Programming in C++ - Primer

Lesson 7 - Typecasts, Iterators & Templates

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

Silicon Hill C++ Academy

2013/12/02

Welcome!

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

Objects

Objects

Encapsulation

Objects

Encapsulation

- helps data safety

Objects

Encapsulation

- helps data safety
- getters/setters

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

- can be overloaded

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

- can be overloaded
- beware of member/non-member operators!

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

- can be overloaded
- beware of member/non-member operators!

Friendship

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

- can be overloaded
- beware of member/non-member operators!

Friendship

- breaks encapsulation

Objects

Encapsulation

- helps data safety
- getters/setters

Inheritance & Polymorphism

- related classes should inherit
- specialization/generalization
- early/late binding

Operators

- can be overloaded
- beware of member/non-member operators!

Friendship

- breaks encapsulation
- helps divided abstractions

Handing over Objects

Handing over Objects

once again three possibilities

Handing over Objects

once again three possibilities

- by value

Handing over Objects

once again three possibilities

- by value
- by pointer

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy
you've got to have

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy
you've got to have

- copy ctor

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy
you've got to have

- copy ctor
- copy assignment operator

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy
you've got to have

- copy ctor
- copy assignment operator
- dtor

Handing over Objects

once again three possibilities

- by value
- by pointer
- by reference

to pass by value means to copy
you've got to have

- copy ctor
- copy assignment operator
- dtor

The Rule of Three

Handing over Objects

Handing over Objects

passing by pointers & references gives original

Handing over Objects

passing by pointers & references gives original
you can't change reference

Handing over Objects

passing by pointers & references gives original
you can't change reference
you have to initialize reference at creation

Handing over Objects

passing by pointers & references gives original
you can't change reference
you have to initialize reference at creation
reference cannot be NULL

Questions?

Advanced Typecasts

Advanced Typecasts

we've already seen:

Advanced Typecasts

we've already seen:

- implicit typecasting

Advanced Typecasts

we've already seen:

- implicit typecasting

```
int i = 'b';
```

Advanced Typecasts

we've already seen:

- implicit typecasting

```
int i = 'b';
```

- explicit typecasting

Advanced Typecasts

we've already seen:

- implicit typecasting

```
int i = 'b';
```

- explicit typecasting

```
std::cout << (char) i << std::endl;
std::cout << char(i) << std::endl;
```

Advanced Typecasts

we've already seen:

- implicit typecasting

```
int i = 'b';
```

- explicit typecasting

```
std::cout << (char) i << std::endl;
std::cout << char(i) << std::endl;
```

works seamlessly only with plain old data (primitives)

Advanced Typecasts

Advanced Typecasts

four new typecasts

Advanced Typecasts

four new typecasts
templates

Advanced Typecasts

four new typecasts
templates
works on pointers & references

Advanced Typecasts

Advanced Typecasts

- dynamic_cast

Advanced Typecasts

- dynamic_cast

most powerful

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

- static_cast

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

- static_cast

only checks for compatibility

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

- static_cast

only checks for compatibility
ie. are the types related to each other?

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

- static_cast

only checks for compatibility
ie. are the types related to each other?
programmer has to take care of correctness

Advanced Typecasts

- dynamic_cast

most powerful
only successful if you're casting into right class

```
Animal *a = new Cat("Moonlight");
Cat *c = dynamic_cast<Cat*>(a);
```

- static_cast

only checks for compatibility
ie. are the types related to each other?
programmer has to take care of correctness

```
Animal *a = new Cat("Moonlight");
Bat *b = static_cast<Bat*>(a);
```

Advanced Typecasts

Advanced Typecasts

- reinterpret_cast

Advanced Typecasts

- reinterpret_cast

will copy anything

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent
you can store pointers as integers etc.

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent
you can store pointers as integers etc.

```
Animal *a = new Cat("Moonlight");
Car *c = reinterpret_cast<Car*>(a);
```

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent
you can store pointers as integers etc.

```
Animal *a = new Cat("Moonlight");
Car *c = reinterpret_cast<Car*>(a);
```

- const_cast

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent
you can store pointers as integers etc.

```
Animal *a = new Cat("Moonlight");
Car *c = reinterpret_cast<Car*>(a);
```

- const_cast

casts const objects into non-const and vice versa

Advanced Typecasts

- reinterpret_cast

will copy anything
usually platform dependent
you can store pointers as integers etc.

```
Animal *a = new Cat("Moonlight");
Car *c = reinterpret_cast<Car*>(a);
```

- const_cast

casts const objects into non-const and vice versa

```
int encode(char c) {...}
const char c = 'x';
std::cout << encode(const_cast<char*>(c));
```

Questions?

Containers Revisited

Containers Revisited

vector dynamic array, adding to the end

Containers Revisited

vector dynamic array, adding to the end

deque dynamic array, adding to both sides

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding
set	set (only unique members), no order
multiset	set with duplicities

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding
set	set (only unique members), no order
multiset	set with duplicities
map	mapping given key to a value, unique keys
multimap	duplicate keys allowed

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding
set	set (only unique members), no order
multiset	set with duplicities
map	mapping given key to a value, unique keys
multimap	duplicate keys allowed
stack	stack LIFO
queue	queue FIFO

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding
set	set (only unique members), no order
multiset	set with duplicities
map	mapping given key to a value, unique keys
multimap	duplicate keys allowed
stack	stack LIFO
queue	queue FIFO
priority_queue	queue with priority

Containers Revisited

vector	dynamic array, adding to the end
deque	dynamic array, adding to both sides
list	linked list, a lot of adding
set	set (only unique members), no order
multiset	set with duplicities
map	mapping given key to a value, unique keys
multimap	duplicate keys allowed
stack	stack LIFO
queue	queue FIFO
priority_queue	queue with priority
string	a string
rope	a string container

Questions?

Iterators

Iterators

allows you to traverse containers

Iterators

allows you to traverse containers
several types of

Iterators

allows you to traverse containers
several types of
each container has it's own

Iterators

allows you to traverse containers
several types of
each container has it's own

```
std::vector<Animal> a;
std::vector<Animal>::iterator it;
for (it = a.begin(); it != a.end() && newpet > *it; it++) {}
a.insert(it, newpet);
```

Questions?

Templates

Templates

allow you to make classes or functions

Templates

allow you to make classes or functions
which work dynamically with different other classes

Templates

allow you to make classes or functions
which work dynamically with different other classes
or you can pass them any parameters

Templates

allow you to make classes or functions
which work dynamically with different other classes
or you can pass them any parameters

```
template<class T>
template<class T = int>
template<int N>
template<class T = char, int N = 10>
int func<T>(T param) {...}
class c {T a, b, c; T[N] ar;};
```

Template Specialization

Template Specialization

oftentimes template does just fine for most classes

Template Specialization

oftentimes template does just fine for most classes
but not for some...

Template Specialization

oftentimes template does just fine for most classes
but not for some...
then, we can specialize

Template Specialization

oftentimes template does just fine for most classes
but not for some...
then, we can specialize

```
template<class T>
class myobject {T ar[80];};

template<>
class myobject<bool> {unsigned char ar[10];};
```

Questions?

Break!