C++ Introduction

CODERS SCHOOL

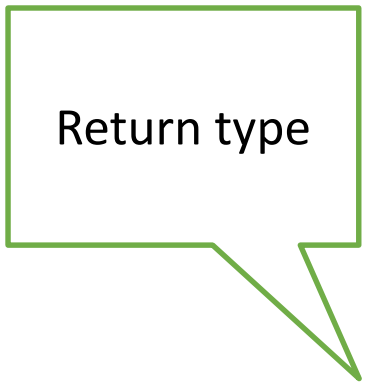
LEVEL UP BEFORE YOU START

Ihor Rudynskyi

Fundamental knowledge

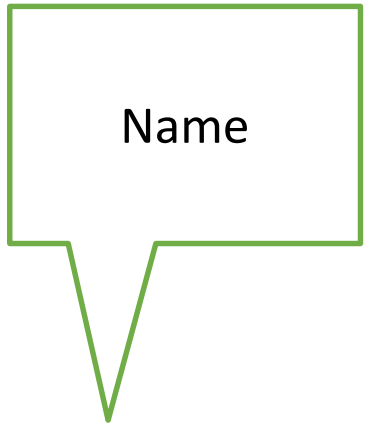
```
int main(int argc, char** argv)
{
    return 0;
}
```

Fundamental knowledge



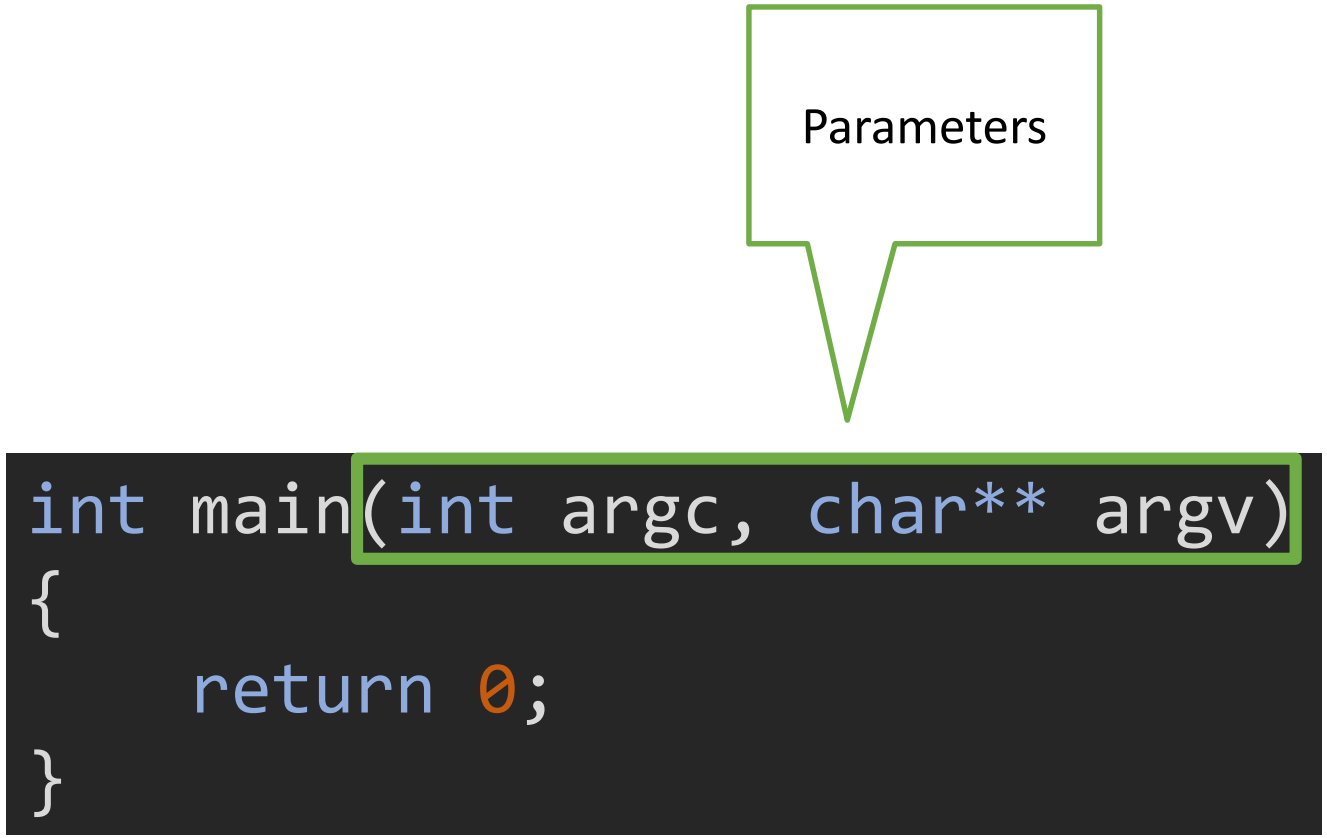

```
int main(int argc, char** argv)
{
    return 0;
}
```

Fundamental knowledge



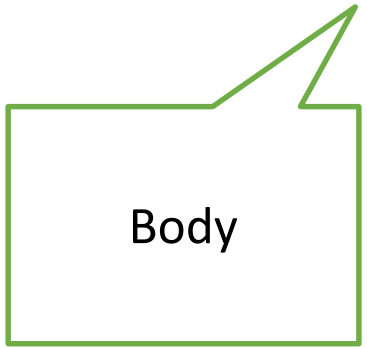

```
int main(int argc, char** argv)
{
    return 0;
}
```

Fundamental knowledge



Fundamental knowledge

```
int main(int argc, char** argv)
{
    return 0;
}
```



Fundamental types

https://en.cppreference.com/w/cpp/language/types
https://en.cppreference.com/w/cpp/types/integer

Fundamental types

- bool - type, capable of holding one of the two values: true or false.
- int - basic integer type.
- char - type for character representation
- float - single precision floating point type.
- double - double precision floating point type.
- void - type with an empty set of values.
- std::size_t - can store the maximum size of a theoretically possible object of any type (including array).

Quick guide

- integer – int
- unsigned integer – unsigned int
- n width integer – int[n]_t
- character - char
- floating point type - double
- length/size/hash - std::size_t

check cppreference

```
template <class T, std::size_t N>
constexpr std::size_t size(const T (&array)[N]) noexcept;
```

Ask the compiler for help
[-Wconversion]

Home work (fix it)

```cpp
bool palindrom(std::string napis)
{
    int dlugosc = napis.size();
    for(int i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```

```
prog.cc: In function 'bool palindrom(std::string)':
prog.cc:7:31: warning: conversion from 'std::__cxx11::basic_string<char>::size_type' {aka 'long unsigned int'}
to 'int' may change value [-Wconversion]
   7 |     int dlugosc = napis.size();
     |                   ~~~~~~~~~~~^~
```

Home work

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```

Count it!

Count elements that are equal to value

```
std::size_t count(int value, std::vector<int> numbers)
{
    std::size_t N = 0;
    for(auto number : numbers)
    {
        if(number == value)
            ++N;
    }
    return N;
}
```

Count it!

Count elements that are equal to value

```
std::size_t count(std::string file_name, std::string path)
{
    if(is_file(path))
    {
        return file_name(path) == file_name;
    }

    std::size_t N = 0;
    for(auto member_path : dir_members(path))
    {
        N += count(file_name, member_path);
    }
    return N;
}
```

Home work (use recursion)

```cpp
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```

Home work

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
        && palindrom(napis.substr(1, napis.size() - 2));
}
```

Recursion? Mehhh

```
std::size_t find_index(int value, std::vector<int> numbers)
{
    for(std::size_t i = 0; i < numbers.size(); ++i)
    {
        if(numbers[i] == value)
            return i;
    }
    return -1;
}
```

Recursion? Mehhh

```
std::size_t find_index_impl(int i, int value, std::vector<int> numbers)
{
    if(numbers.empty())
        return -1;
    if(number.front() == value)
        return i;

    numbers.pop_front();
    return find_index_impl(i + 1, value, numbers);
}

std::size_t find_index(int value, std::vector<int> numbers)
{
    return find_index_impl(0, value, numbers);
}
```

Stack and heap

Stack

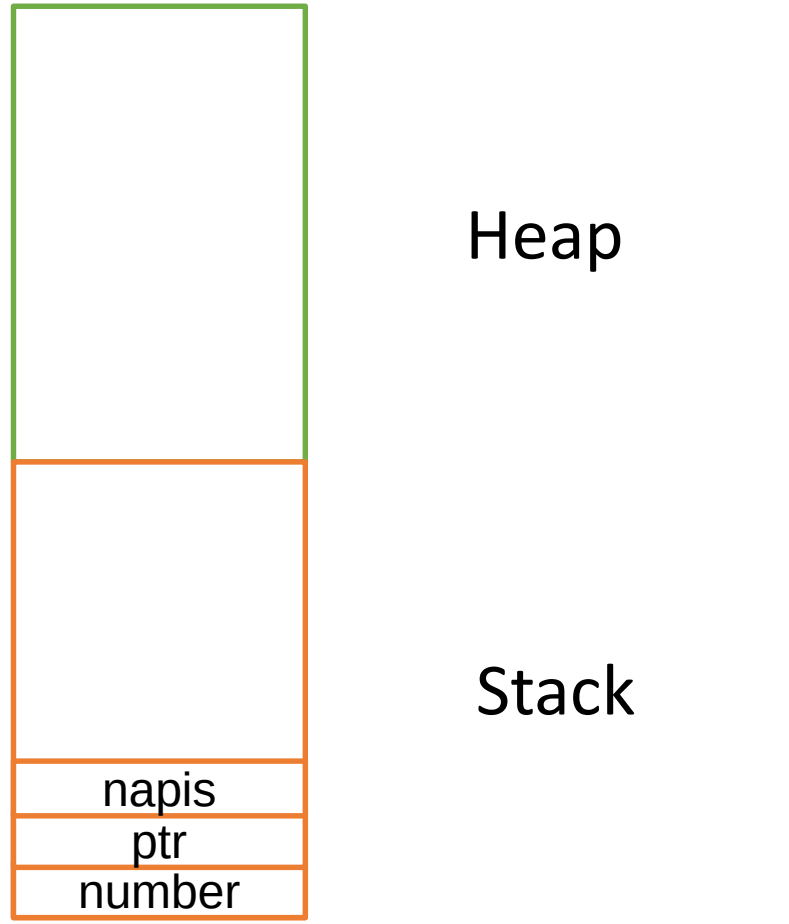
- the stack grows and shrinks as functions push and pop local variables
- there is no need to manage the memory yourself, variables are allocated and freed automatically
- the stack has size limits
- stack variables only exist while the function that created them, is running

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)

Stack and heap

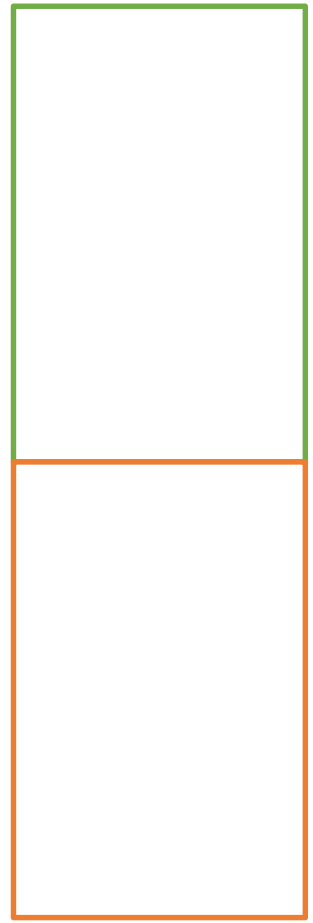
```
{
    int number;
    int* ptr;
    std::string napis;
}
```



- there is no need to manage the memory yourself, variables are allocated and freed automatically

Stack and heap

```
{
    int number;
    int* ptr;
    std::string napis;
}

{
    int* ptr = new int;
    std::string napis = “hi”;
}
```



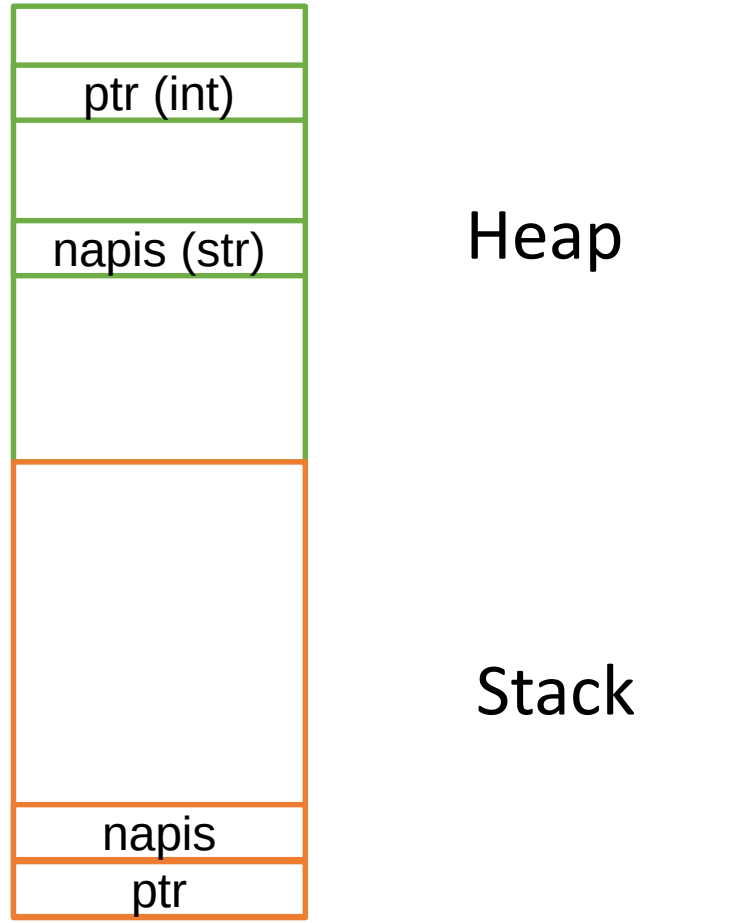
Heap

Stack

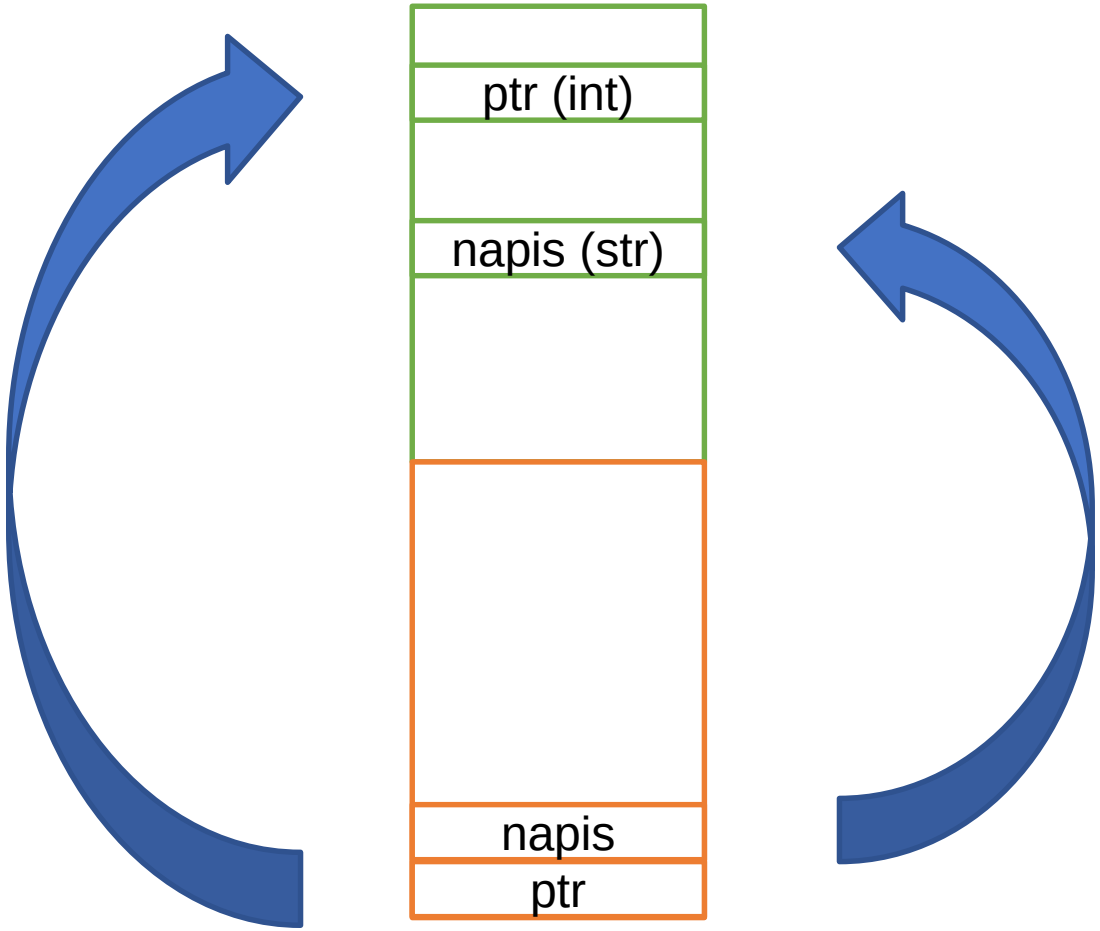
- there is no need to manage the memory yourself, variables are allocated and freed automatically

Stack and heap

```
{
    int number;
    int* ptr;
    std::string napis;
}

{
    int* ptr = new int;
    std::string napis = “hi”;
}
```

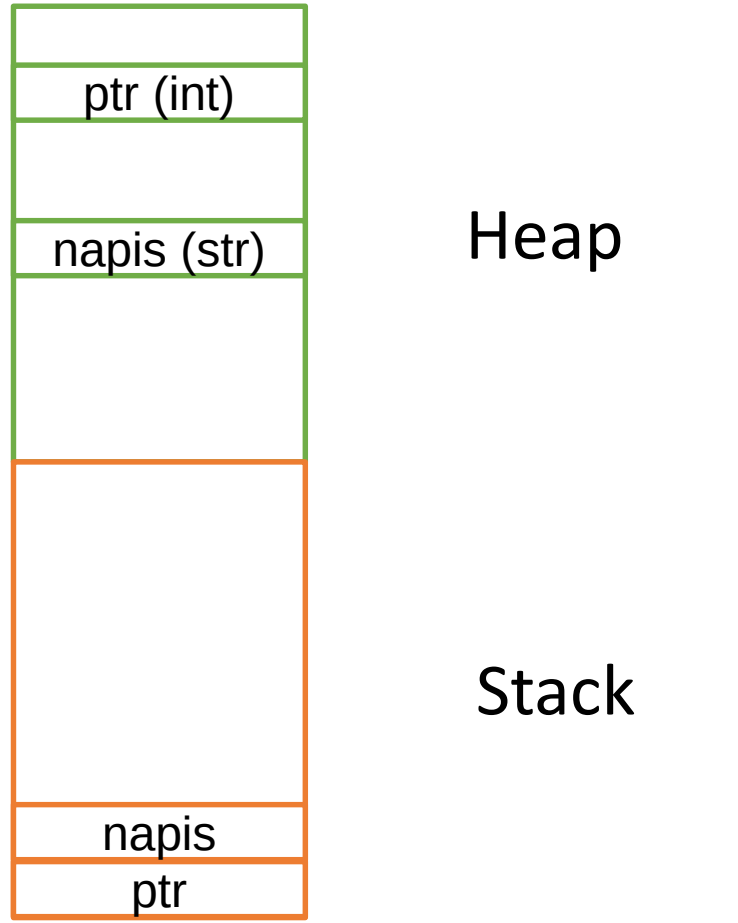


Pointer concept



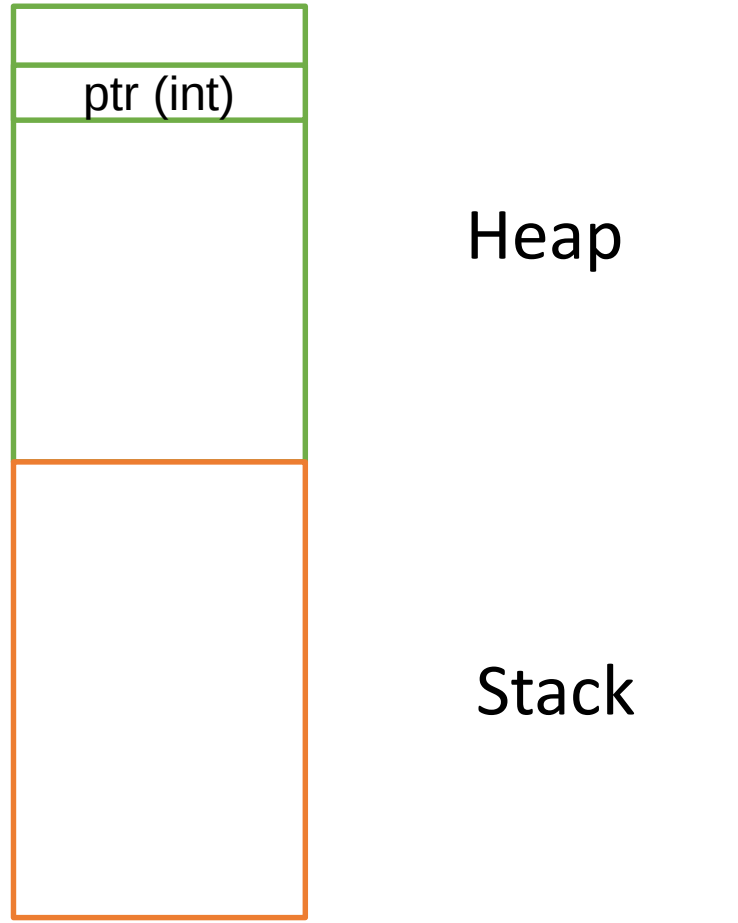
Stack and heap

```
{
    int number;
    int* ptr;
    std::string napis;
}

{
    int* ptr = new int;
    std::string napis = “hi”;
}
```



Stack and heap

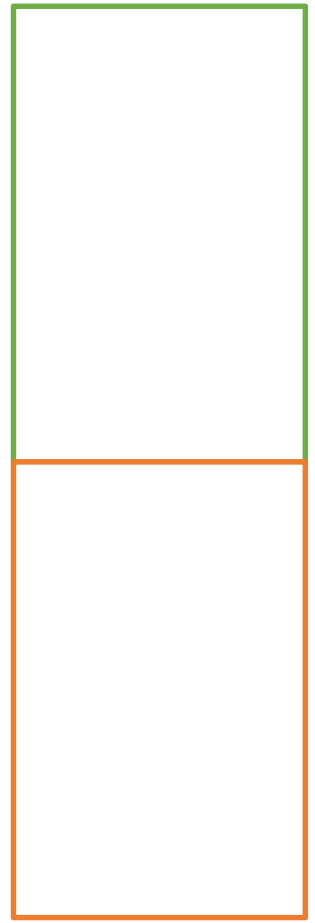
```
{
    int number;
    int* ptr;
    std::string napis;
}

{
    int* ptr = new int;
    std::string napis = “hi”;
}
```



- you must manage memory (you're in charge of allocating and freeing variables)

Stack and heap

```
{
    int number;
    int* ptr;
    std::string napis;
}

{
    int* ptr = new int;
    std::string napis = “hi”;
    delete ptr;
}
```



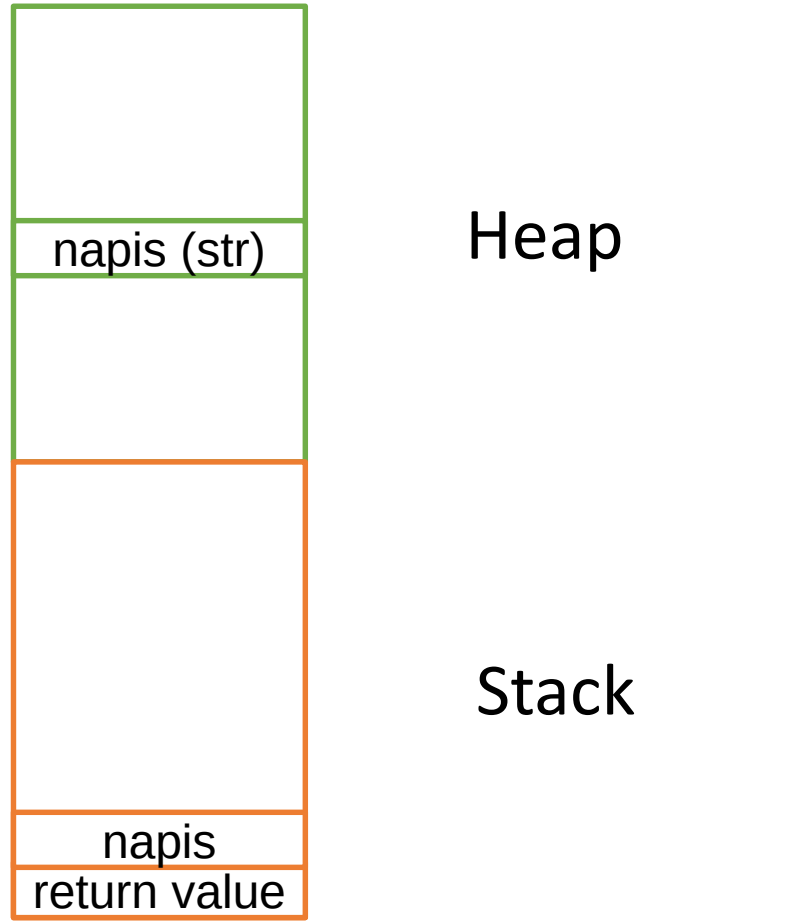
Heap

Stack

- you must manage memory (you're in charge of allocating and freeing variables)

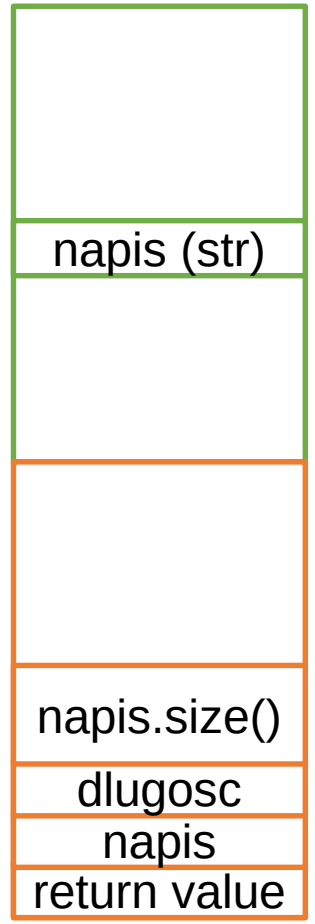
Stack and heap

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```



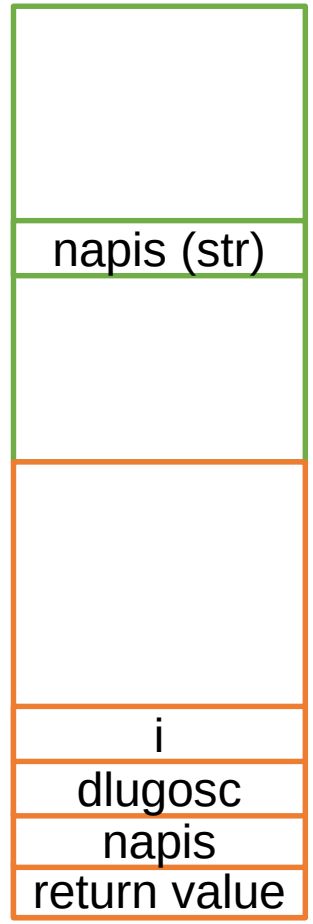
Stack and heap

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```



Stack and heap

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```

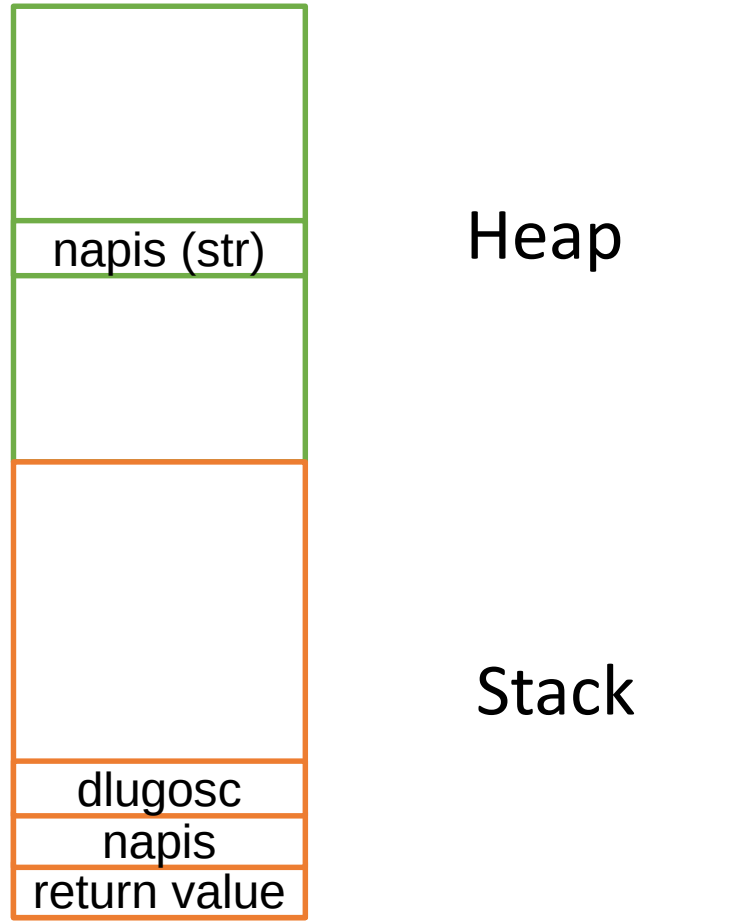


Heap

Stack

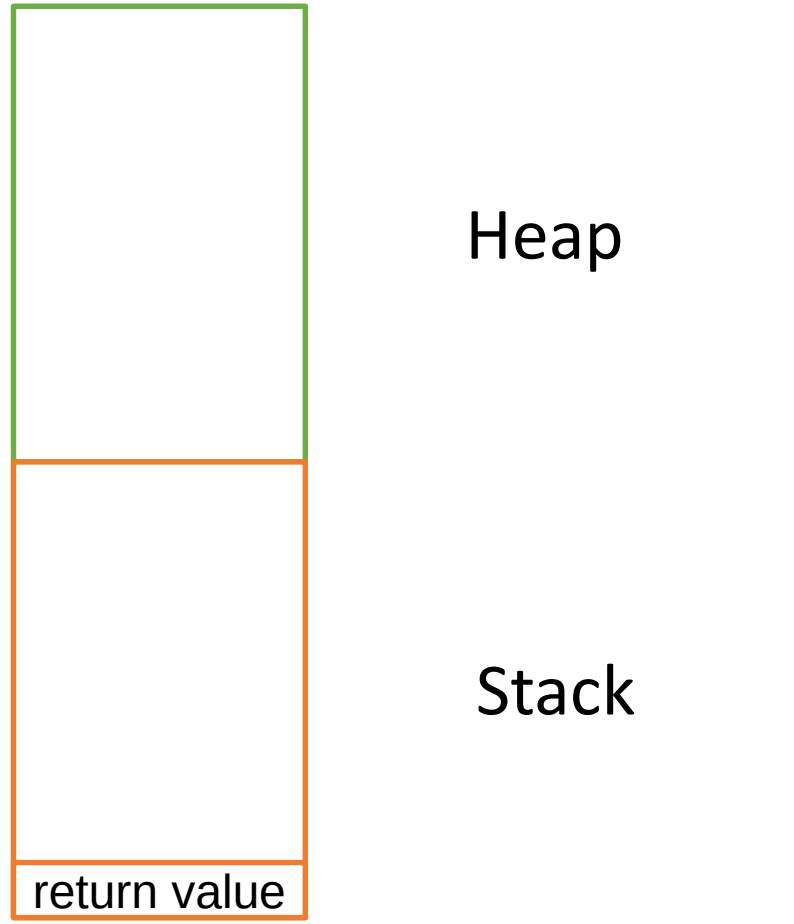
Stack and heap

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```



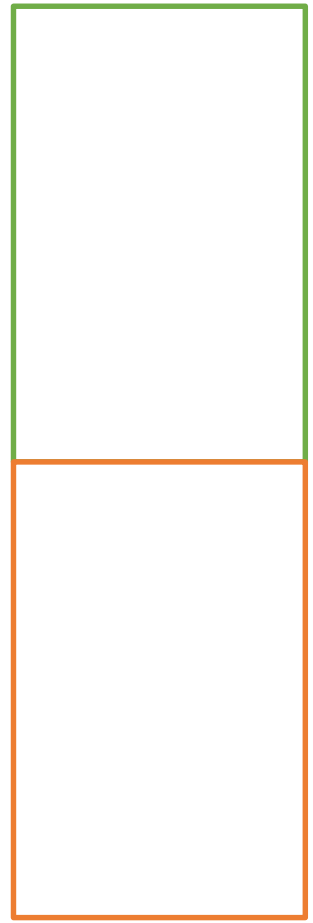
Stack and heap

```
bool palindrom(std::string napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```



Home work

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```

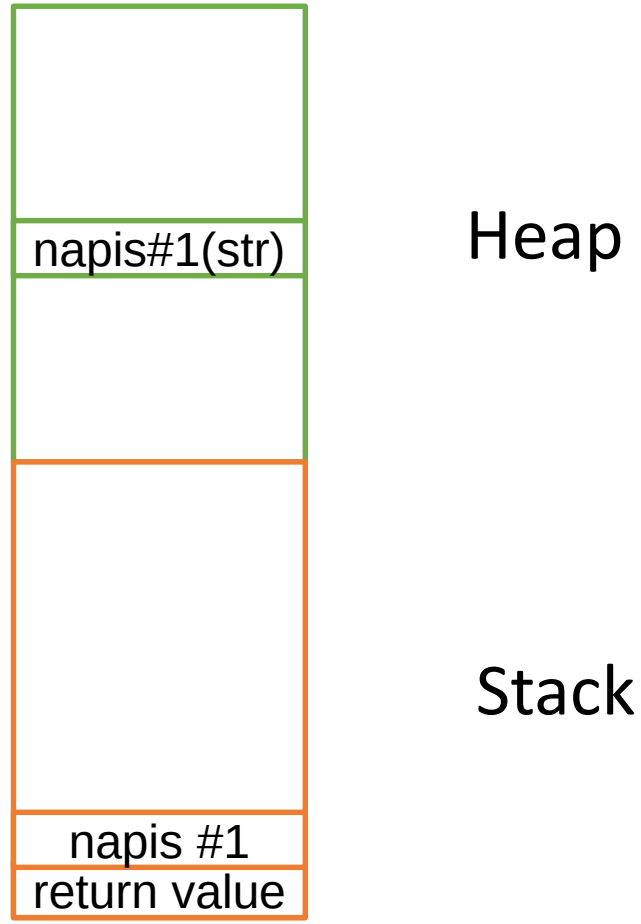


Heap

Stack

Home work

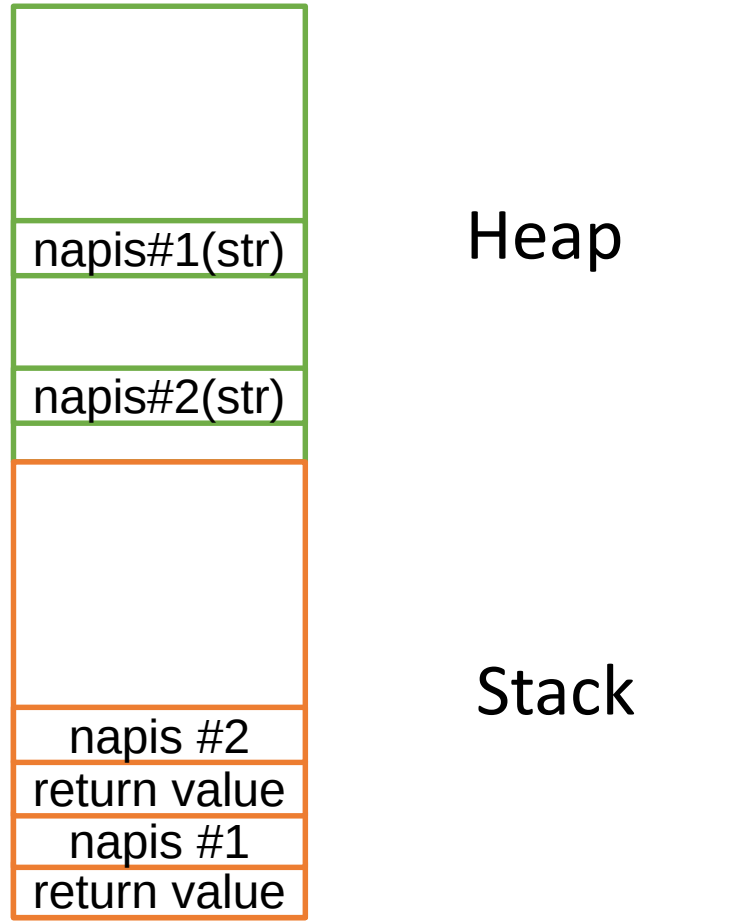
```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```



Home work

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
 && palindrom(napis.substr(1, napis.size() - 2));
}
```

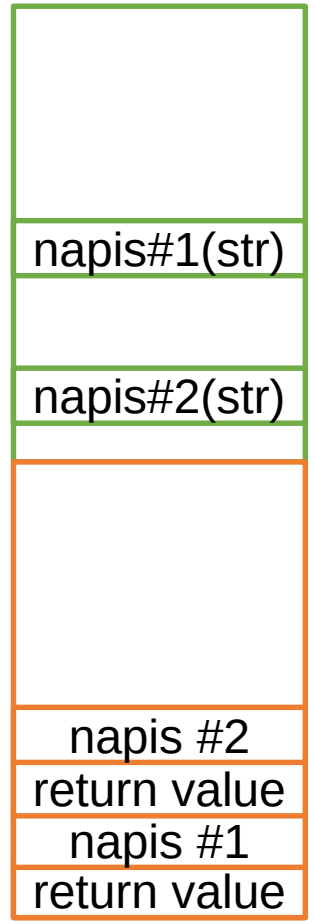
```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```



Home work

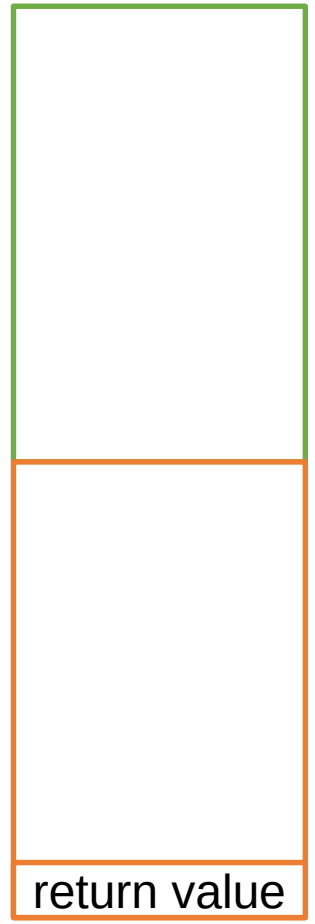
```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
&& palindrom(napis.substr(1, napis.size() - 2));
}
```

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```



Home work

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```



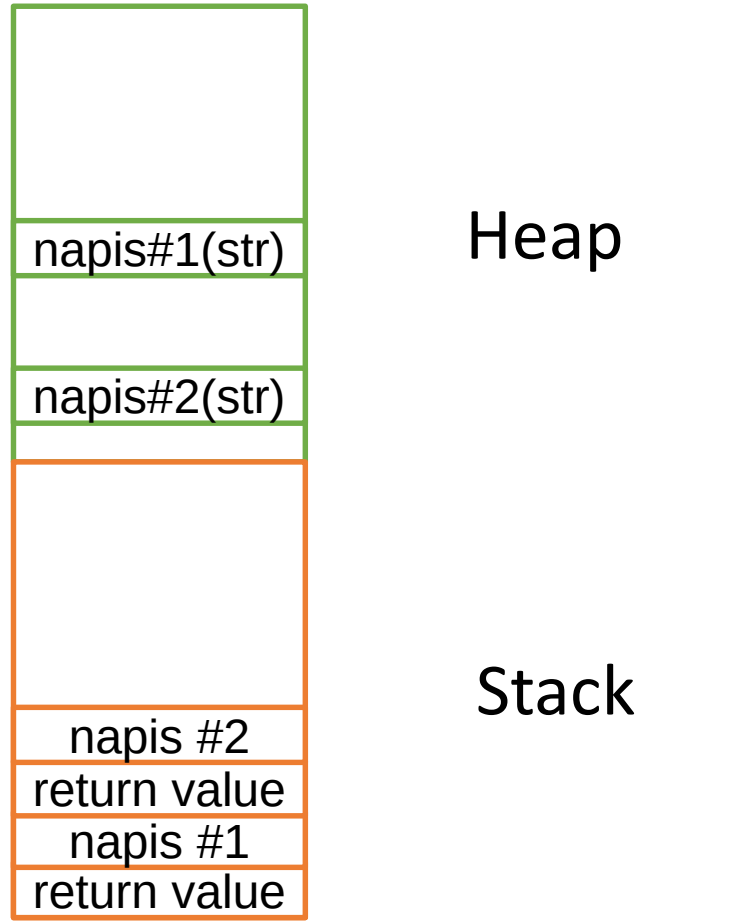
Heap

Stack

Pass by value

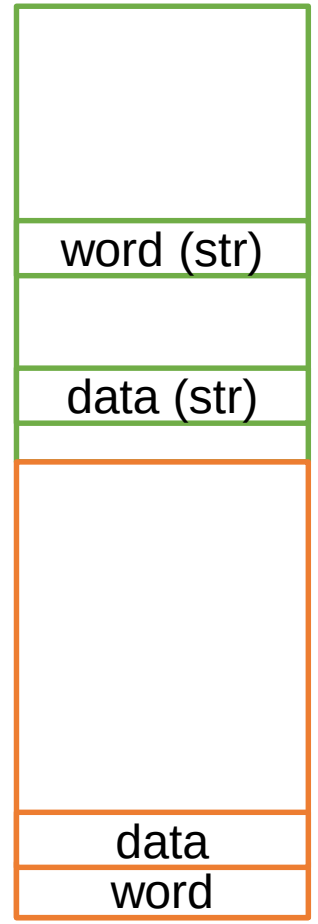
```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
  && palindrom(napis.substr(1, napis.size() - 2));
}
```



Pass by value

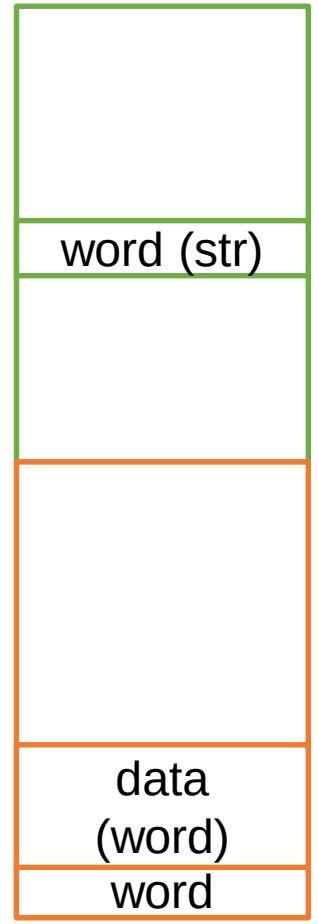
```
void add_something(std::string data)
{
    data += “something”;
}

{
    std::string word = “hi”;
    add_something(word);
}
```



“data” and “word” are 2 different objects

Pass by reference

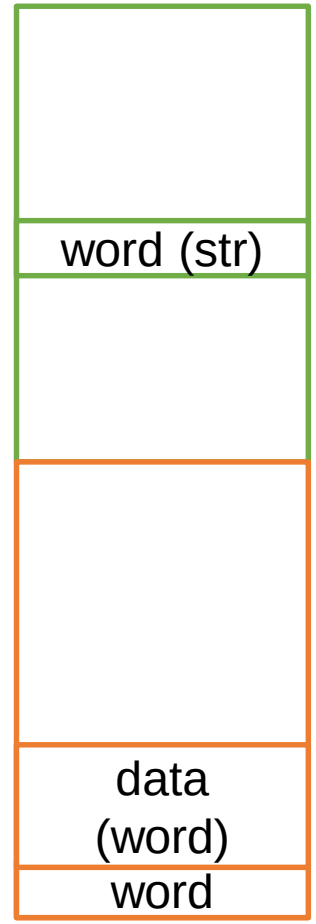
```
void add_something(std::string& data)
{
    data += “something”;
}

{
    std::string word = “hi”;
    add_something(word);
}
```



“data” points at “word”

Pass by pointer

```
void add_something(std::string* data)
{
    *data += “something”;
}

{
    std::string word = “hi”;
    add_something(&word);
}
```



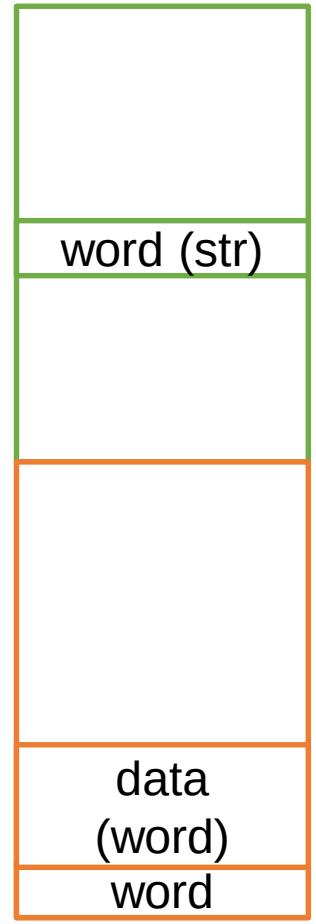
Heap

Stack

“data” points at “word”

Pass by pointer

```
void add_something(std::string data[])
{
    *data += “something”;
}

{
    std::string word = “hi”;
    add_something(&word);
}
```



“data” points at “word”

Home work

```cpp
bool palindrom(const std::string& napis)
{
    std::size_t dlugosc = napis.size();
    for(std::size_t i = 0; i < (dlugosc / 2); ++i)
    {
        if(napis[i] != napis[dlugosc - (i + 1)])
            return false;
    }
    return true;
}
```

Home work

```
bool palindrom(std::string napis)
{
    if(napis.size() < 2) return true;

    return napis.front() == napis.back()
        && palindrom(napis.substr(1, napis.size() - 2));
}
```

Keywords

- Basic stuff: bool, break, case, catch, char, const, continue, do, double, else, enum, false, float, for, if, int, long, return, short, signed, static_cast, switch, throw, true, try, unsigned, void, while, auto

- OOP: class, default, delete, dynamic_cast, friend, namespace, new, private, protected, public, struct, this, virtual, nullptr, operator

- For geeks: alignas, alignof, asm, constexpr, decltype, explicit, extern, goto, inline, mutable, noexcept, static_assert , template, thread_local, typeid, typename, union, using, volatile, reinterpret_cast, sizeof, static, const_cast

- Useless: and, and_eq, bitand, bitor, compl, export, not, not_eq, or, or_eq, register, typedef, xor, xor_eq

Pytania?

CODERS SCHOOL
www.coders.school