



System and Parallel Programming

Dr. Alexandru Calotoiu

C++ SPOTLIGHTS

Overview

Introduction

References

Classes and objects

Exceptions

Templates

Books:

- Stroustrup, *The C++ Programming Language*
- Stroustrup, *Die C++ Programmiersprache*
- Lippman and Lajoie, *C++ Primer*,

Online:

- Bernd Mohr, Programming in C++ (Slides): <http://d-nb.info/973093625/34>

- 1979 Stroustrup starts to extend C with classes
- 1983 “C with classes” renamed to C++
- 1985 First version of C++
- 1989 Second version of C++, multi-inheritance, abstract classes, static/const methods, ...
- 1998 First ISO C++ standard (C++98), templates and STL
- 2003 C++03, clarifications
- 2011 C++11, threading, lambdas, rvalue-references, ...
- 2015 C++14, minor extensions to C++11
- 2017 C++17, minor extensions to C++14

C++ extends C

- Support for object oriented programming
- Generic programming (templates)
- Some more extensions

C++ is compatible to C89

- C includes modifications in C99 and later that are incompatible with C++
- C is still “mostly” compatible with C++

Overview

Introduction

References

Classes and objects

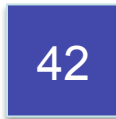
Exceptions

Templates

References are a new name for a variable.

```
int i = 42;
```

```
int &r = i;
```



i, r

If a function declares a parameter as a reference, the variable is just another name for the passed variable.

```
void func( int& r ); // Any modification to  
func( i );           // r modifies i, too
```

Here, `r` and `i` use the same memory location

References vs. Pointers

References have some of the properties of pointers

- Passing objects without creating a copy
- Passed object can be modified inside the function

But avoid some of its problems

- Memory leaks

Use references instead of pointers if possible

Overview



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Introduction

References

Classes and objects

Exceptions

Templates

Programming paradigm

Abstract idea

- Existence of objects
- Objects have attributes which are described in member variables
- Actions, associated with an object, are described in methods

A class describes the members and method for a certain type of objects

- Classes are types

Objects are instances of a class

- Objects are variables

Classes



```
class classname {  
    // member and method declarations  
};  
  
classname A; // Creates an object of the class
```

Example:

```
class Car {  
  
};  
  
Car A;
```

Members are variables of a class

- Writes a variable definition inside the class
- Access to members like for `structs`

```
class Car {  
    double max_speed;  
};  
  
Car A;  
  
Car* B = &A;  
  
B->max_speed = A.max_speed;
```

Methods are functions that are defined in a class

- Access to methods via `.` or `->` like for members
- Can access member variables of the object
- For implementation the method name is prefixed with `classname::`

```
class Car {  
    double max_speed;  
    void drive( double distance );  
};  
  
void Car::drive( double distance ) {  
    double time = distance/max_speed;  
}
```

Overloading



C++ allows multiple methods with the name
The signature must differ

Allowed

```
class A
{
    foo();
    foo(int a);
    foo(double b);
};
```

Error

```
class A
{
    foo();
    foo(int a);
    foo(int b);
};
```

Allow access only to interface functions

private: Only methods of this class can access the symbol

- Default is private

protected: Accessible for methods of this class and derived classes

public: Everybody can access the symbol

```
class Car {  
    private:  
        double max_speed;  
    public:  
        void drive( double distance );  
};
```

If one class is a special case from a more general class

- E.g. an apple tree is a special case of a tree
- The class of the special case can be derived from the base class
- The child class inherits all members and methods from the base class
- The accessibility of the members/methods of the base class can be restricted
 - Is the base class public, the accessibility is unchanged

```
class Tree
{
protected:
    double height
};
```

```
class AppleTree : public Tree
{
public:
    double getHeight { return height; }
};
```


Class Construction (1)

Constructors create a new instance of a class

Constructor definition looks like a method

- With the same name as the class
- Has no return type
- Can have parameters

```
class MyClass {  
public:  
    MyClass( int arg );  
};
```

Class Construction (2)

The constructor can/should call

- constructors of parent classes
- constructors of members

```
class MyClass : public ParentClass {  
private:  
    int data;  
public:  
    MyClass( int arg ) :  
        ParentClass( arg ),  
        data( 42 ) {}  
};
```

Create local objects

```
class MyClass {  
    MyClass();  
    MyClass( int i );  
};  
  
MyClass A(4);  
MyClass B();  
MyClass C; // Invokes MyClass()
```

The compiler may generate some constructors:

- The default constructor:
 - If no custom constructor exists
 - Has no arguments
 - Equals a constructor with an empty body
 - Calls default constructor of base classes and members
- The copy constructor
 - If no copy constructor exists
 - Has a reference of an object of the same type as parameter
 - Initializes all members with the values of the copied object
 - Caution: Pointers copy only addresses, not the object. (shallow copy)

Destructor

A destructor is called when an object is deleted.

Only one destructor can exist for each class

The destructor has the name of the class and a leading '~'

```
class my_class {  
public:  
    virtual ~my_class();  
};
```

If no destructor is provided, the compiler generates a default destructor with an empty body

Base classes usually require virtual destructors

Dynamic Allocation

```
class MyClass {  
    MyClass();  
    MyClass( int i );  
};
```

Dynamic allocation with `new`

```
MyClass* D = new MyClass(4);
```

```
MyClass* E = new MyClass();
```

Dynamically allocated objects must be explicitly deleted

```
delete (D);
```

```
delete (E);
```

Passing objects

```
class myClass;

void foo( myClass A, myClass* B, myClass& C ) {
    // A is a copy of orig, invokes copy constructor
    // B points to orig, changes have side effects
    // C is a new name for orig, have side effects
}

myClass orig();

foo( orig, &orig, orig );
```

Passing parameters: C/C++ vs. Java

Passing native data types

- Java behaves like C/C++ value copy

Passing objects

- Java behaves like C++ references

Overview

Introduction

References

Classes and objects

Exceptions

Templates

Exceptions (1)

Exceptions provide a mechanism to react on exceptional circumstances

- E.g. runtime errors

They are not means to return a value from a function

- Extremely high overhead

Exceptions (2)

```
try
{
    // Code under inspection
    throw my_exception();
}
catch( my_exception& e )
{
    // Reaction on exception e
}
```

If an exception was caught, execution continues after the **catch block**

- Not after the `throw` statement

Throw

The `throw` command expects one parameter that is passed as an argument to the `catch` clause

Can be an arbitrary type

- You can throw basic types, like integers

The C++ standard library provides a base class for all exceptions it throws: `std::exception`

A `catch` block catches only exceptions of matching type

A `catch` block must immediately follow the `try` block

You can chain `catch` blocks

```
try {  
    //something  
} catch( my_exception& e ) {} // exception by reference  
    catch (std::exception& e) {}  
    catch (int e) {}           // passes value by copy  
    catch (...) {}            // catches all exceptions
```

Overview

Introduction

References

Classes and objects

Exceptions

Templates

Motivation – templates

Imagine you implement a data structure, e.g. a stack

And later you need the same data structure again, but for another data type

You could copy/paste all the code and change the data type everywhere

- Duplication of code

You could store void pointers to the data objects

- Limits the data types to pointers
- No type checking by the compiler

Wouldn't it be nice to just write a template from which the compiler generates different versions?

C++ provides templates

They allow to describe an algorithm in an generic way,
generate multiple versions of this algorithm,
and tell the compiler that in each version of the algorithm it
should insert a specific type.

Example – stack with fixed type

```
class stack {  
    int *data;  
    int size, top;  
public:  
    stack(int s) : size(s), top(0) {  
        data = new int[size];  
    }  
    virtual ~stack();  
    void push( int new_node);  
    int pop();  
};
```

Example – stack with template type



```
template <typename T>
class stack {
    T *data;
    int size, top;
public:
    stack(int s) : size(s), top(0) {
        data = new T[size];
    }
    virtual ~stack();
    void push( T new_node);
    T pop();
};
```

Instantiation of non-template class:

```
stack stack_with_fixed_type( 100 );
```

Instantiation of template class:

```
stack<int> stack_with_template( 100 );
```

The code for `stack<int>` is generated when the compiler encounters the instantiation

Some compilers require all template code to be in the header

The `<int>` becomes part of the type name and appears everywhere you would write the type name

Template features

You can instantiate a template type with another template type

```
stack < my_other_template < int > >
```

Creates a stack object that stores objects of type

```
my_other_template < int >
```

You can have multiple type parameters in a template definition:

```
template <typename A, typename B>
```

```
class my_class { ... };
```

You can also generate templates for a method:

```
template <typename T>
```

```
void sort( T* array, int size );
```

Standard Template Library (STL)

Part of the C++ Standard

Defines a set of common data structures and algorithms

Makes heavy use of templates

STL classes are in the namespace `std`

STL containers

Containers are objects that store a collection of objects

Implemented as templates to support different data types

All implement a similar interface

Containers differ in their algorithmic complexity to insert, remove or access an element

Some example STL data structures are:

```
template <typename T> std::vector
```

```
template <typename T> std::deque
```

```
template <typename key_t, typename value_t>
```

```
std::map
```

std::vector

The class `std::vector` is like an array that can grow dynamically

- Uses reallocation internally to adapt size

Fast element access

Appending/removing elements at the end is fast

Slow insertion/removal of elements at other positions

```
std::vector<std::string> my_vec;
```

```
my_vec[0] = "Alex";
```

```
std::string s = my_vec[0];
```

std::map

Maps are associative containers that associate a key with a value

The key can be any comparable type

Keys must be unique

```
std::map<std::string, long> my_map;  
my_map["Alex"] = 123456;  
long n = my_map["Alex"];
```


Iterators (1)

Iterators are objects that point to an element in a container

Iterators provide operators to iterate over the elements of the container

Usage of iterators look similar to pointers

```
std::vector<string> my_vec;  
std::vector<string>::iterator i;  
for( i = my_vec.begin(); i != my_vec.end(); i++ )  
{  
    string current = *i; // Use i-> to access members directly  
}
```

Iterators (2)

Iterators are class objects

Overwrite various operators, e.g.

- Smart pointers \rightarrow
- Dereference operator $*$
- Increment operator $++$
- Comparison operator $==$

Iterator objects are tied to a container and an iteration order

- E.g., different iterator classes for iteration in forward and reverse order

Provide reliable checks for boundaries