C++

Fall 2022

compscicenter.ru

Башарин Егор

https://t.me/egorbasharin

Лекция IV

Functions

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

- `F` — выражение, результат которого:
  - функция / ссылка на функцию
  - указатель на функцию (см. слайды далее)
  - [вызов метода класса]*
- `Arg1, Arg2, …` — список выражений [или списков инициализации]. Может быть пустым.

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

Если F - фукнция [или метод], то допускается перегрузка F

```
#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)
}
```

Выбирается та функция, чей набор параметров наиболее подходящий.

Function call operator

Порядок вычисления выражений

Form:

```
F(Arg1, Arg2, ...)
```

`F`, `Arg1`, `Arg2` представляют собой выражения, которые вычисляются:

- В любом порядке(`until C++17`)
- Сначала `F`, затем все остальное в любом порядке(`since C++17`)

Function call operator

Form:

```
F(Arg1, Arg2, ...)
```

Тип выражения — тип, возвращаемый фукнцией.

Функция

```
size_t fact(size_t n)
{
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Сущность, связывающая последовательность утверждений с именем и набором параметров.

При вызове фунцкии ее параметры инициализируются аргументами, после чего выполняется тело функции.

Способы прерывания функции

- `return` statement
- `throw`-expression

throw, try-catch (basic)

Обработка исключительных ситуаций

Syntax:

```
throw expression;  // (1)
throw;             // (2)
```

```
#include <iostream>
#include <stdexcept> // take error-classes here
void g() {
    throw std::logic_error("msg");
}
void f() {
    try {
        g();
    } catch (const std::logic_error& ex) {
        std::cout << "handle exception: " << ex.what() <
        throw; // rethrow
    }
}
```

**Слайд для получения базового представления об исключениях, подробнее рассмотрим эту тему в дальнейших лекциях*

Тип функции

```
void z(int i, double j) {}
int main() {
    z = 10;
    // error: non-object type 'void (int, double)' is no
}
```

- 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`)]*

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)

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 in Parameter list

```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;
}
```

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)

```cpp
#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.

Ранги преобразований:

- Exact match: no conversion required, lvalue-to-rvalue conversion, qualification conversion
- Promotion: integral promotion, floating-point promotion
- Conversion: integral conversion, floating-point conversion, floating-integral conversion,…

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
}
```

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
```

Calling Conventions

В зависимости от соглашении о вызове определяется:

- способ передачи аргументов: регистры и/или стек
- порядок размещения аргументов в регистрах/стеке
- ответственный за очистку стека: callee/caller
- способ передачи результата в точку вызова
- способы возврата (передачи управления) в точку вызова

Wiki: Click me

32-bit x86 calling conventions

- cdecl — RTL передача параметров, вызывающая функция очищает стек, имена функций начинаются с знака подчеркивания _
- stdcall - Win32, RTL передача параметров, вызываемая функция чистит стек, mangling: `_func@12`
- fastcall
- thiscall

cdecl

```
int sum(int i, int j) {
    return i + j;
}

int main() {
    int res = sum(2, 3);
}
```

```
// Caller
push 3
push 2
call _Z3sumii
add esp, 8
mov dword ptr [res],eax
```

```
// Callee
push ebp
mov ebp, esp
mov eax, dword ptr [ebp + 8]
add eax, dword ptr [ebp + 12]
mov esp, ebp
pop ebp
ret
```

Additional I

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