

## Функции

```
int rollDice()  
{  
    return 4;  
}  
  
int square(int x)  
{  
    return x * x;  
}
```

### Конвенции вызова x32

#### **cdecl**

Исторически принятое соглашение для языка C.

Аргументы функций передаются через стек, справа налево. Аргументы, размер которых меньше 4-х байт, расширяются до 4-х байт. Очистку стека производит вызывающая программа. integer-like результат возвращается через регистр EAX.

Перед вызовом функции вставляется код, называемый прологом (prolog) и выполняющий следующие действия:

- сохранение значений регистров, используемых внутри функции
- запись в стек аргументов функции

После вызова функции вставляется код, называемый эпилогом (epilog) и выполняющий следующие действия:

- восстановление значений регистров, сохранённых кодом пролога
- очистка стека (от локальных переменных функции)

#### **thiscall**

Соглашение о вызовах, используемое компиляторами для языка C++ при вызове методов классов.

Отличается от **cdecl** соглашения только тем, что указатель на объект, для которого вызывается метод (указатель this), записывается в регистр есх.

#### **fastcall**

Передача параметров через регистры: если для сохранения всех параметров и промежуточных результатов регистров не достаточно, используется стек (в gcc через регистры есх и едх передаются первые 2 параметра).

### Смотрим сгенерированный код

```
[[gnu::fastcall]]
void foo1(int x, int y, int z, int a)
{
}

void foo2(int x, int y, int z, int a)
{
}

void bar1()
{
    foo1(1, 2, 3, 4);
}

void bar2()
{
    foo2(5, 6, 7, 8);
}
```

```
g++ -c test.cpp -o test.o -O0 -m32
```

```
objdump -d test.o
```

```
000005c8 <_Z4bar1v>:
5c8: 6a 04          push    $0x4
5ca: 6a 03          push    $0x3
5cc: ba 02 00 00 00 mov     $0x2,%edx
5d1: b9 01 00 00 00 mov     $0x1,%ecx
5d6: e8 b5 ff ff ff call    590 <_Z4foo1iiii>
5dd: c3            ret

000005eb <_Z4bar2v>:
5eb: 6a 08          push    $0x8
5ed: 6a 07          push    $0x7
5ef: 6a 06          push    $0x6
5f1: 6a 05          push    $0x5
5f3: e8 b3 ff ff ff call    5ab <_Z4foo2iiii>
5fd: c3            ret
```

#### System V AMD64 ABI (Linux, MacOS, FreeBSD, Solaris)

- 6 регистров (RDI, RSI, RDX, RCX, R8, R9) для передачи integer-like аргументов
- 8 регистров (XMM0-XMM7) для передачи double/float
- если аргументов больше, они передаются через стек
- для возврата integer-like значений используются RAX и RDX (64 бит + 64 бит)

### Встраивание функций (inline)

Иногда вызова функции не будет - оптимизирующий компилятор выполнит ее встраивание по месту вызова.

Можно подсказать компилятору выполнить встраивание:

```
inline void foo()
{
}
```

Но, компилятор умный и скорее всего проигнорирует inline, но можно попросить настойчивей:

```
// ms vc
__forceinline void foo()
{
}
```

```
// gcc
__attribute__((always_inline)) void foo()
{
}
```

Все равно нет гарантий.

**Тот случай, когда макросы уместны**

```
#ifdef __GNUC__
#define __forceinline __attribute__((always_inline))
#endif
```

## Ссылки

Ссылка - псевдоним объекта.

Главное отличие от указателя - ссылка должна быть проинициализирована при объявлении и до конца своей жизни ссылается только на один объект.

```
int a = 1;
int b = 2;
int* ptr = nullptr;
ptr = &a;
ptr = &b;
int& ref; // Ошибка
int& ref = a; // ref ссылается на a
ref = 5; // Теперь a == 5
ref = b; // ref не стала указывать на b,
          // мы просто скопировали значение из b в a
ref = 7; // a == 7, b == 2
```

```
int a = 2;
int* ptr = nullptr;
int*& ptrRef = ptr; // ptrRef ссылается на ptr
ptrRef = &a; // теперь ptr хранит адрес a
```

## Передача аргументов в функции

**По значению**

```

void foo(int x)
{
    x = 3;
}

int x = 1;
foo(x);
// x == 1

void bar(BigObject o) { ... }

```

- В функции окажется копия объекта, ее изменение не отразится на оригинальном объекте
- Копировать большие объекты может оказаться накладно

#### По ссылке

```

void foo(int& x)
{
    x = 3;
}

int x = 1;
foo(x);
// x == 3

void bar(BigObject& o) { ... }

```

- Копирования не происходит, все изменения объекта внутри функции отражаются на объекте
- Следует использовать, если надо изменить объект внутри функции

```

void swap(int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

```

#### По константной ссылке

```

void foo(const int& x)
{
    x = 3; // ошибка компиляции
}

void bar(const BigObject& o) { ... }

```

- Копирования не происходит, при попытке изменения объекта будет ошибка
- Большие объекты выгодней передавать по ссылке, маленькие - наоборот

#### По указателю

```

void foo(int* x)
{
    *x = 3;
}

void bar(BigObject* o) { ... }

void foo(const int* x)
{
    *x = 3; // ошибка компиляции
}

void bar(const BigObject* o) { ... }

```

- Копирования не происходит
- Если указатель на константный объект, то при попытке изменения объекта будет ошибка
- Есть дополнительный уровень косвенности, возможно придется что-то подгрузить в кеш из дальнего участка памяти
- Реализуется optional-концепция

```

int countObject(time_t* fromDate, time_t* toDate)
{
    const auto begin =
        fromDate == nullptr
            ? objects_.begin()
            : objects_.findFirst(fromDate);
}

```

По универсальной ссылке

```

void foo(int&& x) { ... }
void bar(BigObject&& o) { ... }

```

Поговорим в отдельной лекции.

Перегрузка функций

```

void print(int x) // 1
{
    std::cout << x << std::endl;
}

void print(bool x) // 2
{
    std::cout << (x ? "true" : "false") << std::endl;
}

print(10); // 1
print(true); // 2

```

Опасность перегрузки

```
void print(const std::string& x) // 3
{
    std::cout << "string" << std::endl;
}

print("hello!"); // 2 const char* приводится к bool
```

Перегруженная функция - декорированная функция

## Пространства имен

Проблема:

```
// math.h
double cos(double x);
```

```
// ваш код
double cos(double x);
```

Решение в стиле C:

```
// ваш код
double fastCos(double x);
```

Решение:

```
namespace fast
{
    double cos(double x);
}

fast::cos(x);
cos(x); // вызов из math.h
```

## Поиск имен

- Проверка в текущем namespace
- Если имени нет и текущий namespace глобальный - ошибка
- Рекурсивно поиск в родительском namespace

```

void foo() {} // ::foo

namespace A
{
    void foo() {} // A::foo

    namespace B
    {
        void bar() // A::B::foo
        {
            foo(); // A::foo
            ::foo(); // foo()
        }
    }
}

```

### Ключевое слово using

Добавляет имена из указанного namespace в текущий namespace.

```

void foo()
{
    using namespace A;
    // видимо все из A
}

```

```

void foo()
{
    using namespace A::foo;
    // видима только A::foo()
}

```

```

void foo()
{
    namespace ab = A::B;
    ab::bar(); // A::B::bar()
}

```

**using может приводить к проблемам**

```

using namespace fast;

cos(x); // ???
cos(x); // ???

```

Не используйте using namespace в заголовочных файлах!

### Указатель на функцию

```
void foo(int x)
{
}

typedef void (*FooPtr)(int);

FooPtr ptr = foo;
ptr(5);
```

```
using FooPtr = void (*)(int);
```

### Функции высшего порядка

Функция высшего порядка — функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

### Сценарии использования указателей на функции

#### Настройка поведения

```
void sort(int* data, size_t size, bool (*compare)(int x, int y));

bool less(int x, int y)
{
    return x < y;
}

sort(data, 100, less);
```

#### Функции обратного вызова (callback)

```
using OnMailReceived = void (*)(const Mail& newMail);

void addOnMailReceivedObserver(OnMailReceived handler);

void onMailReceivedHandler(const Mail& newMail)
{
    ...
}

addOnMailReceivedObserver(onMailReceivedHandler);
```

### Конвейеры



```

using MoveFunctionPtr = void (*)(int& x, int& y);

void moveLeft(int& x, int& y) { ... }
void moveRight(int& x, int& y) { ... }

std::vector<MoveFunctionPtr> trajectory =
{
    moveLeft,
    moveLeft,
    moveRight,
};

int x = 0;
int y = 0;
for (auto& func : trajectory)
{
    func(x, y);
}

```

## Лямбда-функции

```

auto lessThen3 = [](int x) { return x < 3; };

if (lessThen3(x)) { ... }

```

### Синтаксис

```
[список_захвата](список_параметров) { тело_функции }
```

```
[список_захвата](список_параметров) -> тип_возвращаемого_значения
{ тело_функции }
```

### Захват переменных

```

int x = 5;
int y = 7;
auto foo = [x, &y]() { y = 2 * x };
foo();

```

Если не указать &, то будет захват по значению, то есть копирование объекта; если указать, то по ссылке (нет копирования, модификации внутри функции отразятся на оригинальном объекте).

```

// Захват всех переменных в области видимости по значению
auto foo = [=]() {};

```

```

// Захват всех переменных в области видимости по ссылке
auto foo = [&]() {};

```

Использование переменных, определённых в той же области видимости, что и лямбда-функция, называют замыканием.

## Примеры захвата

```
[ ] // без захвата переменных из внешней области видимости
[=] // все переменные захватываются по значению
[&] // все переменные захватываются по ссылке
[x, y] // захват x и y по значению
[&x, &y] // захват x и y по ссылке
[in, &out] // захват in по значению, а out – по ссылке
[=, &out1, &out2] // захват всех переменных по значению,
// кроме out1 и out2, которые захватываются по ссылке
[&, x, &y] // захват всех переменных по ссылке, кроме x,
// которая захватывается по значению
```

## mutable

```
int x = 3;
auto foo = [x]() mutable
{
    x += 3;
    ...
}
```

## std::function

```
#include <functional>

using MoveFunction =
    std::function<void (int& x, int& y)>;

MoveFunction getRandomDirection() { ... }

std::vector<MoveFunction> trajectory =
{
    moveLeft,
    moveLeft,
    moveRight,
    [](int& x, int& y)
    {
        ...
    },
    moveLeft,
    getRandomDirection()
};

int x = 0;
int y = 0;
for (auto& func : trajectory)
{
    func(x, y);
}
```

## Как выделить память в C++

```
char* data = (char*) malloc(1024);  
...  
free(data);
```

```
std::unique_ptr<char[]> data(new char[1024]);  
// или так  
auto data = std::make_unique<char[]>(1024);
```

```
char* ptr = data.get() // Указатель  
char x = data[100]; // Элемент 100  
data.reset(); // Явное освобождение памяти
```

## Требования к домашним заданиям

1. В вашем github должен быть репозиторий msu\_cpp\_autumn\_2019
2. Внутри репозитория должны быть директории из двух цифр, вида: 01, 02 и т.д. - это номера домашних заданий
3. Внутри каждой директории могут быть любые файлы реализующие задачу. Обязательным является только файл Makefile
4. В Makefile обязательно должны быть цели test и run. run запускает бинарный файл с домашним заданием, test запускает тесты вашего решения
5. Собираться ваш код должен компилятором C++ поддерживающим 14 стандарт
6. Внешних зависимостей быть не должно

После выполнения домашнего задания присылайте Владиславу Смирнову (vladislav.smirnov@corp.mail.ru) письмо с темой made\_2019\_cpp\_номердз

## Практическая часть

Используя метод рекурсивного спуска, написать калькулятор. Поддерживаемые операции:

- умножение
- деление
- сложение
- вычитание
- унарный минус

Для вычислений использовать тип int, приоритет операций стандартный. Передача выражения осуществляется через аргумент командной строки, поступающие числа целые, результат выводится в cout. Пример:

```
calc "2 + 3 * 4 - -2"
```

Вывод:

```
16
```

Должна быть обработка ошибок, в случае некорректного выражения выводить в консоль ошибку и возвращать код отличный от 0. Тесты обязательны.

EOF