеры)

```
template <class  T, int(T::*func)() const>
int countDuplicates(T* begin, T* end) {
    if (begin == end) return  0;
    int count = 1;
    auto id =  (begin->*func)();
    while ( begin + count < end && ((begin + count)->*func)() == id ) {
        count++;
    }
    return count;
}

struct Data {
    int value_;
    int getValue() const  { return value_; }
    bool operator==(const Data& r) const { return this->value_ == r.value_;}
};

int main() {
    // выделение на стеке
    Data arr[] = { {1}, {1}, {1}, {2}, {2}, {3} };
    int count = countDuplicates<Data, &Data::getValue >(arr, arr + 6);
    return 0;
}
```

Инстанцирование шаблонного класса или функции

- Шаблон класса (или функции) сам по себе не является типом, объектом или любой другой сущностью. Никакой код не генерируется из исходного файла, содержащего только определения шаблонов. Для того, чтобы появился какой-либо код, шаблон должен быть инстанцирован: аргументы шаблона должны быть предоставлены, чтобы компилятор мог сгенерировать фактический класс (или функцию из шаблона функции).
- Все шаблонные функции и методы становятся автоматически inline
- Шаблоны обычно определяют в заголовочных файлах, чтобы использовать в различных единицах трансляции, это не вызывает ошибок линкера так как все функции inline
- Сгенерированный шаблон – это полноценный конкретный класс или функция.

Явное инстанцирование шаблонного класса или функции

```
//MyContainer.h/
template<class TT>
class MyContainer{/*...*/};

// если предполагается, что конкретная реализация
// будет часто использоваться, можно написать ключевое слово
// extern и компилятор не будет реализовывать
// класс/функцию для каждой единицы трансляции,
// (линкер) будет искать реализацию
extern template class MyContainer<int>;

// даем псевдоним для удобства
typedef MyIntContainer<int> string;
```

```
//some.cpp или MyContainer.cpp
// явно истанцируем в одной единице трансляции
template class MyIntContainer<char>;
```

Неявное инстанцирование

- Чаще всего неявное инстанцирование функции происходит во время ее вызова
- Неявное инстанцирование класса происходит при создании объекта

```
template < class T >
struct Z { // определение шаблона
    void f ( )  { }
    void g ( ) ;  // никогда не определено
} ;

template  struct Z < double > ; // явное создание экземпляра Z<double>
Z < int > a ;                  // неявное создание экземпляра Z<int>
Z < char > * p ;               // здесь ничего не создается

p - > f ( ) ;  // неявное создание экземпляра Z<char> и Z<char>::f() происходит здесь.
        // Z<char>::g() никогда не требуется и никогда не создается:
        // его не нужно определять
```

Вывод аргументов шаблона

```
template<typename To, typename From>
To convert(From f){ /*...*/ }

void g(double d) {
    int i = convert<int>(d);    // вызов convert<int, double>(double)
    char c = convert<char>(d);  // вызов convert<char, double>(double)
    int(*ptr)(float) = convert; // создает реализацию convert<int, float>(float)
                                // и сохраняет его адрес в ptr
}

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

void foo {
    max(42, 10); // вызов max<int>
    max(2., 5.); // вызов max<double>
    auto x = max<int>(3, 4.); // max<int>
    vector ivec = {42}; // начиная с  C++17
}
```

Вывод аргументов шаблона

```
#include <cstring>
#include <string>

template <class T, class cmp_t>
void mySort(T *begin, T *end, cmp_t cmp){
    auto mid = (end - begin) / 2;
    if (cmp(*begin, *mid) ) /*...*/
}

struct cmpLikeCstr{
    bool operator()(const std::string  &l, const std::string &r) const {
        return strcmp(l.c_str(), r.c_str()) < 0;
    }
};
int main() {
    // begin - points to first std::string object
    /* ... */
    mySort(begin, end , cmpLikeCstr());
    return 0;
}
```

Специализация шаблона

- Можно указать особую реализацию класса или функции для выбранного типа
- Такая специализация не обязана иметь тот же интерфейс, что и общий шаблон
- Partial template specialization
- Explicit (full) template specialization

```
template <typename T>
inline void writeToFile(ir_ostream &os, const T &x) {
    static_assert(!std::is_pointer<T>::value, "pointers is invalid arguments");
    os.write(reinterpret_cast<const char *>(&x), sizeof(x));
};

template <>
inline void writeToFile<std::string>(ir_ostream &os, const std::string &str) {
    auto size = (uint32_t)str.size();
    os.write(reinterpret_cast<char *>(&size), sizeof(size));
    os.write(str.data(), std::streamsize(str.length()));
};
```

Частичная специализация (только для классов)

```
template<class T, class A>
struct vector {
    /* cose */
};

template<class A>
struct vector<bool, A> {
  /* ... */
};
```

Рекурсивный вариативный шаблон

```
template <class Mod, class... Mods>
void write(ir_ostream &os, const Mod &M, const Mods &...Ms) {
    writeToFile(os, M);
    //std::cout<<"val: "<< M << "\n";
    write(os,Ms...);
}

template <class Mod>
void write(ir_ostream &os, const Mod &M){
    writeToFile(os, M);
    //std::cout<<"val: "<< M << "\n";
}

int main{
  struct { double d = 0.35; std::string s = "my_init_val"; int val = 100; } ss;
  std::ofstream ofile(std::string("my_bin.file"), std::ios::binary);
  ir::write(ofile, ss.d ,ss.s, ss.val);
  return 0;
}
```

приведения типов

- `dynamic_cast<target_type>(arg)` проверяет возможность преобразования в рантайме.
- `static_cast<target_type>(arg)`
- `const_cast<target_type>(arg)` если вы используете это то задумайтесь, может вы что-то спроектировали не очень хорошо
- `reinterpriet_cast<target_type>(arg)`

Не злоупотребляйте const_cast, часто это грязный трюк, скорее всего код был плохо написан.