Programming in C++ - Primer

Lesson 5 - Objects

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

Silicon Hill C++ Academy

2013/11/18

Welcome!

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[]

Containers and Structs

Containers and Structs

Containers

Containers and Structs

Containers

- Strings

Containers and Structs

Containers

- Strings
- Vectors

Containers and Structs

Containers

- Strings
- Vectors
- and many others. . .

Containers and Structs

Containers

- Strings
- Vectors
- and many others. . .

Structs

Containers and Structs

Containers

- Strings
- Vectors
- and many others. . .

Structs

- aggregate data types

Containers and Structs

Containers

- Strings
- Vectors
- and many others. . .

Structs

- aggregate data types
- struct

Containers and Structs

Containers

- Strings
- Vectors
- and many others. . .

Structs

- aggregate data types
- struct
- union

Programming Styles

Programming Styles

What we've been to before?

Programming Styles

What we've been to before?

- naive programming

Programming Styles

What we've been to before?

- naive programming
- procedural programming

Programming Styles

What we've been to before?

- naive programming
- procedural programming
- object oriented programming

Programming Styles

What we've been to before?

- naive programming
- procedural programming
- object oriented programming

other branches like functional programming

Programming Styles

What we've been to before?

- naive programming
- procedural programming
- object oriented programming

other branches like functional programming
the best of it usually integrated

Programming Styles

What we've been to before?

- naive programming
- procedural programming
- object oriented programming

other branches like functional programming
the best of it usually integrated
still new & unpredictable

Why OOP?

Why OOP?

easier to maintain

Why OOP?

easier to maintain

things that belong together are together

Why OOP?

easier to maintain
things that belong together are together
tighter logic

Why OOP?

easier to maintain
things that belong together are together
tighter logic
considered best practice

Questions?

Objects

Objects

better structs

Objects

better structs
their own member functions = methods

Objects

better structs

their own member functions = methods

their own visibility layers

Objects

better structs
their own member functions = methods
their own visibility layers
inheritance & polymorphism

Access Specifiers

Access Specifiers

new keywords

Access Specifiers

new keywords
important for data encapsulation

Access Specifiers

new keywords

important for data encapsulation

define member visibility

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public everyone sees it

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public	everyone sees it
protected	only instances of this class and it's derived classes and friends

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public	everyone sees it
protected	only instances of this class and it's derived classes and friends
private	only instances of this class and friends

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public	everyone sees it
protected	only instances of this class and it's derived classes and friends
private	only instances of this class and friends

best practice:

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public	everyone sees it
protected	only instances of this class and it's derived classes and friends
private	only instances of this class and friends

best practice:
start with public block, then protected, then private

Access Specifiers

new keywords
important for data encapsulation
define member visibility

public	everyone sees it
protected	only instances of this class and it's derived classes and friends
private	only instances of this class and friends

best practice:
start with public block, then protected, then private
(biggest audience first)

Encapsulation

Encapsulation

data should never be freely reachable by anyone

Encapsulation

data should never be freely reachable by anyone
because if they are, anyone can change them!

Encapsulation

data should never be freely reachable by anyone
because if they are, anyone can change them!
so we cannot guard the consistence of our objects

Encapsulation

data should never be freely reachable by anyone
because if they are, anyone can change them!
so we cannot guard the consistence of our objects
members are usually private or protected

Encapsulation

data should never be freely reachable by anyone
because if they are, anyone can change them!
so we cannot guard the consistence of our objects
members are usually private or protected
getter/setter methods in public part

Encapsulation

data should never be freely reachable by anyone
because if they are, anyone can change them!
so we cannot guard the consistence of our objects
members are usually private or protected
getter/setter methods in public part
guarding of the contents

Silicon Hill

Constructor/Destructor

C++ Primer

Jakub Marek

Constructor/Destructor

Constructor brings object to its initial state

Constructor/Destructor

Constructor brings object to its initial state
Destructor frees all alocated memory and prepares object for deletion

Constructor/Destructor

Constructor brings object to its initial state

Destructor frees all alocated memory and prepares object for deletion

often shorthanded as ctor/dtor

Constructor/Destructor

Constructor brings object to its initial state
Destructor frees all alocated memory and prepares object for deletion
often shorthanded as ctor/dtor
named same as class

Constructor/Destructor

Constructor brings object to its initial state

Destructor frees all alocated memory and prepares object for deletion

often shorthanded as ctor/dtor

named same as class

never have return types (unlike any other functions)

Constructor/Destructor

Constructor brings object to its initial state
Destructor frees all alocated memory and prepares object for deletion
often shorthanded as ctor/dtor
named same as class
never have return types (unlike any other functions)
dtors start with ˜

Constructor/Destructor

Constructor brings object to its initial state
Destructor frees all alocated memory and prepares object for deletion
often shorthanded as ctor/dtor
named same as class
never have return types (unlike any other functions)
dtors start with ˜
if you write none, default ones will be provided

Constructor/Destructor

Constructor brings object to its initial state

Destructor frees all alocated memory and prepares object for deletion

often shorthanded as ctor/dtor

named same as class

never have return types (unlike any other functions)

dtors start with ˜

if you write none, default ones will be provided

if you write any, default ones will be gone!

Access Operators

Access Operators

class creates namespace

Access Operators

class creates namespace
instance creates inner block

Access Operators

class creates namespace
instance creates inner block

:: operator class members

Access Operators

class creates namespace
instance creates inner block

:: operator	class members
. operator	instance members

Access Operators

class creates namespace
instance creates inner block

:: operator class members

. operator instance members

-> operator instance members via pointers

Declaration (Header file)

```
#ifndef HEADER_HPP
#define HEADER_HPP

#include <string>
#include <cassert>

class Hero {
    public:
        Hero(std::string name, int maxHealth = 100) : name(name),
            maxHealth(maxHealth), health(maxHealth) {};
        ~Hero() {};
        int getHealth();
        int getMaxHealth();
        std::string getName();
        int harm(int);
        int heal(int);
    protected:
        int health;
        int maxHealth;
        std::string name;
        bool checkAlive();
        void testInvariant();
};

#endif
```

Definition (Source file)

```cpp
#include "header.hpp"

int Hero::getHealth() {
    return health;
}

int Hero::getMaxHealth() {
    return maxHealth;
}

std::string Hero::getName() {
    return name;
}

int Hero::harm(int i) {
    health -= i;
    if (health < 0) {
        health = 0;
    }
    checkAlive();
    testInvariant();
    return health;
}
```

Definition (Source file)

```
int Hero::heal(int i) {
    health += i;
    if (health > maxHealth) {
        health = maxHealth;
    }
    testInvariant();
    return health;
}

bool Hero::checkAlive() {
    return health > 0;
}

void Hero::testInvariant() {
    assert (health >= 0 && health <= maxHealth);
}
```

Questions?

Advanced Topics

Advanced Topics

self-testing code

Advanced Topics

self-testing code

methods are of two kinds:

Advanced Topics

self-testing code
methods are of two kinds:
observers and mutators

Advanced Topics

self-testing code
methods are of two kinds:
observers and mutators

- observers just observe

Advanced Topics

self-testing code
methods are of two kinds:
observers and mutators

- observers just observe
- mutators mutate the object and can break it

Advanced Topics

self-testing code
methods are of two kinds:
observers and mutators

- observers just observe
- mutators mutate the object and can break it

so they need to test invariant

Advanced Topics

self-testing code
methods are of two kinds:
observers and mutators

- observers just observe
- mutators mutate the object and can break it

so they need to test invariant
contracts

Makefiles

Makefiles

so you don't have to compile in hand all the time

```
all:     clean
    g++ -ansi -pedantic -Wall -Wextra -O2 -omyprog myprog.cpp
        module1.cpp module2.cpp -I. -L. -lm
install:
    sudo cp myprog /usr/bin/
clean:
    rm -f *.o
    rm -f myprog
```

Questions?

Break!