11 Функции

11.1 Структруное программирование

Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков (логически объединенных последовательностей инструкций).

Основные принципы структурного программирования:

- Используются операторы последовательности, ветвления и цикла.
- Операторы могут быть вложены друг в друга.
- Повторяющиеся фрагменты объединяются в подпрограммы (**функции**)
- ...

11.2 Определение функции

Функция — элемент программы, который связывает последовательность инструкций с идентификатором (именем функции) и списком параметров

В общем виде определение функции выглядит так:

```
ReturnType Name(Type1 p1, Type2 p2, ...) {
  // function body
}
```

Например:

```
// Max function: returns an int, takes 2 ints
int Max(int x, int y) {
  return x > y ? x : y;
}
```

Далее функция может быть вызвана из какой-нибудь другой функции:

```
int main() {
  int a;
  int b;
  std::cin >> a >> b;
  std::cout << Max(x, y) << '\n';
  return 0;
}
```

Как и любую другую сущность, перед использованием функцию необходимо объявить, но не обязательно определять.

```
int Max(int x, int y); // Function declaration or prototype

int main() {
  int a;
  int b;
  std::cin >> a >> b;
  std::cout << Max(x, y) << '\n';
  return 0;
}

int Max(int x, int y) {  // Function definition
  return x > y ? x : y;
}
```

11.3 Объявление функций

Данный синтаксис называется объявлением (прототипом) функции (без определения).

`int Max(int x, int y)`

Объявление (возможно с определением) должно обязательно предшествовать использованию функции.

При объявлении функции можно опускать имена параметров:

`int Max(int, int )`

11.3.1 ODR (One Definition Rule)

У одной и той же функции может быть несколько объявлений в программе. Но определение обязано быть одно!

```
// OK
int Max(int, int y);
int Max(int a, int b);
int Max(int, int);

int main() { ... }
```

```
// Error
int Max(int x, int y) {
  return x > y ? x : y;
}

int Max(int a, int b) {
  return a > b ? a : b;
}

int main() { ... }
```

11.4 Параметры функций

11.4.1 Определения

Параметры, указанные в объявлении функции, называются *формальными параметрами*. Параметры, переданные в функцию — *фактическими параметрами*.

```
int Max(int x, int y) { // x, y --- formal parameters
  return x > y ? x : y;
}
int main() {
  std::cout << Max(1, -1) << '\n'; // 1, -1 - actual parameters
  return 0;
}
```

11.4.2 Копирование параметров

При передаче фактических параметров они копируются в формальные.

```
int Max(int x, int y) { // x, y - local variables of the Max function
  x = x > y ? x : y;
  return x;
}

int main() { // x, y - local variables of the main function
  int x = 0;
  int y = 1;
  std::cout << Max(x, y) << '\n'; // 1
  std::cout << x << '␣' << y << '\n'; // 0 1
  return 0;
}
```

11.4.3 Параметры по умолчанию

Некоторые параметры могут иметь значение по умолчанию. В этом случае они могут быть опущены при вызове функции.

Параметры со значениями по умолчанию могут идти только в конце списка.

```
int Max(int x, int y = 0); // OK
int Min(int x = 1, int y = 0); // OK
int Sum(int x = 0, int y); // CE

int main() {
  Max(1, 2);
  Max(1);
  Min(1, 2);
  Min(1);
  Min();
  return 0;
}
```

Параметры по умолчанию должны быть указаны хотя бы 1 раз, а в дальнейшем могут быть опущены:

```
int Max(int, int = 0); // so it is also possible
int Max(int, int); // declaring the SAME function

int main() {
  ...
  Max(a);
  ...
}

int Max(int x, int y) {
  return x > y ? x : y;
}
```

11.5 Возвращаемое значение

- Тип возвращаемого значения указывается перед именем функции.
- Результат возвращается с помощью оператора `return` .
- Значение выражения, стоящего справа от `return` , при необходимости (и возможности) преобразуется в возвращаемый тип.

```
int F() { return 0; } // OK
int G() { return 1.0; } // OK 1.0 is converted to 1
int H() { return "Hello"; } // CE: string is not converted to int
```

Операторов `return` может быть несколько:

```
int Calculate(int x, int y, char c) {
  if (c == '+') {
    return x + y;
  }
  if (c == '-') {
    return x - y;
  }
  if (c == '*') {
    return x * y;
  }
  return x / y;
}
```

Замечание: `else` после return необязателен (`return` завершает выполнение функции)

Если функция ничего не вернет, то ошибки не будет, но обращение к результату в этом случае приведет к *UB*:

```
int Max(int x, int y) {
  if (x > 0) {
    return x > y ? x : y;
  }
}

int main() {
  Max(-1, -1); // OK
  std::cout << Max(-1, -1); // UB
  return 0;
}
```

Чтобы указать, что функция ничего не возвращает, используется тип `void` :

```
void Print(int x) {
  std::cout << x << '\n';
}

int main() {
  Print(0);
  return 0;
}
```

Для досрочного завершения такой функции используется `return` без параметра.

```
void Print(int x) {
  if (x < 0) return;
  std::cout << x << '\n';
}
```

11.6 Передача указателей, массивов, ссылок

11.6.1 Проблема

Довольно часто (например, при сортировке) возникает необходимость обменять значения двух переменных: Логично написать функцию, которая бы занималась этим:

```
void Swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Данная функция обменивает лишь локальные копии. Исходные перемнные не меняются:

```
int main() {
  int x = 0;
  int y = 1;
  Swap(x, y); // x == 0, y == 1
  return 0;
}
```

Для решения этой проблемы можно использовать передачу указателей и ссылок. Об этом ниже.

11.6.2 Передача указателей

Чтобы из функции получить доступ к локальным переменным другой функции, можно воспользоваться указателями:

```
void Swap(int* px, int* py) { // accept addresses of ints
  int tmp = *px;
  *px = *py; // change value at px address
  *py = tmp; // change value at py address
}

int main() {
  int x = 0;
  int y = 1;
  Swap(&x, &y); // x == 0, y == 1
  return 0;
}
```

11.6.3 Передача ссылок

Более удобным способом является передача параметров по ссылке:

```
void Swap(int& x, int& y) { // accept references
  int tmp = x;
  x = y; // change the value referred to by x
  y = tmp; // change the value referred to by y
}

int main() {
  int x = 0;
  int y = 1;
  Swap(x, y); // x == 0, y == 1
  return 0;
}
```

11.6.4 Передача по константной ссылке

Чтобы избежать копирования при передаче параметров в функции, можно также воспользоваться передачей по ссылке.

Чтобы гарантировать, что исходная переменная не изменится, можно передать константную ссылку:

```
int Max(const int& x, const int& y) {
  return x > y ? x : y;
}
```

Замечание: копирование примитивных типов — очень дешевая операция(operator), поэтому в таких ситуациях лучше использовать передачу по значению. Передача по ссылке имеет смысл для "тяжелых"объектов (структур, классов)

11.6.5 Передача массивов

Самый простой способ передать массив в функцию — передать указатель на начало + размер массива:

```
int Max(const int* array, int size) {
  int max = array[0]; // assume size > 0
  for (int i = 1; i < size; ++i) {
    if (max < array[i]) max = array[i];
  }
  return max;
}

int main() {
  int array[100]; // or int* array = new int[100];
  for (int i = 0; i < 100; ++i) std::cin >> array[i];
  std::cout << Max(array, 100) << '\n';
  // delete[] array;
  return 0;
}
```

Передача массивов по значению не дает ожидаемого результата.

```
void f(int array[50]); // <=> void f(int* array)

int main() {
  int normal[50];
  f(normal); // OK

  int large[100];
  f(large); // OK, but can't process items beyond 50

  int small[10];
  f(small); // Ok can't access memory beyond 10

  return 0;
}
```

Тип массива в качестве параметра автоматически преобразуется в указатель.
Решением данной задачи является передача массива по указателю:

```
void f(int (*array_ptr)[50]);

int main() {
  int normal[50];
  f(&normal); // OK

  int large[100];
  f(&large); // CE

  int small[10];
  f(&small); // CE

  return 0;
}
```

Или аналогично можно передать по ссылке:

```
void f(int (&array_ref)[50]);

int main() {
  int normal[50];
  f(normal); // OK

  int large[100];
  f(large); // CE

  int small[10];
  f(small); // CE

  return 0;
}
```

11.7 Статические локальные переменные

Статические переменные инициализируются **один раз** и живут до конца программы, то есть сохраняют свои значения между вызовами.

Область действия — глобальная, область видимости — блок, в котором объявлена.

По умолчанию инициализируются нулем.

Пример:

```
int F() {
  static int x = 1;
  return x++;
}

int main() {
  std::cout << F() << '\n'; // 1
  std::cout << F() << '\n'; // 2
  std::cout << F() << '\n'; // 3
}
```

11.8 Возврат массивов, указателей и ссылок

11.8.1 Проблемы

Массивы нельзя не только принимать по значению, но и возвращать из функций:

```
int[10] Function() { // Error
  int array[10]{};
  return array;
}
```

Но даже если мы попробуем вернуть указатель или ссылку, например:

```
int* F() {
  int x = 0;
  return &x;
}

int& G() {
  int x = 0;
  return x;
}
```

То у нас будет проблема, так как возврат указателя или ссылки на локальную переменную функции — опасная операция(operator), так как после завершения работы функции переменная уничтожается. Следовательно, получаем провисший указатель/ссылку.

`std::cout << *F() << ' ' << G() << 'n'; //UB`

11.8.2 Допустимые варианты

Допустимыми вариантами являются:

- Возврат указателя/ссылки на объект в динамической памяти

```
int* CreateArray(int size) {
  return new int[size];
  // hopefully the caller remembers to call delete[]
}
```

- Возврат указателя/ссылки на объект, полученный с помощью указателя/ссылки

```
int& Get(int* p) {
  return *p;
  // we believe that p is a valid pointer
}
```

- Возврат указателя/ссылки на глобальную переменную

```
int x = 0;

int* GetPointer() {
  return &x;
}
```

- Возврат указателя/ссылки на статическую переменную

```
int& Get() {
  static int x = 0;
  return x;
}
```

11.9 Присвоение возвращаемому значению

Если функция возвращает ссылку, то ее результату можно присваивать (и вообще работать с ее результатом как с переменной):

```
int& F();

int& r = F();
F() = 5;
int* p = &F();
```

Теперь должно быть понятно, как работают lvalue выражения (присваивания, ++ и т.п.) — они возвращают не по значению, а по ссылке!

```
int& operator+=(int& x, int y) {
  x = x + y;
  return x;
}
```

11.10 Перегрузка функций

11.10.1 Наследие от C

В языке C функции не могут иметь одинаковые имена, даже если принимают разные типы и количество параметров. Примеры данного наследия можно найти, например, в библиотеке `cmath` :

```
int abs(int x);
long labs(long x);
long long llabs(long long x);
float fabsf(float x);
double fabs(double x);
long double fabsl(long double x);
```

11.10.2 Современный C++

В C++ разрешено заводить функции с одинаковыми именами, различающиеся списком параметров. Данный механизм называется *перегрузкой функций*:

```
int abs(int x);
long abs(long x);
long long abs(long long x);
float abs(float x);
double abs(double x);
long double abs(long double x);
```

11.10.3 Примеры перегрузки

Для того, чтобы перегрузить функцию необходимо, чтобы параметры функции имели либо **разный тип**:

```
int Max(int x, int y) {
  return x > y ? x : y;
}
int Max(const int* array, int size) {
  int res = array[0];
  for (int i = 0; i < size; ++i) res = Max(res, array[i]);
  return max;
}
```

Либо **разное число параметров**:

```
int Minus(int x) { return -x; }
int Minus(int x, int y) { return x - y; }
```

Важно!

Если параметры одинаковые, а функции различаются лишь возвращаемым типом, то это не является перегрузкой функций и является ошибкой.

```
int Abs(int x) { return x > 0 ? x : -x; }
double Abs(int y) { return x > 0 ? x : -x; }
```

11.10.4 Разрешение перегрузки

Процесс выбора подходящей функции называется *разрешением перегрузки*.

```
int F(int) { return 1; }
int F(double) { return 2; }
int F(char) { return 3; }

F(0);   // 1
F(0.0); // 2
F('0'); // 3
```

Разрешение перегрузки происходит только на основе имени функции и типов переданных аргументов (сигнатура функции).

```
long F(long) { return 1; }
long long F(long long) { return 2; }

long x = F(0); // Error: ambiguous call
```

Общее правило: выбирается наиболее подходящая по параметрам функция, требующая наименьшее число преобразований (conversion) типов.

Замечание: расширение (promotion) типа (int -> long -> long long или float -> double) не является преобразованием.

Замечание 2: float -> long double, double -> long double считаются преобразованиями, а не расширениями.

Более подробные правила выглядят следующим образом:

Шаг №1: C++ пытается найти точное совпадение. Это тот случай, когда фактический аргумент точно соответствует типу параметра одной из перегруженных функций.

Например:

```
void print(char *value);
void print(int value);

print(0); // exact match with print(int)
```

Хотя 0 может технически соответствовать и print(char *) (как нулевой указатель), но он точно соответствует print(int). Таким образом, print(int) является лучшим (точным) совпадением.

Шаг №2: Если точного совпадения не найдено, то C++ пытается найти совпадение путем дальнейшего неявного преобразования типов. Таких преобразований достаточно много. Если вкратце, то:

- char, unsigned char и short конвертируются в int
- unsigned short может конвертироваться в int или unsigned int (в зависимости от размера int)
- float конвертируется в double
- enum конвертируется в int

Например:

```
void print(char *value);
void print(int value);

print('b'); // match with print(int) after implicit conversion
```

Шаг №3: Если неявное преобразование невозможно, то C++ пытается найти соответствие посредством стандартного преобразования. В стандартном преобразовании:

- Любой числовой тип будет соответствовать любому другому числовому типу, включая unsigned (например, int равно float)
- enum соответствует формальному типу числового типа данных (например, enum равно float)
- Ноль соответствует типу указателя и числовому типу (например, 0 как char * или 0 как float)
- Указатель соответствует указателю типа void

Например:

```
struct Employee; // let's miss the definition
void print(float value);
void print(Employee value);

print('b'); // 'b' is converted to match the print(float) version
```

Обратите внимание, все стандартные преобразования считаются равными. Ни одно из них не считается выше остальных по приоритету.

Шаг №4: C++ пытается найти соответствие путем пользовательского преобразования. Хотя мы еще не рассматривали классы, но они могут определять преобразования в другие типы данных, которые могут быть неявно применены к объектам этих классов.

Замечание про const: При выборе перегрузки const на верхнем уровне игнорируется. То есть `F(int)` и `F(const int)` задают одинаковые функции!

```
void F(int); /* <=> */ void F(const int);
void F(int*); /* <=> */ void F(int* const);
void F(const int*); /* <=> */ void F(const int* const);

void F(int&); /* != */ void F(const int&);
void F(int*); /* != */ void F(const int*);
```

11.11 Рекурсия

Рекурсия — метод решения задач, основанный на решении аналогичной задачи меньшего размера.

Рекурсивная функция — функция, которая вызывает сама себя.

11.11.1 Прямая рекурсия

Наиболее простой является *прямая рекурсия* — ситуация, когда функция вызывает себя непосредственно (напрямую).

Пример рекурсивной функции — функция, вычисляющая факториал:

```
int Factorial(int n) {
  return n <= 1 ? 1 : n * Factorial(n - 1);
}
```

Важной частью рекурсивных функций является условие завершения рекурсии.

11.11.2 Рекурсия vs Итерации

Ясно, что эту же задачу (факториал) можно решить итеративно:

```
int Factorial(int n) {
  int res = 1;
  for (int i = 2; i <= n; ++i) {
    res *= i;
  }
  return res;
}
```

Рекурсивный алгоритм более лаконичен и "математичен но требует $\mathcal{O}(n)$ памяти для вычисления (стек рекурсии — хранение локальных переменных, адреса возврата, контекста и т.д.)

Сравним следующие алгоритмы:

```
int Fibonacci(int n) {
  return n <= 1 ? 1 : Fibonacci(n - 2) + Fibonacci(n - 1);
}

int Fibonacci(int n) {
  int a = 0;
  int b = 1; // f(0)
  for (int i = 0; i < n; ++i) {
    b += a; // f(i)
    a = b - a; // f(i - 1)
  }
  return b;
}
```

Рекурсивный алгоритм не только требует больше памяти, но и работает за $\Omega(1.5^n)$
Таким образом при использовании рекурсии важно помнить:

- Любой рекурсивный алгоритм можно свести к итеративному
- Рекурсия, как правило, более лаконична, но требует больше ресурсов.
- Рекурсию имеет смысл использовать, если сложность итеративной реализации окупает прирост в производительности.

11.11.3 Косвенная рекурсия

Косвенно рекурсивной называется функция, которая вызывает себя через вызов другой (других) функции.

При использовании косвенной рекурсии важно помнить, что для использования любой сущности необходимо ее сначала объявить. Поэтому для таких функций вначале пишутся все используемые прототипы.

```
int F(int n);
int G(int n);

int F(int n) {
  return n <= 1 ? n : 1 + G(n);
}

int G(int n) {
  return n <= 1 ? n : n + F(n - 1);
}
```

11.12 Указатели и ссылки на функцию

Во время исполнения программы код функции (инструкции) хранится в памяти, а, следовательно, на функцию можно создать указатель или ссылку:

```
// A function that takes an int and returns an int*
int* F(int);

// Pointer to a function that takes an int and returns an int*
int (*G)(int);
```

Указатель/ссылку на функцию можно проинициализировать адресом или самой функцией:

```
int F(int) { ... }

int (*f_pointer)(int) = &F;
// Functions, like arrays, are reduced to pointers.
int (*f_pointer2)(int) = F;


int (&f_reference)(int) = F;
```

11.12.1 Применение

```
(*f_pointer)(1); // call F(1)
// call F(3) (function pointer does not need to be dereferenced)
f_pointer2(3);

f_reference(5); // call F(5)
```

Как правило, указатели и ссылки на функцию используются для передачи их в другие функции.

Например, если хотим задать сортировку по другому критерию:

```
bool Greater(int x, int y) {
  return x > y;
}

void Sort(int* begin, int* end, bool (*compare)(int, int)) {
  ...
  if (compare(x, y))
  ... ...
}

// ...

Sort(array, array + n, Greater); // non-ascending sort
```

Указатели на функции (в отличие от функций) можно хранить в массивах. Таким образом, можно обращаться к ним в цикле:

```
int (*transform[10])(int) = {Action0, Action1, ..., Action9};

int x;
std::cin >> x;
for (int i = 0; i < 10; ++i) {
  x = transform[i](x);
}
```

11.12.2 Правила чтения

К правилам чтения типов добавляется условие: если при движении слева направо встречаем открывающуюся круглую скобку, то далее идут параметры функции

Например:

- `int (*(*a)[5]) (int, float)` — Указатель на массив из 5 указателей на функции `int(int, float)`
- `int* (*b[10]) (int(*)(int))` — Массив из 10 указателей на функции `int* (int(*)(int))`
- `int (*(*a) ()) (const int&)` — Указатель на функцию без аргументов и возвращающую `int(*)(const int&)`

11.13 Шаблоны функций

Шаблоны функций являются логическим продолжением перегрузки функций. Так, например, функция модуля перегруженная для различных типов данных имеют один и тот же код внутри:

```
int abs(int x) { return x > 0 ? x : -x; }
long abs(long x) { return x > 0 ? x : -x; }
long long abs(long long x) { return x > 0 ? x : -x; }
float abs(float x) { return x > 0 ? x : -x; }
double abs(double x) { return x > 0 ? x : -x; }
long double abs(long double x) { return x > 0 ? x : -x; }
```

Для удобства мы хотели бы это поменять и написать один шаблон:

```
template <class T>
T abs(T x) {
  return x > 0 ? x : -x;
}
```

11.13.1 Синтаксис

- В начале объявляется список шаблонных параметров:

```
template <class T> or template <typename T>
```

- Далее следует определение (или объявление) шаблонной функции.

```
template <class T>
T Abs(T x) {
  return x > 0 ? x : -x;
}

template <class T>
T Sum(T x, T y) {
  return x + y;
}
```

- Допустимо использование нескольких шаблонных параметров.

```
template <class T, class U>
void Print(T x, U y) { std::cout << x << ' ' << y; }
```

11.13.2 Применение шаблонов

После того как шаблон функции объявлен его можно использовать.

Нужный тип выводится автоматически *по переданным аргументам* (при наличии).

```
template <class T>
T Abs(T x) { return x > 0 ? x : -x; }

Abs(0.0); // double Abs(double)

template <class T>
T Sum(T x, T y) { return x + y; }

Sum(1, 1); // Ok [T == int]
Sum(1, 0.0); // CE ([T == int] or [T == double])

template <class T>
T GetZero() { return 0; }

GetZero(); // CE - type cannot be inferred
```

Если результат вывода не устраивает или вывод типа невозможен, можно заставить компилятор вызвать функцию с конкретным типом:

```
template <class T>
T Sum(T x, T y) { return x + y; }

Sum<long>(1, 1); // Ok [T == long]
Sum<double>(1, 0.0); // Ok [T == double]

template <class T>
T GetZero() { return 0; }

GetZero<float>(); // Ok [T == float]
```

11.13.3 Вывод типа шаблона при передаче по значению

При передаче аргумента по значению тип `T` выводится по следующим правилам:

1. CV-квалификаторы (`const` , `volatile`) игнорируются.
2. Ссылки отбрасываются.
3. Массивы низводятся до указателей.
4. Функции низводятся до указателей на функцию
5. Типы соответствующие одному шаблонному типу `T` должны совпадать (после выполнения всех действий выше).

Пример:

```
template <class T>
void f(T x, T y);

int x = 0;

const int cx = 1;
f(x, x); // Ok: [T=int]
f(x, cx); // Ok: [T=int]
f(cx, cx); // Ok: [T=int]

int& rx = x;
int arr[11];

f(cx, rx); // Ok: [T=int]
f(&rx, arr); // Ok: [T=int*]
f(&cx, &x); // CE: [T=const int*] or [T=int*]
f(0, 0.0); // CE: [T=int] or [T=double]
f<double>(0, 0.0); // Ok: [T=double] (type is not inferred, but substit
```

В случае 2 типов шаблона, они также определяются автоматически:

```
template <class T, class U>
void f(T x, U y);

// You can output both parameters (if they are outputable)
f(0, 0.0); // Ok: [T=int, U=double]

// You can explicitly specify both
f<double, double>(0, 0.0); // Ok: [T=double, U=double]

// You can specify the first one, the second one will be displayed auto
f<float>(0, 0.0); // Ok: [T=float, U=double]
```

11.13.4 Вывод типа шаблона при передаче по ссылке

Отличие от передачи по значению в том, что при передаче по ссылке или указателю низведений типов не происходит.

```
template <class T>
void f(T& x) { ... }

int x = 0;
const int cx = 1;
int& rx = x;
int arr[10];

f(x); // Ok: void f<int>(int& x);
f(cx); // Ok: void f<const int>(const int& x);
f(rx); // Ok: void f<int>(int& x); no reference to reference
f(arr); // Ok: void f<int[10]>(int (&x)[10])
f(0); // CE: you cannot create references to temporary values

f<const int>(0); // Ok: void f<const int>(const int&)
// or
template <class T>
void f(const T& x) { ... }
```

11.13.5 Параметры шаблона по умолчанию

Можно указать значение шаблонного параметра по умолчанию, тогда, в случае если тип невозможно вывести, будет использоваться значение по умолчанию.

```
template <class T = int>
T GetZero() { return 0; }

GetZero(); // Ok: [T=int]
GetZero<double>(); // Ok [T=double]
```

Можно ссылаться на предыдущие шаблонные параметры

```
template <class T, class U = T>
U f(T x) { ... }

f<int, double>(0); // Ok: [T=int, U=double]
f<float>(0); // Ok: [T=float, U=float]
f(0); // Ok: [T=int, U=int]
```

Важно! Значения аргументов по умолчанию не могут использоваться для вывода шаблонного типа

```
template <class T>
void f(T x = 0) { ... }

f(); // CE: type T is not deduced!
```

Можно делать только так:

```
template <class T = int>
void f(T x = 0) { ... }

f(); // Ok: [T=int]
```

11.13.6 Инстанцирование шаблонов

Шаблонные функции работают не как обычные функции, для них выполнены следующие правила:

- Шаблон функции - это **НЕ** функция!
- Генерации исполняемого кода не происходит, если шаблон ни разу не вызван.
- В случае вызова шаблонной функции создается лишь нужная версия (с вызываемыми параметрами).
- Процесс генерации кода из шаблона называется *инстанцированием* шаблона.

```
template <class T>
void f(T x) { ... }

int main() {
  f(0); // instantiated f<int>
  f(0.0); // instantiated f<double>
  f(1); // Used by f<int> (already instantiated)
  return 0; // there are no other versions of f<T>!
}
```

Можно явно попросить инстанцировать шаблон (даже если он не используется).

```
template <class T>
void f(T x) { ... }

template void f(float); // Explicitly instantiate f<float>

int main() {
  f(0); // instantiated f<int>
  f(0.0); // instantiated f<double>
  f(1); // Used by f<int> (already instantiated)
  return 0; // There are f<int>, f<double>, f<float>
}
```

Это может быть полезно в случае многофайловых программ, когда вы не хотите инстанцировать одну и ту же функцию несколько раз в разных единицах трансляции.

Компиляция шаблонов происходит в два этапа:

1. При объявлении проверяется лишь синтаксис языка и условия, которые не зависят от параметра шаблона
2. Во время инстанцирования происходит полная проверка кода на корректность и генерация машинного кода

```
template <class T>
void f(T x) {
  x = x + x; // Stage 2 (is it possible to add and assign T?)
  g(); // Stage 1 (does not depend on T)
  g(x); // Stage 2 (depends on T)
  static_assert(sizeof(char) == 1); // 1 stage (independent of T)
  static_assert(sizeof(T) > sizeof(char)); // Stage 2 (depends on T)
}
```

11.13.7 Перегрузка шаблонов функций

Как и обычные функции шаблоны можно перегружать.

Общие правила выбора шаблона:

- Точные соответствия всегда побеждают остальные перегрузки.
- Если есть несколько точных соответствий, выиграет соответствие с меньшим числом подстановок и приведений типов.
- При прочих равных обычная функция предпочтительнее шаблона.

```
template <class T, class U>
int f(T x, U y) { return 1; }

template <class T>
int f(T x, T y) { return 2; }

int f(int x, int y) { return 3; }

f(0, 0.0); // 1 (exact match)
f(0.0, 0.0); // 2 (fewer substitutions)
f(0, 0); // 3 (priority over non-template function)

//If we want to use a template
f<>(0, 0); // 2
```

11.13.8 Специализация шаблона

Представим такую ситуацию:

```
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

struct Complex {
  double re;
  double im;
};

Complex c{3, 4}; // 3 + 4i
abs(c); // CE: no match for operator>
```

Решение - специализация шаблона.

Специализация шаблона позволяет определить реализацию для конкретного набора параметров

```
// general pattern
template <class T>
T abs(T x) { return x > 0 ? x : -x; }

// specialization
template <>
Complex abs(Complex x) {
  return {sqrt(x.re * x.re + x.im * x.im), 0};
}

Complex c{3, 4};
abs(c); // Ok: [T=Complex] {5, 0}
```

```
template <class T> T GetZero() { return 0; }
template <> Complex GetZero() { return {0, 0}; }
```

11.13.9 Параметры шаблона не являющиеся типами

В качестве шаблонных параметров помимо типов могут выступать еще и:

- Целые числа
- Указатели
- Ссылки
- `std::nullptr_t`
- Числа с плавающей точкой (C++20)
- Некоторые классы специального вида (C++20)

```
template <int N, int M>
int Sum() { return N + M; }

Sum<3, 8>(); // Ok: 11

int x = 1;
Sum<1, x>(); // CE (N and M must be compile-time constants!)
```

```
template <class T, size_t N>
size_t ArraySize(const T (&array)[N]) { return N; }

int arr[11];
ArraySize(arr); // Ok: 11 (N is deduced from the type of arr)
```

```
template <void (*FPtr)(int)>
void Call(int x) { FPtr(x); }

Call<f>(10);
```

12 Пользовательские типы

12.1 Перечисления (enum)

12.1.1 Общее представление

Перечисление — пользовательский тип данных, значения которого ограничены набором именованных констант некоторого целого типа.

Проще говоря, перечисление —- тип с некоторым небольшим числом значений.