Programming in C++ - Primer

Lesson 6 - Advanced Objects

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

Silicon Hill C++ Academy

2013/11/25

Welcome!

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

Objects

Objects

Declaration

Objects

Declaration

- header file

Objects

Declaration

- header file
- included wherever we use it

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file
- is usually the only include

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file
- is usually the only include

Access rights

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file
- is usually the only include

Access rights

- public

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file
- is usually the only include

Access rights

- public
- protected

Objects

Declaration

- header file
- included wherever we use it
- includes whatever is needed

Definition

- source file
- includes the header file
- is usually the only include

Access rights

- public
- protected
- private

Encapsulation

Encapsulation

black boxes

Encapsulation

black boxes
noone should care, how it works inside

Encapsulation

black boxes
noone should care, how it works inside
providing interface

Encapsulation

black boxes
noone should care, how it works inside
providing interface
changing interface might break other code

Encapsulation

black boxes
noone should care, how it works inside
providing interface
changing interface might break other code
changing internals shouldn't

Encapsulation

black boxes
noone should care, how it works inside
providing interface
changing interface might break other code
changing internals shouldn't
getters/setters guard the invariant

Encapsulation

black boxes
noone should care, how it works inside
providing interface
changing interface might break other code
changing internals shouldn't
getters/setters guard the invariant
user of your code only needs the contract, headers & object files

Questions?

Inheritance

Inheritance

objects can have special cases

Inheritance

objects can have special cases
two ways of inheritance:

Inheritance

objects can have special cases
two ways of inheritance:

- specialization

Inheritance

objects can have special cases
two ways of inheritance:

- specialization
- generalization

Inheritance

objects can have special cases
two ways of inheritance:

- specialization
- generalization

think use wise and data wise

Inheritance

objects can have special cases
two ways of inheritance:

- specialization
- generalization

think use wise and data wise
C++ knows multiple inheritance! (unlike Java)

Polymorphism

Polymorphism

the parent pointer can contain the derived object

Polymorphism

the parent pointer can contain the derived object because it's special case

Polymorphism

the parent pointer can contain the derived object
because it's special case
you can overshadow parent's methods

Polymorphism

the parent pointer can contain the derived object
because it's special case
you can overshadow parent's methods
can still get the proper ones

Polymorphism

the parent pointer can contain the derived object
because it's special case
you can overshadow parent's methods
can still get the proper ones
but you need virtual keyword

Polymorphism

the parent pointer can contain the derived object
because it's special case
you can overshadow parent's methods
can still get the proper ones
but you need virtual keyword
early/late binding

Inheritance & Polymorphism

Inheritance & Polymorphism


```
class Square {
    public:
        Square(int a = 1) : a(a) {};
        int getA() {return a;};
        int setA(int a) {this->a = a; return this->a;};
        virtual int area() {return a * a;};
    protected:
        int a;
}

class Rectangle : public Square {
    public:
        Rectangle(int a = 1, int b = 1) : Square(a), b(b) {};
        int getB() {return b;};
        int setB(int b) {this->b = b; return this->b;};
        virtual int area() {return a * b;};
    protected:
        int b;
}
```

Questions?

Operators

Operators

operators are functions like any other

Operators

operators are functions like any other
so we can overshadow and overload them!

Operators

operators are functions like any other
so we can overshadow and overload them!

```
class Complex {
    public:
        Complex(int r = 0, int i = 0) : r(r), i(i) {};
        Complex operator+(const Complex &c)
            {return Complex(this->r + c.r, this->i + c.i);};
        Complex& operator=(const Complex &c)
            {this->r = c.r; this->i = c.i; return *this;};
        Complex& operator+=(const Complex &c)
            {*this = *this + c; return *this;};
    protected:
        int r, i;
};
```

Operators

operators are functions like any other
so we can overshadow and overload them!

```
class Complex {
    public:
        Complex(int r = 0, int i = 0) : r(r), i(i) {};
        Complex operator+(const Complex &c)
            {return Complex(this->r + c.r, this->i + c.i);};
        Complex& operator=(const Complex &c)
            {this->r = c.r; this->i = c.i; return *this;};
        Complex& operator+=(const Complex &c)
            {*this = *this + c; return *this;};
    protected:
        int r, i;
};
```

what to give and get from operators is a bit of alchemy

Copying

Copying

by copying an object, we get a shallow copy

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy
we need to implement that!

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy
we need to implement that!

- overloading operators

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy
we need to implement that!

- overloading operators
- creating copy constructor

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy
we need to implement that!

- overloading operators
- creating copy constructor

```
class Complex {
    public:
        Complex(int r = 0, int i = 0) : r(r), i(i) {};
        Complex(Complex c) : r(c.r), i(c.i) {};
        Complex& operator=(const Complex &c)
            {this->r = c.r; this->i = c.i; return *this;};
    protected:
        int r, i;
}
```

Copying

by copying an object, we get a shallow copy
it only copies the pointers, we have
sometimes, we need a deep copy
we need to implement that!

- overloading operators
- creating copy constructor

```
class Complex {
    public:
        Complex(int r = 0, int i = 0) : r(r), i(i) {};
        Complex(Complex c) : r(c.r), i(c.i) {};
        Complex& operator=(const Complex &c)
            {this->r = c.r; this->i = c.i; return *this;};
    protected:
        int r, i;
}
```

when copying member objects, we need to do that manually!

Questions?

Friendship

Friendship

Friendship Is Magic!

Friendship

Friendship Is Magic!

(puns intended ;))

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators
- multiple classes as parts of same abstraction

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators
- multiple classes as parts of same abstraction

you can friend the whole class or just one function

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators
- multiple classes as parts of same abstraction

you can friend the whole class or just one function
a friend is not a member of given class!

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators
- multiple classes as parts of same abstraction

you can friend the whole class or just one function
a friend is not a member of given class!
friendship isn't propagated though inheritance

Friendship

Friendship Is Magic!
(puns intended ;))
allows to propagate class internals
which breaks encapsulation!
but sometimes, it's needed

- with certain operators
- multiple classes as parts of same abstraction

you can friend the whole class or just one function
a friend is not a member of given class!
friendship isn't propagated though inheritance
(parent's friend isn't child's friend)

Friendship


```
class a {
    public:
        a(int i) : i(i) {};
        friend int showl(a instance);
        int getl();
    private:
        int i;
};

int a::getl() {
    return this->i;
}

int showl(a instance) {
    return instance.i;
}
```

Questions?

Break!