C++

Башарин Егор

eaniconer@gmail.com

Лекция IV

Functions

Функции

Тип функции

```
void z(int i, double j) {}
int main() {
    z = 10;
    // error: non-object type 'void (int, double)'
    // is not assignable
}
```

- Функция не является объектом
  - нельзя передать по значению в другую функцию
  - нельзя вернуть из другой фукнции
  - нельзя создать массив функций
- Тип функции состоит из типа возвращаемого значения и типов параметров (после array-to-pointer, function-to-pointer преобразований), [`noexcept (since c++17)`]*

Пример 1

```
int f(int) {
    return 42;
}

using F = int(int);

int g(F func) {
    return func(42);
}

int main() {
    std::cout << g(f);

    g = 13; // <source>:17:7: error: non-object type 'int (F *)'
    // (aka 'int (int (*)(int))') is not assignable
}
```

Пример 2

```
int f(int) {
    return 42;
}

using F = int(int);

// error: function cannot return function type 'F' (aka 'int (int)')
F g() {
    return f;
}
```

Пример 3

```
int f(int) {
    return 42;
}

using F = int(int);

int main() {
    // error: 'f' declared as array of
    // functions of type 'F' (aka 'int (int)')
    F f[10];

    // error: cannot allocate function
    // type 'F' (aka 'int (int)') with new
    F* fp = new F[10];
}
```

Пример 4

```
using F = int(int);

void g(F f, int a[100], int b[10][11]) {

}

int main() {
    // error: non-object type 'void (
    //    F *,
    //    int *,
    //    int (*)[11])'
    // (aka 'void (int (*)(int), int *, int (*)[11])') is not assignable
    g = 10;
}
```

declaration & definition

declaration — представляет имя и тип функции

definition — связывает имя и тип с телом

declaration

- Может быть в любой области видимости

Syntax (*simplified*):

```
decl-specifier-seq init-declarator-list;       // Simple declaration
noptr-declarator ( parameter-list )            // Declarator
```

- `decl-specifier-seq` содержит возвращаемый тип, может включать в себя `static`, `inline`, `constexpr`
- `init-declarator-list` — список деклараторов [с инициализаторами]
- `noptr-declarator` содержит имя
- `parameter-list` — список параметров (возможно пустой)

declaration

```
int main() {
    int value = 1;
    int v = value, *pointer = &value, func(), (*pFunc)(int);
}
```

Указатель на функцию

`noptr-declarator` имеет вид:

```
(*name)
```

```
void f() {}

int main() {
   void(*ptr)() = nullptr;
   ptr = f;
   ptr();
}
```

Parameter list

Параметры разделяются запятой, каждый из параметров имеет следующий синтакис:

```
decl-specifier-seq declarator [= initializer]
decl-specifier-seq abstract-declarator [= initializer]
```

```
// объявление одной и той же функции:
void f(void);
void f();
```

Ellipsis

```cpp
#include <cstdarg>

int add_nums(int count, ...)
{
    int result = 0;
    std::va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        result += va_arg(args, int);
    }
    va_end(args);
    return result;
}
```

- В конце списка параметров
- Наименее приоритетный вариант при перегрузке
- Нет проверки типов
- count нельзя передавать по ссылке (UB)
- Аккуратное обращение с va_-макросами (UB)

Variadic function documentation: [Click me]

Parameter list

Правила для определения типа параметра

1. Из `decl-specifier-seq` и `declarator` формируется тип
2. Тип массива (bound/unbound) преобразуются к указателю
3. Тип функции преобразуется к указателю
4. Отбрасывается const верхнего уровня (касается только типа функции, в теле const остается)

Function definition

Определения [non-member]* функций встречается только в блоках с [пространствами имен]*.

Syntax:

```
decl-specifier-seq declarator function-body
```

Где `function-body` — `compound-statement` или `function-try-block`

Тип возвращаемого значения и типы параметров не могут быть `incomplete`.

Пространства имен и member-фукнции будут разобраны позже

Имя функции

```
void fun() {
    std::cout << __func__ << std::endl;
}
void fun(int) {
    std::cout << __func__ << std::endl;
}
int main() {
    fun();
    fun(1);
}
```

Default arguments

Позволяют вызвать функцию не передавая часть аргументов.

Syntax:

```
decl-specifier-seq declarator = initializer              (1)
decl-specifier-seq [abstract-declarator] = initializer   (2)
```

```
void g(int a = 1, int * = nullptr);
```

Default arguments

Значения по умолчанию для параметров функции, находящихся в объявлении правее параметра с значением по умолчанию:

- либо заданы в текущем объявлении
- либо заданы в одном из предыдущих объявлений

```
#include <iostream>
int sum(int i, int j, int k = 1);
int sum(int i, int j = 5, int k);
int sum(int i = 1, int j, int k);
int sum(int i, int j, int k) {
    return i+j+k;
}

int main() {
    std::cout << sum();
}
```

Inline function

- Определение фукнции должно быть доступно в единице трансляции, в которой она используется
- inline функция с внешней линковкой:
  - может иметь более одного определения в программе (не более одного в TU)
  - `inline` должен быть у всех определений функций во всех TU
  - функция имеет один и тот же адрес в разных TU

TU — translation unit

Overloading (перегрузка)

```
#include <iostream>

int sum(int i) { return i; }                // (1)
int sum(int i, int j) { return i + j; }     // (2)

int main() {
    std::cout << sum(1) << std::endl;       // call (1)
    std::cout << sum(1, 2) << std::endl;    // call (2)
}
```

Адрес перегруженной функции

Имя функции (помимо call expression) может быть использовано в следующих случаях:

- инициализация указателя/ссылки
- присваивание (правый операнд)
- как аргумент функции [или user-defined оператора]*
- `return`-statement
- `static_cast`

Выбор перегруженной фукнции

Копиляция кода с вызовом функции:

- name lookup: ([ADL]*, [Template argument deduction]*). В результате получаем множество сущностей.
- если сущностей более одной, то выполняется `overload resolution` (выбор самой подходящей)

1. Viable functions

выбор жизнеспособных(viable) функций:

- допустимое количество аргументов
- присутствует неявное преобразование каждого аргумента к типу соответствующего параметра

2. Best Viable function

Для каждой пары функций F1 и F2, последовательно проверяются неявные преобразования типов аргументов к типам параметров.

Если хотя бы одно преобразование аргумента к типу из F1 лучше, чем преобразование того же аргумента к типу из F2, то выбирается F1.

Language Linkage

Взаимодействие кода, написанного на разных языках (`C`, `C++`).

Syntax:

```
extern string-literal { [declaration-seq] }
extern string-literal declaration
```

`string-literal` — имя языка: `"C"`, `"C++"`

Применяет языковую спецификацию к типам функций (calling convention), именам функций и переменных (name mangling) с внешней линковкой

Language Linkage

```
#include <iostream>

extern "C" {
    int c_function(int); // c-function declaration
}

int main() {
    std::cout << c_function(33); // call c-function from c++
}
```

Linkage

```cpp
// a.cpp

// external linkage (символы видны вне единицы трансляции)
void f() { }
int i;
extern const int j = 300; // force extern for const

// internal linkage (символы видны внутри единицы трансляции)
static void g() { }
const int ci = 1; // by default
static int si = 2;
```

```
$ clang++ -c a.cpp
$ nm a.o
0000000000000000 T __Z1fv
0000000000000070 S _i
0000000000000008 S _j
```

Language Linkage

```
extern "C" {
    int func(int i) { std::cout << i; return i; }
}

extern "C" int func2(int j) { return j; }
```

Эти функции можно использовать в C-шном коде.

Name mangling

```cpp
// a.cpp
extern "C" int g(int) { }
extern "C++" int f(int) { }

int main() {}
```

```
nm a.out
```

```
0000000100003f70 T __Z1fi
0000000100003f60 T _g
0000000100003f80 T _main
```

Headers

Заголовочные файлы, используемые с C и C++ коде

```
// a.h
#ifdef __cplusplus
extern "C" {
#endif

void f(int);
void g(double);

#ifdef __cplusplus
}
#endif
```

Additional I (Conversions)

Implicit conversions

```
bool b = 10;
int a = b;
b = a;

char x = 'x';
char y = 'y';
auto z = x + y;
```

static-cast

```
void process(void* int_data, size_t sz) {
    // const int* p = int_data;
    // cannot initialize a variable of type 'const int *'
    // with an lvalue of type 'void *'

    const int* p = static_cast<int*>(int_data);
}
```

```
void process(const int* int_data, size_t sz) {
    unsigned char* p
        = static_cast<unsigned char*>(int_data);
    // error: static_cast from 'const int *'
    // to 'unsigned char *' is not allowed
}
```

reinterpret-cast

```
int a = 10'009'100;

unsigned char* p = reinterpret_cast<unsigned char*>(&a);
for (size_t i = 0; i < sizeof(a); ++i) {
    std::cout << (int)p[i] << "-";
}
```

c-style cast (DO NOT USE)

Последовательно пробует сделать преобразования в следующем порядке

- `const_cast`
- `static_cast`
- `reinterpret_cast`

Additional II

Comma operator

Form:

```
expr1, expr2
```

- `expr1` вычисляется, а результат отбрасывается
- результат `expr2` будет результатом выражения
- не путать с запятой, используемой для перечисления аргументов функции или для перечисления элементов списка инициализации

Comma operator

Почему возникает ошибка компиляции?

```
int m = 1, 2, 3; // compile-time error
```

Conditional operator

Form:

```
expr1 ? expr2 : expr3
```

- `expr1` вычисляется и результат контекстно приводится к типу `bool`
- в случае `true` вычисляется второй операнд
- в случае `false` вычисляется третий операнд