D C++ for Java Programmers

D.1 Introduction

- General differences to Java
- Objects and Classes in C++
- Constructors and Destructors
- Inheritance
- Exceptions
- Odds and Ends
- Operator overloading
- No: Templates
- No: Standard Template Library (STL)
1 A Short History of C++

- 1980: Dennis Ritchie extends C to *C with Classes*
- 1983: Bjarne Stroustrup introduces C++ V1.0
- 1989: ANSI approves Standard C with elements from C++
- 1989: ANSI committee X3J16 begins standardization of C++ (V2.0)
- 1991: *The Annotated C++ Reference Manual* defines C++ V3.0 including *Templates* and *Exceptions*
- 1993: C++ V3.1 includes *Namespaces* and *Run-Time Type Identification*
- 1997: ISO WG21 and ANSI X3J16 adopt C++ and the *Standard Template Library (STL)* as standard ISO/IEC FDIS 14882
2 What is C++?

- Super-set of C
- A better C
  - Strong typing
  - Prototypes
  - Overloading
- Extends C to include object-oriented concepts
  - Objects
  - Classes
  - Inheritance
  - Polymorphism
- *BUT*: C++ does not enforce an object-oriented style of programming
  ➡️ Therefore you learn Java first!
3 Literature


- *ANSI C++ Public Comment Draft*, December 1996. See tutorial web page


D.2 General Differences to Java

- Input and output
- Inlining
- Scope operator
- Namespaces
- Memory management
- Function overloading
- Reference variables
- Default parameters
- Constants
1 Input and output

- Input and output to *Streams* via *Operators*
  - `cin`  
    - Input stream (global)
  - `cout, cerr, (clog)`  
    - Output streams (global)
  - `>>`  
    - Input operator
  - `<<`  
    - Output operator

- Example:

```cpp
#include <iostream>

void main() {
    int test; // i/o test variable
    cin >> test;
    cout << "test=" << test << "\n";
}
```

- C: `scanf` and `printf` are not type-safe (format string)
2 Inlining

- Reserved word `inline`:

```c
inline return_type function_name( parameter_list ) {
  function_body
}
```

- Compiler tries to optimize function calls
- Instead of a function call the body of the whole function is inserted
  - Faster calls, but larger programs
- Further optimizations possible (e.g. for calls with constant parameters)
- Not possible for recursive functions
- `Function body must be implemented in the header file (.H or .hh)!!!`

- Differences to pre-processor macros (`#define`):
  - Macros are expanded as normal text
    - No type checking, often mysterious syntax errors
  - No repeated expansion for `inline` functions
3 Scope operator

- New operator :: for accessing scopes
- Mainly used with classes and namespaces
- Here: Accessing hidden variables with the same identifier in other scopes
- Example:

```cpp
#include <iostream>

int test = 4711; // global variable

void main() {
    int test = 1234; // local variable

    cout << "The global variable is " << ::test << "\n";
    cout << "The local variable is " << test << "\n";
}
```
4 Namespaces

- New reserved word `namespace`:

```cpp
namespace namespace_name {
    declarations/definitions
}
```

- Opens a new namespace for identifiers
- Can be nested
- Access via scope operator `::`
- Like `package` in Java, but no relation to file organisation

- Example:

```cpp
namespace Date {
    struct Time {
        int year;
        ...
    }
}
Date::Time today;
```
4 Namespaces (2)

- Import of identifiers from other name spaces via `using`:

  ```cpp
  using namespace_name::identifier;
  ```

  ◆ Like `import package.identifier;` in Java

- Import of complete name spaces:

  ```cpp
  using namespace namespace_name;
  ```

  ◆ Like `import package.*;` in Java

- Example:

  ```cpp
  namespace Date {
    struct Time { ... }
  }

  namespace MyApp {
    using Date::Time;
    Date::Time today;
  }
  ```
5 Memory management

- Two operators in C++:
  - Memory allocation with `new`
    ```
    type *pointer_to_type;
    pointer_to_type = new type;
    ```
    - If allocation fails a `std::bad_alloc` exception is thrown (or a `NULL` pointer is returned)
    - C: No explicit type casting necessary
  - Memory deallocation with `delete`
    ```
    delete pointer_to_type;
    ```
    - Programmer is responsible for deallocation
    - Pointer is still accessible after deallocation
    - Common source of programming errors
    - `delete` for a `NULL` pointer is allowed

- C: memory management with `malloc` and `free`
5 Memory management (2)

Example:

```c
int *x=0;       // okay
delete x;       // okay
x = new int;    // okay
delete x;       // okay
delete x;       // wrong
```

Special syntax for arrays:

```c
int *ap = new int[7];
delete[] ap;     // not delete ap !!!
```

Never ever mix `malloc`/`free` with `new`/`delete`

➤ Caution: E.g. `strdup` does an implicit `malloc`

Unfortunately no Garbage Collection in C++
6 Function overloading

- Same function name for different implementations
  - Works for pure C functions and C++ methods

- Overloaded functions are distinguished by:
  - Number of parameters
  - Type of parameters
  - Sequence of parameter types
  - *Not:* Return type of function (Return value may be ignored)

- Example:

  ```
  void Print(); // okay
  void Print(int, char*); // okay
  int Print(float); // okay
  int Print(); // error, not distinguishable
  ```
7 Reference variables

- Address operator & in variable declaration
  
  ```
  type &reference_variable = variable_of_type;
  ```

- Reference variable
  - No real variables
  - Proxy or alias for another variable
  - Must be initialized during declaration (with lvalue - a thing that can be on the left side of an assignment, i.e. it can take a value)

- Example:
  ```
  int x = 5;    // variable
  int &rx = x;  // reference to x
  x = 6;       // x==6 and rx==6
  rx++;        // x==7 and rx==7
  ```

- Operations on reference variables affect the referenced variables

- Similar to pointers with implicit dereferencing but less flexible
7 Reference variables (2)

- Reference parameters
  - Allow implicit call-by-reference semantics
  - No pointers necessary
  - Caller writes down call with normal syntax
  - Disadvantage: syntax of call does not show semantics

Example:

```cpp
#include <iostream>

void increment(int& x) {
    x++;   
}

void main() {
    int x = 5;
    increment( x );
    cout << "x=" << x << "\n";  // x==6
}  
```
7 Reference variables (3)

- Returning references is also possible

- Function returns a variable (*lvalue*) not a value

```c
int global = 0; // global variable

int& func() {
    return global; // returns reference to global
}

int main() {
    int x;
    x = func() + 1; // x = global + 1;
    func() = x; // global = x;
}
```

- Returning references to local variables is forbidden

```c
int& func() {
    int x = 0;
    int& rx = x;
    return rx; // forbidden
}
```
8 Default parameters

- Function parameters may contain *default* values

- Will be used when the actual parameter in a call is missing
  ➔ Only at the end of the parameter list, no gaps allowed

Example:

```c
void print(char* string, int nl = 1);

print( "Test", 0 );
print( "Test" ); // is equal to print( "Test", 1 )
print(); // wrong, char* parameter is missing
```

Caution: overloading and default parameters may generate ambiguities

```c
void print(char* string);
void print(char* string, int nl = 1);

print( "Test" ); // which function ???????????
```
9 Constants

- Reserved word `const` modifies declaration
  - `const` variables are read-only (`final` in Java)
  - Initialization during declaration

- Example:

```c
const int k = 42;
char* const s1 = "Test1";
const char* s2 = "Test2";
const char* const s3 = "Test3";

k = 4;        // error: k is const
s1 = "New test";  // error: pointer is const
*s1 = 'P';  // okay, characters are not const
s2 = "New test";  // okay, pointer is not const
*s2 = 'P';  // error: characters are const
```

- Should be preferred to `#define`, because managed by the compiler
  - Definition of local constants
  - Pointer to constants possible (like pointers to variables)
D.3 Objects and Classes in C++

- Extension of *structs*
- Classes
- Visibility
- Object creation
- Object access
- Member functions (methods)
1 Extension of structs

- New concept for structs
  - Every struct defines a type
  - Local functions in structs

- Example:

```cpp
struct Person {
    char* name;
    int age;

    void setName(char*);
    void setAge(int);
};
```

- Disadvantage: unrestricted access to all parts from the outside
# Classes

- Class declaration in C++ with reserved word `class`:

  ```
  class class_name {
    Declaration of member variables and functions
  };
  ```

  - Contains declaration of data and methods (in C++ called `members`)
  - Sending a message means in C++: accessing a member

- Example:

  ```
  class Person
  {
    char* name;
    int age;
    
    void setName( char* );
    void setAge( int );
  };
  ```
3 Visibility

Different visibility for parts of an object:

- **private:** Member can be accessed only from within its class
- **public:** Member can be accessed from anywhere
- **protected:** like `private`, but subclasses have access

- Parts can be declared in any order and can be repeated
- **public** parts are the interface for other objects
- Default visibility is `private`!
3 Visibility (2)

Example:

```cpp
class Person {
private:
    char* name; // private member variables
    int age;
public:
    void setName( char* ); // public member functions
    void setAge( int );
};
```
4 Object creation

- Syntax is the same like declaring a variable

- Static creation:

  Person peter;
  Person john;

  ◆ Object deleted when identifier goes out of scope

- Dynamic creation:

  Person* peter;
  peter = new Person; // object is created now

  ◆ Object explicitly deleted

  delete peter; // object is deleted now
5 Object access

Access from outside the object

- Private member variables are not accessible
- Private member functions are not accessible
- Public member variables and functions are accessible

Access operators

- As in structs with the dot operator.
- With pointers to objects use the arrow operator ->

Example:

```cpp
Person peter;
Person* john = new Person;

peter.setName( "Peter Smith" );  // okay, public
cout << peter.name;             // error, private
john->setAge( 35 );             // okay, public
cout << john->age;              // error, private
delete john;
```
6 Member functions (methods)

■ Definition *within* the class declaration:
  ◆ Function body comes directly after the declaration (as in Java)
  ◆ Function becomes automatically *inline*
  ◆ Usually used in header files (*h, H* or *hh*)

■ Definition *outside* the class:
  ◆ Within the class only declaration of the function prototype
  ◆ During definition you first have to name the class
  ◆ Afterwards comes the function name separated by the scope operator ::
  ◆ Usually used in implementation files (*C, cc, or cpp*)
7 Member functions (methods) (2)

Example:

◆ Header (Person.h)

```cpp
#ifndef PERSON_H
#define PERSON_H

class Person {
private:
    char* name;
    int age;

public:
    void setName( char* n ) { // inline
        name = n;
    }
    void setAge( int );
};
#endif
```

◆ Implementation (Person.cpp)

```cpp
#include "Person.h"

void Person::setAge( int i ) {
    age = i;
}
```
8 Constant Objects

- Variable declared `const`
  - Initialized when declared
  - Cannot be changed afterwards
  - Very useful for method parameters

- Silly example:
  ```c++
  const Person nobody;
  ```

- Only operations that do not alter the object may be executed
  - Easy for member variable access
  - Methods that do not alter members

- How does the compiler know?
  - It does not!
  - Needs a hint from the programmer
8 Constant objects (2)

- Methods may be declared `const`

- `Const` methods do not change the object they are called at

- Example:

```cpp
class Person {

private:
    char* name;
    int age;

public:
    int getAge() const {
        return age;
    }
};
```
D.4 Constructors and Destructors

- Constructors
- Destructors
- Member objects
- Copy constructor
- Arrays of objects
1 Constructors

- Like in Java
- Class method
- Method name is the name of the class
- *No* return type (not even `void`)
- Different constructors through overloading
- Declaration usually in the `public` part of the class
- Purpose: New object is automatically initialized after creation
  - Constructor has to put object in a consistent state
- Compiler creates a minimal default constructor (no arguments) if not declared in class
1 Constructors (2)

- Called during:
  - Creation of an object via the operator `new`
  - Creation of a static object

- Minimal default constructor (created by the compiler):

```cpp
Person::Person() {}
```

- Default constructor (replaces minimal constructor):

```cpp
Person::Person() {
    name = NULL;
    age = 0;
}
```
1 Constructors (3)

Other constructors:

```cpp
Person::Person( char *n, int i = 0 ) {
    name = n;
    age = i;
}
```

- Default values are possible
2 Destructors

- Similar to `finalize` in Java
- Class method
- Method name is the name of the class with ~ in front
- No return type (not even `void`)
- Only one destructor possible, no overloading
- Destructors have no parameters
- Declaration usually in the `public` part of the class
- Purpose: Cleaning up before deleting the object
- Compiler creates a default destructor (does nothing) if not declared in class
2 Destructors (2)

- Called during:
  - Destruction of an object via the operator `delete`
  - Leaving the scope of a static object

- Minimal default destructor (created by the compiler):
  
  ```cpp
  Person::~Person() {}
  ```
3 Member objects

- Objects of other classes as members within a class

```cpp
class Workplace {
    Person worker;
    ...
};
```

- Access via operators . und -> as usual

- Problems during initialization:
  - Will the constructors of the member objects be called?
  - If yes, when will they be called?
  - Which constructors will be called?
  - Which parameter values will be used?

- Similar problem with object destruction:
  - When will the destructors of the member objects be called?
  - No problem: There is only one destructor which has no parameters
3 Member objects (2)

- Definition of an initialization list in the constructor:

```cpp
class_name::class_name( parameter_list )
    : member1( parameters ), member2( parameters ), ...
{ ... }
```

- Example:

```cpp
class Person {
public:
    Person( char* );
    ...
};

class Workplace {
    Person worker;
    ...
};

Workplace::Workplace( char* name )
    : worker( name )
{ ... }
```
4 Copy constructor

When is a copy constructor used?

- Object is a value parameter in a function call (*call-by-value*)
- Object is a return value of a function
- Initialization of an object with an existing object

```cpp
Person peter( john );
```

Example:

```cpp
Person::Person( const Person& p ) {
    name = p.name;
    age = p.age;
}
```

Important: use reference operator &

Default copy constructor (created by the compiler) copies bit-by-bit
5 Arrays of objects

Static arrays

- Without initialization
  - For all elements the standard constructor is called

  ```
  Person test[4];  // calls 4 times Person::Person()
  ```

- With initialization
  - Initialization expressions are used for the first elements, for the rest
    the standard constructor is called

  ```
  Person test[4] =
  { "Peter", Person("John") };
  // test[0] and test[1]: Person::Person( char* )
  // test[2] and test[3]: Person::Person()
  ```
5 Arrays of objects (2)

- Dynamically allocated arrays
  - The default constructor is always called
  ```
  Person *table;
  table = new Person[4];  // 4 times Person::Person()
  ```

- Access as usual via operator []
  ```
  Person table[4];
  table[0].SetName( "Peter" );
  ```

- Destruction of arrays
  - For all elements the destructor is called
  - Dynamically allocated arrays have to be deleted via delete[]
D.5 Inheritance

- Single Inheritance
- Scope operator
- Modification of visibility
- Constructors und Destructors
- Type casting
- Virtual methods
- Polymorphism
- Virtual destructors
- Abstract base class
- Multiple inheritance
1 Inheritance

- Like in Java
- Reuse of existing implementations (classes)
- New class *inherits* features from the existing class

Denotation:

- Class that inherits: Subclass
- Class that is inherited from: Superclass or Base class

In C++: *Derivation* of new classes from existing ones

Derivation/Inheritance is a "is-a" relation

One base class: Single inheritance, otherwise Multiple inheritance
1 Inheritance (2)

Syntax:

```cpp
class subclass : [modifier] superclass1, [modifier] superclass2, ...
{
    Declaration of new member variables and
    new or re-implemented member functions (methods)
}
```

Not inherited

- Constructors
- Destructor
- Assignment operator
1 Inheritance (3)

Rule in C++: Everything that is not re-implemented, is inherited

```
class Person { ...
public:
    void print();
    void setName( char* );
};

class Employee : public Person { ...
public:
    void print();
    void setName( char* );
    void setSalary( float );
};
```

behaves like

```
class Employee : public Person { ...
public:
    void print();             // from Employee
    void setName( char* );   // from Person
    void setSalary( float ); // from Employee
};
```


2 Scope operator

- Often access to re-implemented methods of a superclass is needed
- **Scope-Operator** ::
  
  ```
  class_name::method( ... )
  ```

- No **super** as in Java

- Example:

  ```
  class Employee : public Person { ...
  public:
    void print() {
      // print();    // no, endless recursion
      Person::print();
      cout << "Salary:" << salary << "\n";
    }
  }
  ...
  Employee a;
  a.print();
  a.Person::print();
  ```
3 Modification of visibility

- Specification how members of a base class should be visible in the subclass

- **public** modifier for inheritance:
  - public stays public
  - protected stays protected
  - private not accessible in subclass

- **protected / private** modifiers for inheritance:
  - public becomes protected / private
  - protected becomes protected / private
  - private not accessible in subclass
3 Modification of visibility (2)

- Usually only **public** inheritance is used

- **protected** and **private** inheritance make the interface smaller
  ➔ Subclass is no longer a subtype of the superclass

- Default modifier is **private**!
4 Constructors

- Initialization of superclass members via superclass constructors

- Subclass constructor calls superclass constructor via *initialisation list*

  ```
  class_name::class_name( parameter_list )
  : superclass1( parameters ), superclass2( parameters ), ...
  { ... }
  ```

- Superclass constructors are called *before* subclass constructor

- Subclass members are initialized *after* superclass members

- Example:

  ```
  Employee::Employee( char* n, int a, float s )
  : Person( n, a ), salary( s )
  {
  ... 
  }
  ```
5 Destructors

- Destruction of superclass members has to happen in the destructor of the superclass.

- Superclass destructor is *automatically* called *after* the subclass destructor (other way round as with constructors).

- Example:

```cpp
Employee::~Employee()
{
    // Destroy only new members in employee
}
```
6 Pointers to objects

- Pointer to a subclass object can be assigned to a pointer to a superclass object:
  - Subclass is extension of superclass, therefore also subtype

- Doesn’t work the other way round:
  - Explicit type casting necessary
  - Not very nice but sometimes unavoidable

- General rule:
  Specialized type can be assigned to a more general type.

- Pointers have a static and a dynamic type:
  - static: Class from pointer declaration
  - dynamic: Class of the object that the pointer points to (can be the class from the pointer declaration or any subclass of it)

- Static type defines accessible interface (members and methods)
7 Type casting

C-style casts:

```cpp
class Person { ... };
class Employee : public Person { ... };
...
Employee* e = new Employee; // okay
Person* p = new Person; // okay
Person* pe = e; // okay
Employee* e1 = p; // compiler error
Employee* e2 = pe; // compiler error
Employee* e3 = (Employee*) pe; // okay
Employee* e4 = (Employee*) p; // unrecognisable error
```

- Compiler doesn’t look at dynamic type
- Before ANSI-C++ there was no Run-Time Type Information (RTTI)
- Avoid them !!!

In ANSI-C++ use `static_cast` or `reinterpret_cast` for low-level type casting

```cpp
type variable = static_cast<type>( parameter );
ype variable = reinterpret_cast<type>( parameter );
```
7 Type casting (2)

- Dynamic casts:

  ```
  type variable = dynamic_cast<type>( parameter );
  ```

  - Uses Run-Time Type Information to determine if valid
  - Like all Java casts
  - Returns NULL if cast fails, no exceptions thrown !!!

- Example:

  ```
  class Person { ... };
  class Employee : public Person { ... };
  ...
  Employee* e = new Employee;
  Person* p = new Person;
  Person* pe = e;
  Employee* e3 = dynamic_cast<Employee*>( pe ); // okay
  Employee* e4 = dynamic_cast<Employee*>( p ); // returns NULL
  ```

- Additionally `const_cast` for casting away constness
8 Virtual methods

■ Up to now:
  ◆ Type of pointer (static type) not type of object pointed to (dynamic type) defines interface semantics of a call
  ◆ Access to subclass members only after type casting of the pointer

■ Aim is polymorphism: Execution of the suitable subclass method without explicitly knowing the subclass (*This is what you always have in Java!*)

■ Solution: *Virtual* methods
  ➔ Object defines semantics, not the pointer

■ Syntax with reserved word *virtual*:

```cpp
class class_name {
    virtual return_type method_name( parameter_list )
    { ... }
};
```

■ *virtual* has to be specified in the base class and is inherited
9 Polymorphism

Example:

```cpp
class Person { ... 
 public: virtual void print(); 
};
class Employee: public Person { ... 
 public: void print(); 
};
...
Person* p = new Person;
Person* pe = new Employee;
p->print(); // Person::print()
pe->print(); // Employee::print()
```

- Called method is determined at run-time
- Called object has a defined type, therefore method to be called is unambiguous
- Compiler generates *vtables* (jump tables for virtual methods)
  - Every object contains pointer to vtable of its class, therefore larger objects
10 Virtual destructors

- Dynamically allocated objects may be assigned to superclass pointers

- Problem: If object is deleted, only the superclass destructor is called because of the static type of the superclass pointer
  - Objects are not destroyed properly

- Solution: Virtual destructor:

```cpp
class class_name {
    virtual class_name::~class_name()
    {
        ...;
    }
};
```

- `virtual` has to be specified in the base class

- Is inherited by all subclasses although destructor names are different in subclasses
11 Abstract classes

- Abstract classes:
  - No all methods that were declared are also implemented
  - There can be no instances/objects of this class
  - Subclasses can only have instances if all declared methods are also implemented

- Abstract classes can be used
  - As superclasses without instances (class with abstract methods in Java)
  - To define a type/interface (interface in Java)

- Syntax for methods that are not implemented (pure virtual):

  ```
  class class_name {
      virtual return_type method_name( parameter_list ) = 0;
  }
  ```

- Pointers to abstract classes are possible but have to be initialized with object of a subclass that is not abstract
12 Multiple inheritance

- Subclass has *multiple* superclasses (forbidden in Java)
- Subclass contains *every* superclass as an implicit part
- The subclass constructor can call constructors of every superclass in the initialization list

```cpp
class Base1 { ... 
public: Base1( int, char* ); 
};
class Base2 { ... 
public: Base2( int, float ); 
};
class Derived : public Base1, public Base2 { ... 
public: Derived( char *s, int i ) : 
    : Base1( i, s ), Base2( i, 4.2 ) { } 
};
```

- When an object of the subclass is destroyed the destructors of all superclasses are called
12 Multiple inheritance (2)

- Problem: Ambiguities through name clashes

- Two or more superclasses have the same member:
  - Member variables with the same name
  - Methods with the same name and the same parameters

- First automatic resolution of ambiguities, then access control (visibility)
  ➔ Making one member private doesn’t help

- Explicit resolution of name clashes for variables:
  - Specify the superclass before the variable name using the scope operator ::

- Possible solution for methods:
  - Reimplement method and use the desired superclass method(s) via the scope operator ::
12 Multiple inheritance (3)

- Superclass contains common features (intersection set) of all subclasses (generalization)

- Problem with multiple inheritance: Common base class is contained multiple times

- Example:
12 Multiple Inheritance (4)

- Implementation with a *virtual* base class

- Example:

```
class subclass : virtual public superclass {
    Declaration of member variables and functions
};
```

**Syntax for *virtual inheritance***:
12 Multiple inheritance (5)

Example:

```cpp
class Boat {
protected: char* name;
public: Boat( char* n ) : name( n ) { }
};

class SailingBoat : virtual public Boat {
protected: Sail mySail;
public: SailingBoat( char* n ) : Boat( n ) { }
};

class MotorBoat : virtual public Boat {
protected: Motor myMotor;
public: MotorBoat( char* n ) : Boat( n ) { }
};

class SailingBoatWithMotor : public SailingBoat, public MotorBoat {
public: SailingBoatWithMotor( char* n )
    : Boat( n ), SailingBoat( n ), MotorBoat( n )
    { }
};
```
D.6 Exceptions

- Exception syntax
- How exceptions work
- Example: Resource allocation
- Differences to Java
- Exceptions in ANSI C++
- Solution for the new problem
1 Exception Syntax

- 3 reserved words:
  - `try` tries to execute the following block
  - `throw` creates an exception and starts exception handling
  - `catch` catches an exception from the `try` block and processes it in the following block

- Example:

```java
try {
    computation
    if error: throw exception_class( ... );
} catch( exception_class variable ) {
    exception processing
}
```
2 How Exceptions Work

- Linear processing of the `catch` list
- Grouping of error types through inheritance
  - `catching` a base class also catches all subclasses
- Exceptions are propagated upwards until a `catch` clause is found who’s type matches the type of the exception
- All destructors are called when leaving a block because of an exception
- There is no suitable `catch` clause ➔ Program is aborted
- `catch(...)` catches all exceptions
3 Differences to Java

■ No **finally** block

■ Similar functionality can be achieved through:

```java
catch( ... ) {
    // clean up
    throw;       // re-throw caught exception
}
```

♦ Attention: Not executed if there are other catch clauses that match or when no exception was thrown

■ Exceptions do **not** belong to a method’s type
  ➤ Can be thrown anywhere
  ➤ Compiler cannot check if all thrown exceptions are caught at some point
4 Exceptions in ANSI C++

- Functions and methods *may* specify an exception list

- Reserved word `throw` in function prototype:

```cpp
return_type method_name ( parameter_list ) throw ( exception_list ) {
    Body of method
}
```

- Similar to `throws` in Java

- Exception list is a guarantee to the caller

- `std::unexpected()` is called if an exception that is not in the list leaves the function

- Functions without an exception list may still throw any exception
D.7 Odds and Ends

- This pointer
- Static members
1 This pointer

- **this** points to the called object itself
- Implicit parameter in every method call
- Looks like: `class_name * const this`
- If method is `const`: `const class_name * const this`

Example:

```cpp
class Person {
    char* name;

public:
    void print() { cout << this->name; } // = name
    void insertInto( List* l ) { l->insert(this) }
    void prettyPrint() {
        cout << "Data: ";
        this->print(); // = print()
    }
};
```
2 Static members

- Normally every object contains its own set of variables
- Except for: member variables declared as `static`
- `static` members exist once for each class, no matter how many objects of that class were created
- Makes it possible to use it as a shared variable for all instances of a class
  ➔ Class variable
- Access rights can be specified as with instance variables
2 Static members (2)

- Global initialization outside the class (access rights don’t matter for initialization)

- Example:

```cpp
class BankAccount {
    static float interestRate;
    ...
};

float BankAccount::interestRate = 0.5;
```
2 Static members (3)

- Methods that only access other static members may be declared static themselves

- static methods can be called without an object

- No access to dynamic (per instance) members of the class

- No this pointer
D.8 Operators

- Operator overloading
- Global operators
- Operators as members
- Binary operators
- Unary operators
- Allocation operators
1 Operator overloading

- In C++ (in contrast to Java) operators can be overloaded to work with new types

- Looks like function or method overloading

- New reserved word `operator`

```
return_type operator operator ( parameter_list )
{ ... };
```

- Operators that can be overloaded

```
+  -  *  /  %  ^  &  |  ~  !
=  <  >  +=  -=  *=  /=  %=  ^=  &=
||=  <<=  >>=  <<=  >>=  ==  !=  <=  >=  &&  ||
||  ++  --  ,  ->*  ->  ()  []  new delete
```

- Operators that cannot be overloaded

```
.  .*  ::  ?:
```

- Operator precedence and associativity cannot be changed
2 Global operators

- Work like (global) functions
- Can be friends of classes
- Always have the object itself as a parameter

Example:

```cpp
class Person {
    char* name;
    friend ostream& operator << ( ostream&, Person );
};

ostream& operator << ( ostream& os, Person& p ) {
    os << p.name;
    return os;
}

...Person p( "Peter" );
cout << p;       // call as operator
operator << ( cout, p ); // call as function
```
3 Operators as members

- Operator is treated like a method of the class
- Access to all members, there is a `this` pointer
- One parameter less than the same global operator (object via `this`)

Example:

```cpp
class Complex { double real, imag;
public: Complex( double r=0, double i=0 )
    : real( r ), imag( i ) { }
    Complex operator + ( const Complex& ) const;
};

Complex Complex::operator + ( const Complex& c ) const {
    Complex result( real+c.real, imag+c.imag );
    return result;
}
...
Complex c1, c2, c3;
c1 = c2 + c3; // normal call
c1 = c2.operator + ( c3 ); // generated by the compiler
```
4 Binary operators

- As a global operator: Two parameters
- As a member: One parameter
- Examples (only member operators):
  - Assignment operator
    
    \[
    \text{class} & \ 	ext{class} & \ : \ : \ 	ext{operator} = ( \ 	ext{class} & \ )
    \]

  - Index operator
    
    \[
    \text{element\_type} & \ 	ext{class} & \ : \ : \ 	ext{operator} [] ( \ \text{index\_type} \ )
    \]

    - Index type usually `int`
  - Arithmetic operators and their combination with the assignment operator
5 Unary operators

- As a global operator: One parameter
- As a member: No parameters
- Except for: Postfix operators

Examples (only member operators):

- Prefix increment operator
  ```cpp
class& class::operator ++ ( )
  ```

- Postfix increment operator
  ```cpp
class& class::operator ++ ( int )
  ```
  - int is just a dummy parameter to distinguish it from the prefix version

- Cast operator
  ```cpp
class::operator target_type ( )
  ```
  - Target type of the cast is operator name and return type at once
6 Allocation operators

- Custom memory allocation strategies
- Global operators for all classes
- Operators for allocation on a per-class basis
  - Override global operators
  - E.g. memory pool for short-lived objects
- Operator syntax
  - Allocation operator
    ```
    void* operator new ( size_t )
    ```
  - Deallocate operator
    ```
    void operator delete ( void * )
    ```
  - For arrays operators `new[]` and `delete[]`