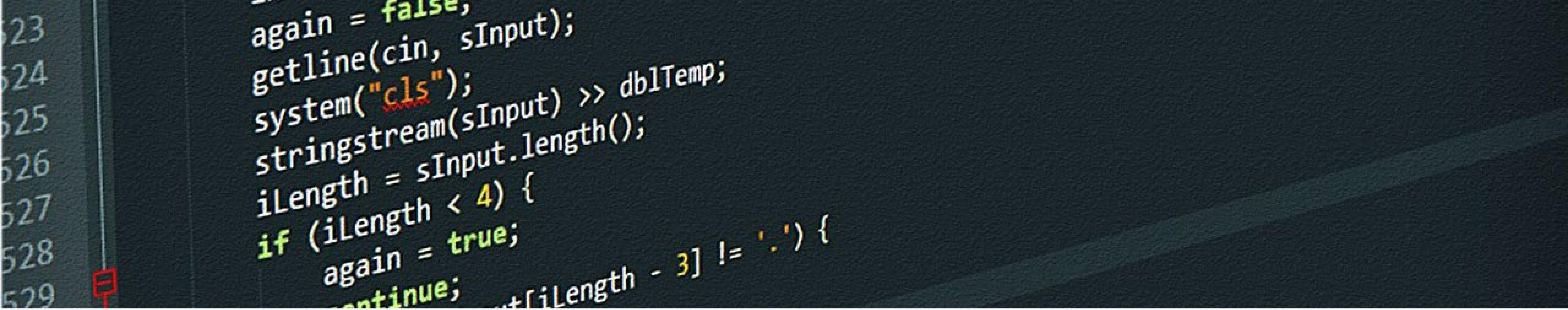


Thomas

C23-08 Polymorphism, typecasting

C23 - Advanced Algorithms and Programming

v5

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Slides created using material from R. Peters and P. Puschmann / HTW Berlin

Polymorphism

Example – Without polymorphism (1/2)

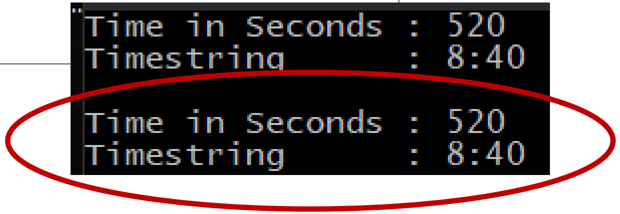
```
#pragma once
#include <iostream>
#include <string>
#include <sstream>
#include <iomanip>

class Timer
{
public:
    Timer(int min = 0, int sec = 0)
        : m_minute{ min }, m_second{ sec } { }

    // Time in seconds
    double getTime() const { return m_minute * 60 + m_second; }

    std::string toString() const {
        std::stringstream s;
        s << m_minute << ":" << m_second; // Format: mm:ss
        return s.str();
    }

protected:
    int m_minute, m_second;
};
```

```
class HighPrecisionTimer : public Timer
{
public:
    HighPrecisionTimer(int min = 0, int sec = 0, int milli = 0)
        : Timer{ min, sec }, m_millisecond{ milli } { }

    // Time in seconds (overwritten)
    double getTime() const
    { return Timer::getTime() + m_millisecond / 1000.0; }

    // overwritten
    std::string toString() const {
        std::stringstream s;
        // Format: mm:ss.MMM
        s << m_minute << ":" << m_second << ".";
        s << std::setw(3) << std::setfill('0') << m_millisecond;
        return s.str();
    }

private:
    int m_millisecond;
};
```

Polymorphism

Example – Without polymorphism (2/2)

```cpp
#include "Timer.h"

void printTime(const Timer& t)
{
    std::cout << "Time in Seconds : " << t.getTime() << "\n";
    std::cout << "Timestring      : " << t.toString() << "\n";
}

int main()
{
    Timer t1(8, 40); // 8:40min
    printTime(t1);

    std::cout << "\n";

    HighPrecisionTimer t2(8, 40, 20); // 8:40min + 20 milliseconds
    printTime(t2);
}
```



Polymorphism

Example – Explanation

- As soon as the `HighPrecisionTimer` object is used via a reference to a `Timer` object, the functions of the base class (`Timer`) are called.

  - ➢ In the function `printTime()` the functions of the base class are called:
    - `Timer::getTime` instead of `HighPrecisionTimer::getTime`
    - `Timer::toString` instead of `HighPrecisionTimer::toString`

- The function call is already defined at compile time: **"Early Binding"**

- This behavior can be influenced with the keyword **`virtual`** …

Polymorphism
Motivation

- If a virtual function of a class is called via a base class pointer or reference, the relevant function implementation of the derived class is called

- This mechanism completely hides the different implementations of the virtual function
- The class „behind the pointer / reference“ can take several different forms, depending on the actual (derived) class of the instance.
- Hence the term „polymorphism“

- Advantage: Code reusability
  - The actual class of the referenced instance is not important
    Program code that uses a base class is also valid for all derived classes
  - Derived classes can override certain functions of the base class to achieve a specific behavior

Polymorphism

Example – With polymorphism (1/2)


```cpp
#pragma once
#include <iostream>
#include <string>
#include <sstream>
#include <iomanip>

class Timer
{
public:
    Timer(int min = 0, int sec = 0) : m_minute{ min }, m_second{
sec } { }

    // Time in seconds
    virtual double getTime() const {
        return m_minute * 60 + m_second;
    }

    virtual std::string toString() const {
        std::stringstream s;
        s << m_minute << ":" << m_second; // Format: mm:ss
        return s.str();
    }

protected:
    int m_minute, m_second;
};
```

```cpp
class HighPrecisionTimer : public Timer
{
public:
    HighPrecisionTimer(int min = 0, int sec = 0, int milli = 0)
        : Timer{ min, sec }, m_millisecond{ milli } { }

    // Time in seconds (overwritten)
    double getTime() const
    { return Timer::getTime() + m_millisecond / 1000.0; }

    // overwritten
    std::string toString() const {
        std::stringstream s;
        // Format: mm:ss.MMM
        s << m_minute << ":" << m_second << ".";
        s << std::setw(3) << std::setfill('0') << m_millisecond;
        return s.str();
    }

private:
    int m_millisecond;
};
```

Polymorphism

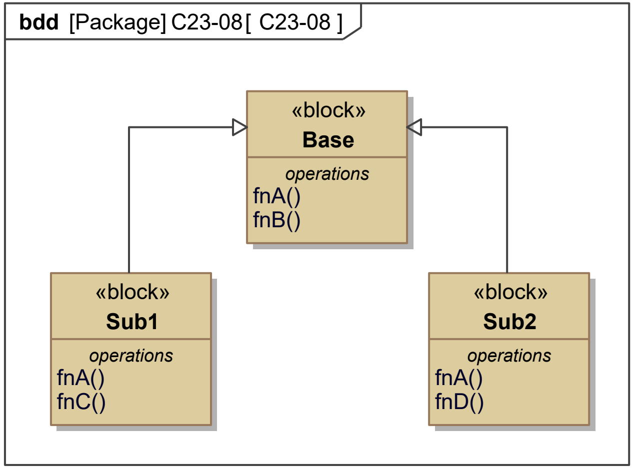
Example – With polymorphism (2/2)

```cpp
#include "Timer.h"

void printTime(const Timer& t)
{
    std::cout << "Time in Seconds : " << t.getTime() << "\n";
    std::cout << "Timestring      : " << t.toString() << "\n";
}

int main()
{
    Timer t1(8, 40); // 8:40min
    printTime(t1);

    std::cout << "\n";

    HighPrecisionTimer t2(8, 40, 20); // 8:40min + 20 milliseconds
    printTime(t2);
}
```

```
Time in Seconds : 520
Timestring      : 8:40

Time in Seconds : 520.02
Timestring      : 8:40.020
```

Polymorphism
Summary

- Situation
  - Two instances of the derived classes:
    `Sub1 s1;` `Sub2 s2;`
  - A pointer to base class
    `Base* pS1 = &s1;`
  - A reference to base class
    `Base& rS2 = s2;`



- If `Base::fnA` is <u>not a virtual function</u>, a call to `pS1->fnA()` or `rS2.fnA()` calls the function `Base::fnA()`

- If `Base::fnA` is <u>a virtual function</u>, then a call to
  - `pS1->fnA()` calls the function `Sub1::fnA()`
  - `rS2.fnA()` calls the function `Sub2::fnA()`

Note: Calls to `pS1->fnC()` or `rS2.fnD()` are not possible!

Polymorphism

Virtual member functions – late binding

- For virtual member functions in the declaration, the actual function to be called is decided at runtime when accessing it via pointers or references
  → **„Late binding“** or **„Dynamic binding“**

- Polymorphism does not work without late binding because the compiler can only know at runtime which class is invoked via a reference or pointer.

Polymorphism
Simple example

```cpp
#include < iostream>

class A {
public:
    void print() {
        std::cout << "Hello A" << std::endl;
    }
};

class B : public A {
public:
    void print() {
        std::cout << "Hello B" << std::endl;
    }
};

int main() {
    B b;
    A& refA = b; // 'b' object via a reference to 'A
    refA.print();
}
```

```
Hello A
```

```cpp
#include <iostream>

class A {
public:
    virtual void print() {
        std::cout << "Hello A" << std::endl;
    }
};

class B : public A {
public:
    void print() {
        std::cout << "Hello B" << std::endl;
    }
};

int main() {
    B b;
    A& refA = b; // 'b' object via a reference to 'A
    refA.print(); // late binding
}
```

```
Hello B
```

Polymorphism

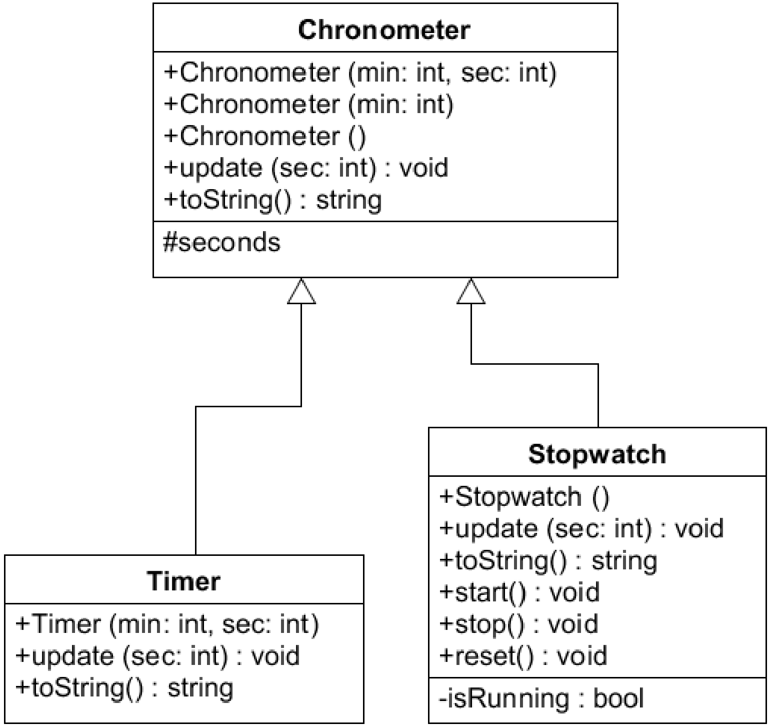
Simple example – several hierarchy levels

```cpp
#include < iostream>
class A {
public:
    virtual void print() { std::cout << "Hello A" << std::endl; }
};

class B : public A {
public:
    // when overwriting, the 'virtual' is also inherited!
    void print() { std::cout << "Hello B" << std::endl; }
};

class C : public B {
public:
    void print() { std::cout << "Hello C" << std::endl; }
};

int main() {
    C c;
    A& refA = c; // 'c' object via a reference to 'A'
    B& refB = c; // 'c' object via a reference to 'B'
    refA.print(); // late binding
    refB.print(); // late binding
}
```

```
Hello C
Hello C
```

Polymorphism
Chronometer example

The example shows that late binding only happens at runtime, applying interactive user input.



Polymorphism
Chronometer example

Chronometer.h

```cpp
#pragma once
#include <string>
#include <sstream>
#include <iomanip>
#include <iostream>

class Chronometer
{
public:
    Chronometer(int min = 0, int sec = 0) : m_seconds{ min * 60 + sec } { }

    virtual void update(unsigned int sec) { m_seconds += sec; }

    virtual std::string toString() { // Output format: 'h:mm:ss'.
        std::stringstream s;
        s << std::setfill('0') << m_seconds / 3600; // hours
        s << ":" << std::setw(2) << (m_seconds / 60) % 60; // minutes
        s << ":" << std::setw(2) << m_seconds % 60; // seconds
        return s.str();
    }

    virtual ~Chronometer() {
        std::cout << "Destructor called by Chronometer!" << std::endl;
    }

protected:
    unsigned int m_seconds;
};
```

Polymorphism
Chronometer example

Timer.h

```cpp
#pragma once
#include "Chronometer.h"

class Timer : public Chronometer
{
public:
    Timer(unsigned int min, unsigned int sec) : Chronometer{ min, sec } { }

    virtual void update(unsigned int sec) {
        m_seconds = m_seconds > sec ? m_seconds - sec : 0;
    }

    virtual std::string toString() {
        if (m_seconds > 0)
            return Chronometer::toString() + " remaining";
        return "Time out!";
    }

    virtual ~Timer() {
        std::cout << "Destructor called by Timer!" << std::endl;
    }
};
```

Polymorphism

Chronometer example

Stopwatch.h

```
#pragma once
#include "Chronometer.h"

class Stopwatch : public Chronometer
{
public:
    Stopwatch() : Chronometer(0), m_isRunning(false) { }
    void start() { m_isRunning = true; }
    void stop() { m_isRunning = false; }
    void reset() { m_seconds = 0; }

    virtual void update(unsigned int sec) {
        if (m_isRunning)
            m_seconds += sec;
    }

    virtual std::string toString() { return Chronometer::toString() + " passed"; }

    virtual ~Stopwatch() {
        std::cout << "Destructor called by Stopwatch!" << std::endl;
    }

private:
    bool m_isRunning;
};
```

Polymorphism
Chronometer example

Chronometer.cpp

```
#include "Timer.h"
#include "Stopwatch.h"
#include <thread>
#include <chrono>

int main() {
    int choice, seconds{ 0 };
    std::cout << "Stopwatch (0) or Timer (1): "; std::cin >> choice;
    std::cout << "Duration in seconds: "; std::cin >> seconds;
    Chronometer* pChrono;
    if (choice == 0) {
        Stopwatch* pStopwatch = new Stopwatch();
        pStopwatch->start();
        pChrono = pStopwatch;
    }
    else {
        pChrono = new Timer(0, seconds);
    }

    for (int i = 0; i < seconds; i++) {
        pChrono->update(1);

        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

        std::cout << pChrono->toString() << std::endl;
    }
    // Destructor of Chronometer must be virtual,
    // otherwise the destructor of derived class is not called
    delete pChrono;
}
```

Polymorphism

Chonometer example

Input:

0 (Stopwatch)

8 seconds

```
Stopwatch (0) or Timer (1): 0
Duration in seconds: 6
0:00:01 passed
0:00:02 passed
0:00:03 passed
0:00:04 passed
0:00:05 passed
0:00:06 passed
Destructor called by Stopwatch!
Destructor called by Chronometer!
```

Input:

1 (Timer)

8 seconds

```
Stopwatch (0) or Timer (1): 1
Duration in seconds: 6
0:00:05 remaining
0:00:04 remaining
0:00:03 remaining
0:00:02 remaining
0:00:01 remaining
Time out!
Destructor called by Timer!
Destructor called by Chronometer!
```

Polymorphism

Chronometer example – comparison

- Without virtual functions the loop would have to be implemented twice in `main()` to call the specific `toString` or `update` function:

```cpp
Chronometer* pChrono;
if (choice == 0) {
    Stopwatch* pStopwatch = new Stopwatch();
    pStopwatch->start();
    pChrono = pStopwatch;
}
else {
    pChrono = new Timer(0, seconds);
}

for (int i = 0; i < seconds; i++) {
    pChrono->update(1);
    std::this_thread::sleep_for(
        std::chrono::milliseconds(1000));
    std::cout << pChrono->toString() << std::endl;
}
delete pChrono;
```

With virtual functions

```cpp
if (choice == 0) {
        Stopwatch stopwatch;
        stopwatch.start();
        for (int i = 0; i < seconds; i++) {
            stopwatch.update(1);
            std::this_thread::sleep_for(
                std::chrono::milliseconds(1000));
            std::cout << stopwatch.toString() << std::endl;
        }
    }
    else {
        Timer timer(0, seconds);
        for (int i = 0; i < seconds; i++) {
            timer.update(1);
            std::this_thread::sleep_for(
                std::chrono::milliseconds(1000));
            std::cout << timer.toString() << std::endl;
        }
    }
```

Without virtual functions

Polymorphism

Chronometer example – virtual destructor

- If the destructors were not `virtual`, the destructors of `Timer` and `Stopwatch` would not be called via the chronometer reference:

```cpp
class Chronometer
{
public:
    Chronometer(unsigned int min = 0, unsigned int sec = 0)
        : m_seconds{ min * 60 + sec } { }
    // ...

    virtual ~Chronometer() {
        std::cout << "Destructor called by Chronometer!"
            << std::endl;
    }

protected:
    unsigned int m_seconds;
};
```

```
Stopwatch (0) or Timer (1): 0
Duration in seconds: 2
0:00:01 passed
0:00:02 passed
Destructor called by Stopwatch!
Destructor called by Chronometer!
```

```cpp
class Chronometer
{
public:
    Chronometer(unsigned int min = 0, unsigned int sec = 0)
        : m_seconds{ min * 60 + sec } { }
    // ...

    ~Chronometer() {
        std::cout << "Destructor called by Chronometer!"
            << std::endl;

protected:
    unsigned int m_seconds;
};
```

```
Stopwatch (0) or Timer (1): 0
Duration in seconds: 2
0:00:01 passed
0:00:02 passed
Destructor called by Chronometer!
```

Polymorphism

Additional keywords – override and final

- Override specifies that a virtual function overrides another virtual function
- The `final` specifier specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

```
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    void foo() const override; // Error: B::foo does not
                               // override A::foo
                               // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
    void bar() override; // Error: A::bar is not virtual
};

int main() {}
```

```
struct Base
{
    virtual void foo();
};

struct A : Base
{
    void foo() final; // Base::foo is overridden
                      // and A::foo is the final override
    void bar() final; // Error: bar cannot be final
                      // as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden
                         // as it is final in A
};

struct C : B // Error: B is final
{
};
```

Polymorphism

Pure virtual member functions

- Pure virtual member functions have no implementation (not even an empty function body) Instead of a function body an assignment of `0` follows.

  - **Purely virtual function:**

```
virtual std::string toString() = 0;
```

  - … in contrast to empty implementation:

```
std::string toString() { };
```

Abstract classes

- Classes with at least one pure virtual function are called "abstract classes"
- Abstract classes cannot be instantiated
- Pure virtual functions must be overwritten in derived classes and implemented with a function body so that objects can be instantiated

Polymorphism

Pure virtual member functions

Motivation

- Base classes often serve as a template for derived classes
  They define the interface and derived classes implement the interface

- It is *not* useful to implement a interface-defining base class completely
- If the function bodies were empty, the compiler would not return an error if derived classes did not implement these functions. Purely virtual functions, yield a compiler error when instantiation is tried.

Interface

- Abstract classes are called "interface" if all member functions are purely virtual

Polymorphism

Interface example

- Definition of an interface for a login functionality

AccountHandlerInterface.h

```
#pragma once
#include <string>

class AccountHandlerInterface
{
public:
    virtual bool connectToDatabase() = 0; // could be any database
    virtual bool checkPassword(const std::string& rUsername,
    const std::string& rPassw) = 0;
};
```

Polymorphism

Interface example

- Using the interface

LoginGui.h

```
#pragma once
#include "AccountHandlerInterface.h"
#include <iostream>

namespace GUI {
    // could be implemented somewhere in the source code
    bool login(AccountHandlerInterface& accountHander)
    {
        if (accountHander.connectToDatabase() == true)
        {
            std::string username, password;
            for (int i = 0; i < 3; i++) {
                std::cout << "User name: ";
                std::cin >> username;
                std::cout << "Password: ";
                std::cin >> password;
                if (accountHander.checkPassword(username, password)) {
                    std::cout << "Password correct!" << std::endl;
                    return true;
                }
                std::cout << "Password wrong!" << std::endl;
            }
        }
        std::cout << "That was 3 attempts. Program is terminated..." << std::endl;
        return false;
    }
}
```

Polymorphism
Interface example

- Concrete implementation of the interface

HomepageAccountHandler.h

```
#pragma once
#include <string>

class AccountHandlerInterface
{
public:
virtual bool connectToDatabase() = 0; // could be any database
virtual bool checkPassword(const std::string& rUsername,
const std::string& rPassw) = 0;
};
```

Polymorphism
Interface example

- Creating and using an object

Main.cpp

```cpp
#include "LoginGui.h"
#include "HomepageAccountHandler.h"

int main() {
    HomepageAccountHandler handler("Admin", "1234");
    GUI::login(handler);
}
```

```
Connected to MySQL database!
User name: Emma
Password: Emma
Password wrong!
User name: Admin
Password: 1234
Password correct!
```

Typecasting
Up- and Down-Casts

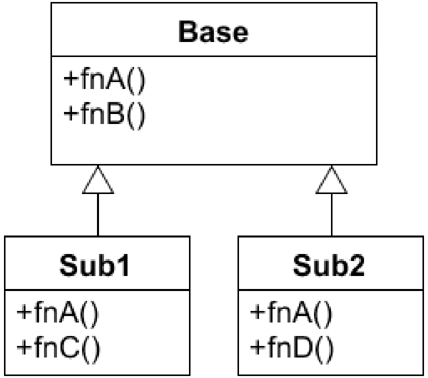
(to cast = convert)

Two instances of the derived classes are given:

```
Sub1 s1;        Sub2 s2;
```

And a pointer and reference of the base class :

```
Base* pS1 = & s1;    Base& rS2 = s2;
```



A conversion to the base class (up-cast) is implicitly possible

`s1` is implicitly converted to a pointer or `s2` to a reference of a base class

A conversion to the derived class (down-cast) requires additional measures

The compiler cannot know which derived class is actually hidden behind `pS1` or `rS2`, therefore the type check must be done at runtime.

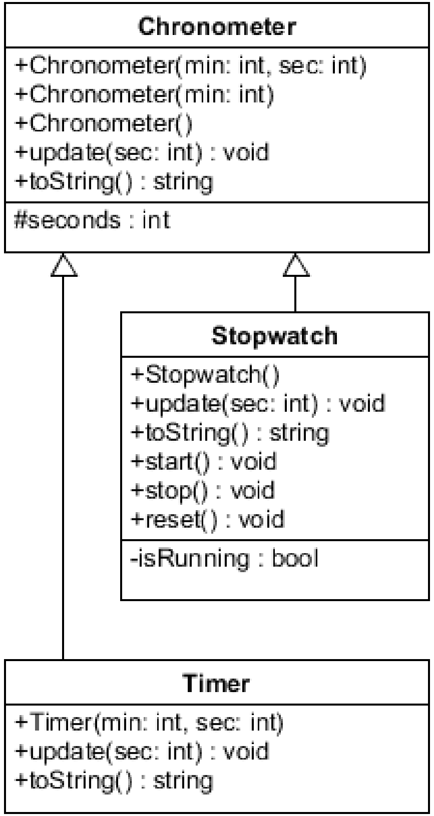
Typecasting

Motivation – re-useability

- The following code should be used for all chronometers (including derived classes) work.

```
for runChronometer(Chronometer& rChrono, int nSeconds)
{
      // a stopwatch should be here via
      // Stopwatch::start() can be started

      for (int i = 0; i < nSeconds; i++) {
          // Ok, here are only virtual functions.
          rChrono. update(1);
          sleep(1); // sleep one second
          std::cout<< rChrono. toString() << std::endl;
     }
     // ... and can be stopped with Stopwatch::stop()
}
```

The stopwatch is problematic here because it has to be started and stopped separately!



Typecasting
Static cast

- Static cast is like a C-style conversion with additional type checking for references and pointers:

```
target = static_cast<target type>( source );
```

```
class Base { };
class MyClassA : public Base { };
class MyClassB : public Base { };

int main() {

    // int* p1 = static_cast<int*>(new double(5)); // Compiler Error
    int* p2 = (int*) new double(5); // (erroneously) works with the C-style

    MyClassA a;
    // Up-cast (to the base class) is allowed
    Base* pBase = static_cast<Base*>(&a);
    // Down-cast (to derived classes) is allowed
    MyClassA& rA = static_cast<MyClassA&>(*pBase);
    MyClassA& rA2 = (MyClassA&)*pBase; // this (erroneously) also works
    // But  careful: no check if this is the correct derived class!
    // -> Using pB would lead to a program crash!
    MyClassB* pB = static_cast<MyClassB*>(pBase);
}
```

Typecasting
Dynamic cast

- Type checking at runtime using **`typeid`** or **`dynamic_cast`** is necessary to find out which derived class is really the base class of a pointer or reference

Prerequisite:

- RTTI (Run-Time Type Information) must be enabled in the compiler
- The base class must be virtual
  (i.e., contain at least one virtual function, e.g. a virtual destructor)

Typecasting
Dynamic cast - typeid

- With `typeid(object)` the actual type of an object can be determined
  This also works if the object is passed as a pointer or reference to the base class

```
typeid(TYPE) // returns a std::type_info class

// TYPE comparisons:
if ( typeid(TYPE) == typeid(TYPE) )
{
// ...
}

if ( typeid(TYPE) != typeid(TYPE) )
{
// ...
}

// returns the name of the type as C-string:
const char* name = typeid(TYPE). name();
```

Typecasting

Dynamic cast example – typeid


```
#include <iostream>
#include <typeinfo>

class Base { public: virtual ~Base() { } }; // Base class must be virtual!
class MyClassA : public Base { };


int main() {
    MyClassA a;
    Base& rBase = a;
    Base* pBase = &a;

    if (typeid(rBase) == typeid(MyClassA)) {
        std::cout << "rBase is a reference to a!" << std::endl;
        // because the type was checked, the conversion is now safe:
        MyClassA & rA = static_cast<MyClassA&>(rBase);
    }

    if (typeid(*pBase) == typeid(MyClassA)) {
        std::cout << "pBase is a pointer to a!" << std::endl;
        MyClassA * pA = static_cast<MyClassA*>(pBase);
    }
}
```

Typecasting

Dynamic cast – `dynamic_cast`

Syntax

```
target = dynamic_cast< target type>( source );
```

- For down-casts with runtime verification if the *target type* is actually a derived class of the *source type*
- If the target type is not a derived class of source, `dynamic_cast` ...
  - returns `nullptr` for pointers
  - throws a `std::bad_cast` exception for references

Typecasting

Dynamic cast example – dynamic_cast

```cpp
#include <iostream>
#include <typeinfo> // for std::bad_cast exception

class Base { public: virtual ~Base() { } }; // Base class must be virtual!
class MyClassA : public Base { };
class MyClassB : public Base { };

int main() {
    MyClassA a;
    Base& rBase = static_cast<Base&>(a);
    MyClassA& rA = dynamic_cast<MyClassA&>(rBase); // works
    try {
        MyClassB& rB = dynamic_cast<MyClassB&>(rBase);
    }
    catch (const std::bad_cast& e) {
        std::cout << "Downcast to rB failed!" << std::endl;
    }
    MyClassA* pA = dynamic_cast<MyClassA*>(&rBase);
    MyClassB* pB = dynamic_cast<MyClassB*>(&rBase);
    if (pB == nullptr) std::cout << "Downcast to pB failed!" << std::endl;
}
```

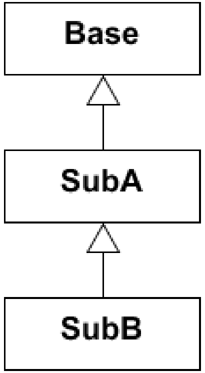
```
Downcast to rB failed!
Downcast to pB failed!
```

Typecasting

Dynamic cast over class hierarchies

Comparison between `typeid` and `dynamic_cast`

```
SubB b;
Base* pBase = & b; // ok, implicit conversion

// dynamic_cast also converts to the base class SubA:
SubA* pA = dynamic_cast< SubA*>(pBase); // -> works
SubB* pB = dynamic_cast< SubB*>(pBase); // -> works

// This is not possible with typeid!
typeid(*pBase) == typeid(SubA) // -> false!
typeid(*pBase) == typeid(SubB) // -> true!
```



- `dynamic_cast` is more flexible in the inheritance hierarchy, but slow
- `typeid` in combination with `static_cast` is faster, but inflexible
- `dynamic_cast` has become accepted as a better programming style

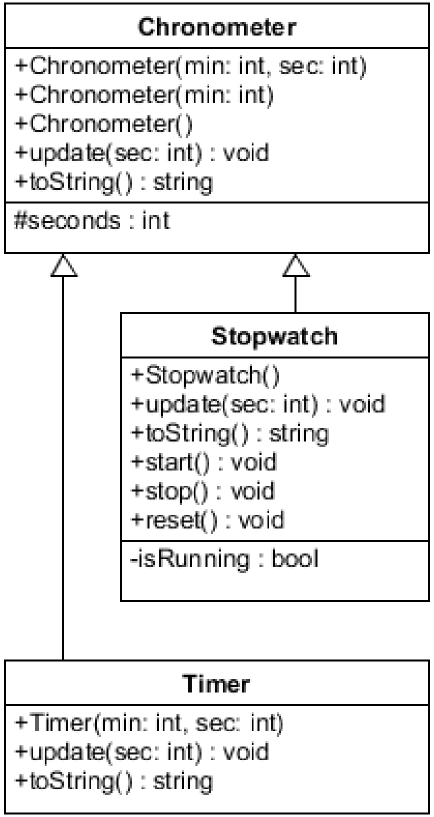
Typecasting

Dynamic cast – application

Downcasts

- With a dynamic cast the following code can be used for all chronometer classes:

```
void runChronometer(Chronometer& rChrono, int nSeconds)
{
    Stopwatch* pSW = dynamic_cast<Stopwatch*>(&rChrono);
    if (pSW != NULL) { pSW->start(); }

    for (int i = 0; i < nSeconds; i++) {
        rChrono.update(1);
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        std::cout << rChrono.toString() << std::endl;
    }

    if (pSW != NULL) { pSW->stop(); }
}
```



Typecasting

Const cast

Syntax

```
target = const_cast< target type>( source );
```

```
#include < iostream>

int main()
{
    int a = 10;
    const int& crA = a;
    // crA = 3; // not allowed - (rA is const)
    int& rA = const_cast<int&>(crA); // removes the const
    rA = 5; // works now
    std::cout << a << std::endl;

    const_cast<int&>(crA) = 3; // can also be done directly
    std::cout << a << std::endl;

    // be careful with 'real' constants!
    const char* pHello = "hello World!";
    char* p = const_cast<char*>(pHello);
    p[0] = 'H'; // non-handled exception!
    std::cout << pHello << std::endl;
}
```

Typecasting
Summary

- In C-style all types with brackets can be converted in C-style: (*target type*)
- This approach is largely unsafe

- C++ offers different safe variants for casting:
  - **Static cast** – for normal type conversion
    the compiler additionally checks if the conversion is possible.
  - **Dynamic cast** – to convert from pointers/references of a base class to a derived class
  - **Const cast** – to convert a constant type to a non-constant type (use with care!)
  - **Reinterpret cast** – to force a conversion and circumvent the compiler type check
    (don‘t use! – bad style and error prone)

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de