шаблоны функций

Практическое объектно-ориентированное программирование

03.09.2025

План

- Обобщенное программирование
- Специализация
- Вывод типов в функциях
- Перегрузка

Возводим число в степень

Начнем с первого:
unsigned nth_power(unsigned x, unsigned n); // return x^n
- Как написать тело этой функции?

Выбираем правильный алгоритм

```
unsigned nth_power(unsigned x, unsigned n) {
unsigned acc = 1;
if ((x < 2) || (n == 1)) return x;
while (n > 0)
if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
else { x *= x; n /= 2; }
return acc;
}
```

- Разумеется вариант перемножить x ровно n раз в цикле не рассматривается.

Ищем возможности обобщения

```
unsigned nth_power(unsigned x, unsigned n) {
unsigned acc = 1;
if ((x < 2) || (n == 1)) return x;
while (n > 0)
if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
else { x *= x; n /= 2; }
return acc;
}
```

- Как обобщить этот алгоритм?

Наивное обобщение

```
template <typename T> T nth_power(T x, unsigned n) {
T acc = 1;
if ((x < 2) || (n == 1)) return x;
while (n > 0)
if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
else { x *= x; n /= 2; }
return acc;
}
```

- Тут всё хорошо?

Проблемы наивного обобщения

```
template <typename T> T nth_power(T x, unsigned n) {
T acc = 1;
if ((x < 2) || (n == 1)) return x;
while (n > 0)
if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
else { x *= x; n /= 2; }
return acc;
}
```

- Присвоение единицы сомнительно (вдруг T это матрица?), а сравнение просто неверно (вдруг T знаковый тип?).

Заводим traits

```
template <typename T, typename Trait = default_id_trait<T>>
T nth_power(T x, unsigned n) {
T acc = Trait::id();
if ((x == acc) || (n == 1)) return x;
while (n > 0)
if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
else { x *= x; n /= 2; }
return acc;
}
```

Определяем требования

- Какие требования мы предъявляем к типу T?

```
template <typename T>
T do_nth_power(T x, T acc, unsigned n) {
while (n > 0) {
if ((n & 0x1) == 0x1) {
acc *= x; n -= 1;
}
x *= x; n /= 2;
}
return acc;
}
```

- Можем ли мы формализовать перечень?

Добавляем концепт

- Концепт это список требований к типу.

```
template <typename T> concept multiplicative = requires(T t) {
{ t *= t } -> std::convertible_to<T>;
};
```

- Его можно использовать через ключевое слово requires.

```
template <typename T>
T do_nth_power(T x, T acc, unsigned n)
requires multiplicative<T> && std::copyable<T>
```

- Начинайте пользоваться стандартными концептами.

Интермедия: class vs typename

- Во многих местах шаблонный параметр написан как typename.

template <typename T> int foo(T x);

- Во многих других как class.

template <class T> int foo(T x);

- Особой разницы нет. Раньше class использовался чтобы подчеркнуть, что там ожидается нетривиальный объект, но это уже никому не нужно.
- Предпочтительно (если мы знаем что ожидать) писать концепт.

template <std::integral T> int foo(T x);

Инстанцирование

• Инстанцирование это процесс порождения экземпляра специализации.

```
template <typename T>
T max(T x, T y) { return x > y ? x : y; }
....
max<int>(2, 3); // порождает template<> int max(int, int)
```

• Мы называем этот процесс неявным (implicit) инстанцированием.

• Оно порождает код через подстановку параметра в шаблон и осуществляется по требованию (то есть лениво).

Управление инстанцированием

• Инстанцирование может быть явно запрещено в этой единице трансляции.
extern template int max<int>(int, int);
• Инстанцирование может быть явно вызвано.
template int max<int>(int, int);
• Эта техника может использоваться для уменьшения размера объектных файлов при инстанцировании тяжёлых функций.

Явная специализация

• Кроме инстанцирования (явного или неявного), шаблонная функция или структура может быть явно специализирована.

```
template <typename T> T max(T x, T y) { .... }
template <> int max<int>(int x, int y) { .... }
template <> float max(float x, float y) { .... }
```

• Специализация обязана физически следовать за основным шаблоном.

```
template <> int min<int>(int x, int y) { .... } // ошибка
template <typename T> T min(T x, T y) { .... } // primary тут
```

Правила игры

- Общее правило для функций и не только [temp.spec.general]
- Явное инстанцирование единожды в программе.
- Явная специализация единожды в программе.
- Явное инстанцирование должно следовать за явной специализацией.

```
template <typename T> T max(T x, T y) { .... }
template <> int max<int>(int x, int y) { .... }
template int max<int>(int, int); // вынудили инстанцировать
```

- Нарушение влечет за собой IFNDR.
- Как вы думаете, как играет запет инстанцирования со специализацией?

Инстанцирование и специализация

• Явная специализация может войти в конфликт с инстанцированием

```
template <typename T> T max(T x, T y);
// OK, указываем явную специализацию template <> double
max(double x, double y)  return 42.0;
// никакой implicit instantiation не нужно
int foo()  return max<double>(2.0, 3.0);
// процесс implicit instantiation нужен и он произошёл
int bar()  return max<int>(2, 3);
// ошибка: мы уже породили эту специализацию
template <> int max(int x, int y)  return 42;
```

Удаление специализаций

• Частным случаем явной специализации является её запрет.

```
// для всех указателей template <typename T> void foo(T*);
// но не для char* и не для void*
template <> void foo<char>(char*) = delete;
template <> void foo<void>(void*) = delete;
```

• Как вы думаете, что произойдёт если мы сначала сгенерируем специализацию, а потом запретим её?

Non-type параметры

- Параметры не являющиеся типами могут быть **структурными типами**
- Структурные типы это:
- Скалярные типы (кроме плавающей точки).
- Левые ссылки.
- Структуры у которых все поля и базовые классы public и не mutable. И при этом все поля и поля всех базовых классов тоже структурные типы или массивы.

```
struct Pair { int x, y; };
template <int N, int *PN, int &RN, Pair P> int foo();
```

- Базовая интуиция: всё должно быть compile-time known.

Специализация по nontype параметрам

• Нет никаких проблем в том, чтобы специализировать класс по нетиповым параметрам.
template <typename T, int N> foo(T(&Arr)[N]);
template <> foo<int, 3>(int(&Arr)[3]) {
// тут более эффективная реализация для трёх целых
• Обратите внимание: при явной специализации функций вы обязаны указать все параметры.
• Как вы видите себе специализацию по указателям, ссылкам и структурным типам?
• Массив в шаблонном параметре редуцируется до указателя (как в функции).

Шаблонные шаблонные параметры

- Параметрами могут быть шаблоны классов

```
template <template<typename> typename Cont, typename Elt>
void print_size(const Cont<Elt> &a);
```

- Разумеется специализация по ним тоже возможна.
- Пока что кажется, что это переусложнение.

```
template <typename Container>
void print_size(const Container &a);
```

- Это работает не хуже (а собственно лучше, например для vector<int>).
- Мы вернемся к этому при разговоре о шаблонах классов.

Вывод типов до подстановки

- Для параметров, являющихся типами, работает вывод типов

int x = max(1, 2); // → int max<int>(int, int);

- При выводе режутся ссылки и внешние cv-квалификаторы

const int& a = 1;
const int& b = 2;
int x = max(a, b); // → int max<int>(int, int);

- Вывод не работает, если он не однозначен

unsigned x = 5; do_nth_power(x, 2, n); // FAIL
int a = 1; float b = 1.0; max(a, b); // FAIL

Вывод типов после подстановки

- Вывод типов внутри шаблонной функции даёт точки вывода, где разрешить тип можно только после подстановки

```
template <typename T> T max(T x, T y) { .... }
template <typename T> T min(T x, T y) { .... }
template <typename T> bool
test_minmax(const T &x, const T &y) {
if (x > y) return test_minmax(y, x);
return min(x, y) == x && max(x, y) == y;
}
```

- Таким образом вывод и подстановка включаются попеременно.

Вывод уточнённых типов

• Иногда шаблонный тип аргумента может быть уточнён ссылкой или указателем и cv-квалификатором

template <typename T> T max(const T& x, const T& y);

• В этом случае выведенный тип тоже будет уточнён

int a = max(1, 3); // → int max<int>(const int&, const int&);

• Уточнённый вывод иначе работает с типами: он сохраняет cv-квалификаторы.

template <typename T> void foo (T& x);

const int &a = 3;

int b = foo(a); // → void foo<const int>(const int& x);

Вывод ещё более уточнённых типов

- Вывод типов работает шире, чем люди обычно думают

```
template<typename T> int foo(T(*p)(T));
int bar(int);
foo(bar); // → int foo<int>(int(*)(int));
```

- Могут быть выведены даже параметры, являющиеся константами

```
template<typename T, int N> void buz(T const(&)[N]);
buz(1, 2, 3); // → void buz<int, 3>(int const(&)[3]);
```

- Общее правило: вывод типов матчит сложные композитные типы.

Частичный вывод типов

• В некоторых случаях у нас просто нет контекста вывода.

```
template <typename DstT, typename SrcT>
DstT implicit_cast(SrcT const& x) {
return x;
}
double value = implicit_cast(-1); // fail!
```

• Тогда мы можем указать необходимое и положиться на вывод остального.

```
double value = implicit_cast<double, int>(-1); // ok
double value = implicit_cast<double>(-1); // ok
```

Параметры по умолчанию

• Допустим у вас есть функция, берущая по умолчанию плавающее число.

template <typename T> void foo(T x = 1.0);

• Увы, если его не указать, вывод типов работать не будет.

foo(1); // ok, foo<int>(1);

foo<int>(); // ok, foo<int>(1.0 narrowed to int);

foo(); // fail

• Тем не менее, ситуацию можно исправить. Трюк не так уж и сложен. Догадки?

Шаблонные параметры по умолчанию

• Допустим у вас есть функция, берущая по умолчанию плавающее число.

template <typename T = double> void foo(T x = 1.0);

• Заметьте: во втором случае ниже вывода типов всё ещё нет.

foo(1); // ok, foo<int>(1);

foo<int>(); // ok, foo<int>(1.0 narrowed to int);

foo(); // ok

• Параметр по умолчанию шаблона в данном случае подсказывает компилятору что делать.

Вывод специализирующего типа

• Очень интересной техникой является оставить специализирующий тип выводу типов.
template <typename T> T foo(T x) { code for all }
template <> int foo(int x) { code for int } // $\rightarrow$ foo<int>
• Это удобно и это часто применяется. Но иногда сложно понять по чему специализируем.

Общий обзор правил перегрузки

- Выбирается множество **перегруженных имён**.
- Выбирается множество **кандидатов**.
- Из множества кандидатов выбираются **жизнеспособные** (viable) кандидаты для данной перегрузки.
- Лучший из жизнеспособных кандидатов выбирается на основании **цепочек неявных преобразований** для каждого параметра.
- Если лучший кандидат **существует и является единственным**, то перегрузка разрешена успешно, иначе программа ill-formed.

Спрятанные имена

Олды на месте? Что на экране?

- Вопрос для новичка: что на экране?

```
struct B {
void f(int) { std::cout « "B«< std::endl; }
};
struct D : B {
void f(const char*) { std::cout « "D«< std::endl; }
};
int main() {
D d; d.f(0);
}
```

Проблема: операторы

- Обычно оператор может находиться в любом пространстве имён.

```
std::cout « "Hello"!;
```

- Это вполне может быть эквивалентно следующему.

```
operator« (std::cout, "Hello"!);
```

- Чтобы это работало, это должен быть оператор из пространства имён std.

```
std::operator« (std::cout, "Hello"!);
```

- Но компилятор не может об этом догадаться из записи std::a « b

Решение: поиск Кёнига

- Эндрю Кёниг предложил решение в начале 90-х

1. Компилятор ищет имя функции из текущего и всех охватывающих пространств имён.
2. Если оно не найдено, компилятор ищет имя функции в пространствах имён её аргументов.

```
namespace N { struct A; int f(A*); }
int g(N::A *a) { int i = f(a); return i; }
```

Решение: поиск Кёнига

```
typedef int f;
namespace N { struct A; int f(A*); }
int g(N::A *a) { int i = f(a); return i; }
```

Поиск Кёнига и шаблоны

- Следующий пример не работает

```
namespace N {
struct A;
template <typename T> int f(A*);
}
int g(N::A *a){
int i = f<int>(a); // FAIL
return i;
}
```

- Кто-нибудь может угадать причину?

Поиск Кёнига и шаблоны

• Можно заставить это работать, введя f как имя шаблонной функции

```
namespace N {
struct A;
template <typename T> int f(A*);
}
template <typename T> void f(int); // неважно какой параметр
int g(N::A *a) {
int i = f<int>(a); // теперь всё ok
return i;
}
```

Идея построения цепочки

- С наивной точки зрения цепочку преобразований входят:
- С высшим приоритетом: стандартные преобразования.
- Немного ниже: пользовательские преобразования.
- С низшим приоритетом: троеточия.
- Сложности начинаются когда их комбинируется много разных.

```
struct S { S(int){} };
void foo(int); // 1
void foo(S); // 2
void foo(...); // 3
foo(1); // → 1
```

Стандартные преобразования

- Трансформации объектов (ранг точного совпадения).

int arr[10]; int *p = arr; // [conv.array]

- Коррекции квалификаторов (ранг точного совпадения).

int x; const int *px = &x; // [conv.qual]

- Продвижения (ранг продвижения).

int res = true; // [conv.prom]

- Конверсии (ранг конверсии).

float f = 1; // [conv.fpint]

Таблица

Table 16: Conversions [tab:over.ics.scs]

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		7.3.1
Array-to-pointer conversion			7.3.2
Function-to-pointer conversion			7.3.3
Qualification conversions	Qualification Adjustment		7.3.5
Function pointer conversion			7.3.13
Integral promotions	Promotion	Promotion	7.3.6
Floating-point promotion			7.3.7
Integral conversions	Conversion	Conversion	7.3.8
Floating-point conversions			7.3.9
Floating-integral conversions			7.3.10
Pointer conversions			7.3.11
Pointer-to-member conversions			7.3.12
Boolean conversions			7.3.14

Рис.: Преобразование типов

Пользовательские преобразования

• Задаются implicit конструктором либо оператором преобразования.

```
struct A {
operator int(); // 1
operator double(); // 2
};
int i = A{}; // calls (1)
```

• При этом (1) лучше чем (2) потому что для него нужно меньше стандартных преобразований.

• Интуитивно: у цепочки короче хвост значит она лучше.

Перегрузка и шаблоны.

• Шаблон может выиграть перегрузку. При этом запускается вывод типов.

void foo(double x); // 1

template <typename T> void foo(T x); // 2

foo(1); // → несомненно, 2

• Для выигравшего перегрузку шаблона запускается инстанцирование или ищется специализация.

template <> void foo<int>(int x); // 3

foo(1); // → что вы думаете?

Что если вывод удался дважды?

- Рассмотрим более сложный пример

```
template <typename T> void f(T); // 1
template <typename T> void f(T*); // 2
```

- В точке вызова у нас нечто вроде int ***a;

```
foo(a); // → ?
```

- Здесь вывод работает, но шаблоны сами по себе перегружены
- Тогда внезапно вывод будет повторён дважды

Контрольный вопрос

1. template <typename T, typename U> void foo(T, U);
2. template <typename T, typename U> void foo(T*, U*);
3. template <> void foo<int*, int*>(int*, int*);

```
int x;
foo(&x, &x); // ???
```