Programming in C++ - Primer

Lesson 3 - Beyond the Pointers

Jakub 'Eremiell' Marek
<marekj14@fel.cvut.cz>

Silicon Hill C++ Academy

2013/11/11

Welcome!

Diving deeper

Diving deeper

Arrays

Diving deeper

Arrays

- type name[size]

Diving deeper

Arrays

- type name[size]
- fixed size

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function
- must have different input types

Diving deeper

Arrays

- type name[size]
- fixed size
- 0 to size - 1

Variable visibility

- global/local
- block visibility
- overshadowing

Overloading

- same name, different function
- must have different input types
- doesn't care about names

Pointers

Pointers

Passing value by:

Pointers

Passing value by:

- Value

Pointers

Passing value by:

- Value
- Reference

Pointers

Passing value by:

- Value
- Reference

Pointers are:

Pointers

Passing value by:

- Value
- Reference

Pointers are:

- Addresses of memory

Pointers

Passing value by:

- Value
- Reference

Pointers are:

- Addresses of memory
- Variables like any other

Pointers

Passing value by:

- Value
- Reference

Pointers are:

- Addresses of memory
- Variables like any other

Two kinds of pointers:

Pointers

Passing value by:

- Value
- Reference

Pointers are:

- Addresses of memory
- Variables like any other

Two kinds of pointers:

- Pointers *

Pointers

Passing value by:

- Value
- Reference

Pointers are:

- Addresses of memory
- Variables like any other

Two kinds of pointers:

- Pointers *
- References &

Pointers

Pointers

Two pointer related operators:

Pointers

Two pointer related operators:

- Reference &

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

- Static

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

- Static
- Dynamic

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

- Static
- Dynamic

Dynamic allocation operators:

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

- Static
- Dynamic

Dynamic allocation operators:

- new

Pointers

Two pointer related operators:

- Reference &
- Defererence *

Segfaults
Command line arguments
Two kinds of memory allocation:

- Static
- Dynamic

Dynamic allocation operators:

- new
- delete, delete[]

C Strings

C Strings

arrays of chars

C Strings

arrays of chars
terminated by null byte \0

C Strings

arrays of chars
terminated by null byte \0
if not terminated properly, will brake

C Strings

arrays of chars
terminated by null byte \0
if not terminated properly, will brake

```
#include <iostream>

int main() {
    char cs[6] = {'a', 'b', 'c', 'd', 'e', '\0'};
    char s[] = "deHlorW !";
    std::cout << s[2] << s[1] << s[3] << s[3] << s[4] << s[7];
    std::cout << s[6] << s[4] << s[5] << s[3] << s[0] << s[8] << std::endl;
    std::cout << sizeof(s) << std::endl;
    return 0;
}
```

Why C++ Strings?

Why C++ Strings?

C strings are inflexible

Why C++ Strings?

C strings are inflexible
they're arrays = they're fixed size

Why C++ Strings?

C strings are inflexible
they're arrays = they're fixed size
easy to miss the end \0

C++ Strings

C++ Strings

class string in library <string>

C++ Strings

class string in library <string>
fully mutable

C++ Strings

class string in library <string>
fully mutable
encapsulated object

C++ Strings

class string in library <string>
fully mutable
encapsulated object
easily convertible

C++ Strings

class string in library <string>
fully mutable
encapsulated object
easily convertible

```
std::string s(<cstring>)
s.c_str()
```

Questions?

Vectors

Vectors

just as strings are encapsulated char arrays, vectors are encapsulated arrays in general

Vectors

just as strings are encapsulated char arrays, vectors are encapsulated arrays in general
class vector in library <vector>

Vectors

just as strings are encapsulated char arrays, vectors are encapsulated arrays in general
class vector in library <vector>
fully mutable

Vectors

just as strings are encapsulated char arrays, vectors are encapsulated arrays in general
class vector in library <vector>
fully mutable

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v();
    v.push_back(5);
    std::cout << v[0] << std::endl;
    return 0;
}
```

Standard Template Library (STL)

Standard Template Library (STL)

A lot of useful stuff

Standard Template Library (STL)

A lot of useful stuff

- maps

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams
- iterators

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams
- iterators
- algorithms

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams
- iterators
- algorithms
- constants

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams
- iterators
- algorithms
- constants
- regexes

Standard Template Library (STL)

A lot of useful stuff

- maps
- queues
- stacks
- lists
- C libraries
- file streams
- iterators
- algorithms
- constants
- regexes
- multitheading

Questions?

Structs

Structs

simple objects

Structs

simple objects
can only have members, no methods

Structs

simple objects
can only have members, no methods
from C

Structs

simple objects
can only have members, no methods
from C
form new data types

Structs

simple objects
can only have members, no methods
from C
form new data types

```
struct s {
    int i;
    double d;
};
```

Unions

Unions

structs with overlaying memory

Unions

structs with overlaying memory
make different data types save and load into same space

Unions

structs with overlaying memory
make different data types save and load into same space
good for coding/decoding various HW flags etc.

Unions

structs with overlaying memory
make different data types save and load into same space
good for coding/decoding various HW flags etc.

```
union u {
    int i;
    char a, b, c, d;
    unsigned b0: 1;
    unsigned b1: 1;
    unsigned b2: 1;
    unsigned b3: 1;
};
```

Questions?

Partial using

Partial using

using namespace is bad

Partial using

using namespace is bad
writing std:: is a bit tedious

Partial using

using namespace is bad
writing std:: is a bit tedious
you can use just some elements

Partial using

using namespace is bad
writing std:: is a bit tedious
you can use just some elements

```
#include <iostream>

using cout;
using endl;

int main() {
    cout << "abc" << endl;
    return 0;
}
```

Whole Number Logic

Whole Number Logic

Boolean is a number

Whole Number Logic

Boolean is a number

Can only be 0 or 1

Whole Number Logic

Boolean is a number

Can only be 0 or 1

Any other type gets explicit conversion

Whole Number Logic

Boolean is a number

Can only be 0 or 1

Any other type gets explicit conversion

- 0 -> 0 (False)

Whole Number Logic

Boolean is a number

Can only be 0 or 1

Any other type gets explicit conversion

- 0 -> 0 (False)
- anything else -> 1 (True)

Whole Number Logic

Boolean is a number

Can only be 0 or 1

Any other type gets explicit conversion

- 0 -> 0 (False)
- anything else -> 1 (True)

NULL, EOF, \0 etc. classifies as 0

Whole Number Logic

Boolean is a number
Can only be 0 or 1
Any other type gets explicit conversion

- 0 -> 0 (False)
- anything else -> 1 (True)

NULL, EOF, \0 etc. classifies as 0
so you can:

```
if (i) {...}
if (!i) {...}
if (getState()) {...}
while (line = instr.getline()) {...}
if (ptr) {...}
if (!ptr) {...}
```

Questions?

Break!