# Advanced Programming

## Part II: Objects in `C++` - Inheritance and Type Conversion

Alexander Breuer, Tobias Weinzierl

January 09, 2014

## Part II: Overview

**Theory** Inheritance as a Concept

- Understanding ”is-a”
- Interface and Implementation

**Basics in `C++`** How to use inheritance?

- Derived Class
- Virtual Member Functions
- Extended Access Control

**Vtables** Inheritance Under the Hood

- Realization of Virtual Functions
- Extending Vtables for Inheritance
- Discussion of the ”Performance Overhead”

**Type Conversion** Changing the Type of Variables

- Implicit
- Explicit: Old School
- Dynamic, Static, Reinterpret, Const

# Understanding ”is-a”

## Introduction

- ”is-a” relationship is equivalent to inheritance in `C++`
- Sofware design: Use inheritance only, if you can **proof ”is-a”** for the **current** status and **future** developments of your project
- Warning: Our intuotion of ”is-a” is often inexact and error-prone, when it comes to software design

## Students

- *Student*: study(), askSuperVisor()
- *TumStudent is-a student*: useParabolicSlides()
- $\Rightarrow$ Student.study() ✓, TumStudent.askSuperVisor() ✓, TumStudent.useParabolicSlides() ✓, Student.useParabolicSlides() ✗

## Understanding ”is-a”: Birds

### Birds

- *Bird*: fly()
- *Penguin is-a bird*
- ⇒ Bird.fly() ✓, Penguin.fly() ✓?

*What happenend?*

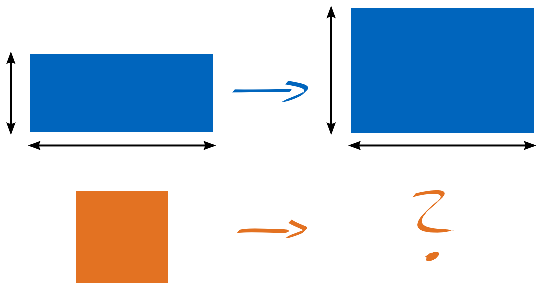
- Problem of our intuotion/language, ”birds fly” means ”in general birds have the ability to fly”
- Correct model, if our project contains no non-flying bird and will not contain non-flying birds in future

### Birds (2)

- *Bird*
- *FlyingBird is-a Bird*: fly()
- *NonFlyingBird is-a Bird*
- *Penguin is-as a NonFlyingBird*
- ⇒ FlyingBird.fly() ✓NonFlyingBird.fly() ✗, Peguin.fly() ✗

## Understanding ”is-a”: Rectangles

### Sketch

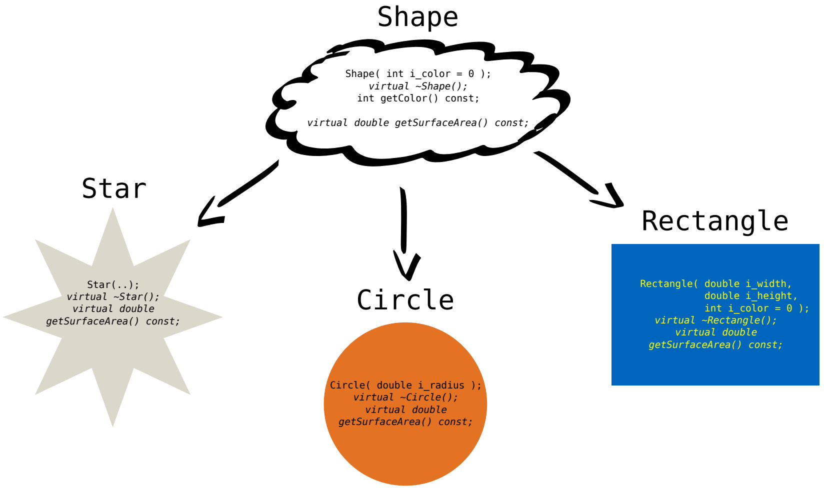


### Rectangles

- *Rectangle*: increaseSize()
- *Square is-a Rectangle*
- ⇒ Rectangle.increaseSize() ✓, Square.increaseSize()✓?

## Inheritance: Interface and Implementation

### Sketch



```
Shape( int i_color = 0 );
virtual ~Shape();
int getColor() const;

virtual double getSurfaceArea() const;
```

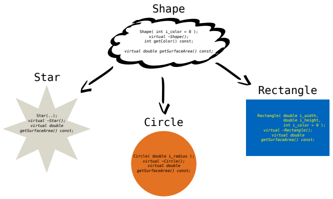
```
Star(..);
virtual ~Star();
virtual double
getSurfaceArea() const;
```

```
Circle( double i_radius );
virtual ~Circle();
virtual double
getSurfaceArea() const;
```

```
Rectangle( double i_width,
           double i_height,
           int i_color = 0 );
virtual ~Rectangle();
virtual double
getSurfaceArea() const;
```

## Inheritance: Interface and Implementation (2)

### Sketch



```
Shape( int i_color = 0 );
virtual ~Shape();
int getColor() const;

virtual double getSurfaceArea() const;
```

```
Star(..);
virtual ~Star();
virtual double
getSurfaceArea() const;
```

```
Circle( double i_radius );
virtual ~Circle();
virtual double
getSurfaceArea() const;
```

```
Rectangle( double i_width,
           double i_height,
           int i_color = 0 );
virtual ~Rectangle();
virtual double
getSurfaceArea() const;
```

### Description

- Inheritance splits into: Function **interfaces** and function **implementations**
- Allows for three different type implementations:
  - Implementation provided by base class
  - Default-implementation provided base class (can be overwritten)
  - Interface provided by base class (needs to be implemented)

## Part II: Overview

**Theory** Inheritance as a Concept

- Understanding ”is-a”
- Interface and Implementation

**Basics in** `C++` How to use inheritance?

- Derived Class
- Virtual Member Functions
- Extended Access Control

**Vtables** Inheritance Under the Hood

- Realization of Virtual Functions
- Extending Vtables for Inheritance
- Discussion of the ”Performance Overhead”

**Type Conversion** Changing the Type of Variables

- Implicit
- Explicit: Old School
- Dynamic, Static, Reinterpret, Const

## Inheritance in C++ : Basics

### Source

```
class Shape {
  //private:
  public:
    Shape() {
      std::cout << "constructed␣a␣new␣shape" << std::endl;
    }

    virtual ~Shape(){};
};

class Star:      public Shape {};
class Circle:    public Shape {};
class Rectangle: public Shape {};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/basics.cpp`

### Concept

*Shape* is the **base** class for the **derived** classes *Star*, *Circle* and *Rectangle*

## Inheritance in C++ : Basics (2)

### Code

```
class Rectangle: public Shape {};

int main() {
  Shape     l_shape1;
  Shape     l_shape2;
  Star      l_star1;
  Circle    l_circle;
  Rectangle l_rectangle;

}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/basics.cpp`

### Rules

A derived class inherits every member besides:

- Constructor and destructor of the base class (default constructors called per default)
- Assignment operator members
- friends (upcoming)

## Inheritance in C++ : Basics (3)

### Code

```cpp
class Rectangle: public Shape {};

  l_shape1 = l_shape2;
  l_shape2 = l_rectangle;
  Shape l_shape3( l_star1 );

  // forbidden
  //l_circle = l_shape2;
  //l_star1    = l_rectangle;
  //Star l_star2( l_shape1 );
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/basics.cpp

### Rules

A derived class inherits every member besides:

- Constructor and destructor of the base class (default constructors called per default)
- Assignment operator members
- friends (upcoming)

## Implementations and Interfaces

```cpp
class Shape {
  //private:
    int m_color;

  public:
    Shape( int i_color=0 ):
     m_color( i_color ) {};
    virtual ~Shape(){};

    int getColor() const { return m_color; };

    // purely virtual function
    virtual double getSurfaceArea() const = 0;

    // virtual function with default implementation
    //virtual double getSurfaceArea() const { return 0.0; };
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual.cpp`

### Keyword

**virtual** allows for interfaces and default implementations
**Non-virtual** functions are resolved **statically** at compile time, **virtual** functions **dynamically** at run time

## Implementations and Interfaces (2)

```cpp
class Circle: public Shape {
  //private:
    double m_radius;

  public:
    Circle( double i_radius ):
     Shape(), m_radius( i_radius ) {
      std::cout << "constructed␣a␣circle,␣radius:␣"
                << m_radius << std::endl;
    };
    virtual ~Circle(){};

    virtual double getSurfaceArea() const { return M_PI * ↙
         ↳  m_radius * m_radius; };


};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual.cpp`

### Description

Interfaces/**pure virtual** functions **have to** be implemented in derived classes; **virtual** interfaces **can** be overwritten; **non-virtual** usually **shouldn't**

## Implementations and Interfaces (3)

### Source

```
class Rectangle: public Shape {
  //private:
    double m_width, m_height;
  public:
    Rectangle( double i_width, double i_height, int i_color = 0 ):
     Shape( i_color),  m_width( i_width ), m_height( i_height) {
      std::cout << "constructed␣rectangle,␣width/height:␣"
                << m_width << "/" << m_height << std::endl;
    };
    virtual ~Rectangle{};

    virtual double getSurfaceArea() const { return m_width * m_height; };
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual.cpp`

### Description

Interfaces/**pure virtual** functions **have to** be implemented in derived classes; **virtual** interfaces **can** be overwritten; **non-virtual** usually **shouldn't**

## Implementations and Interfaces (4)

```
int main() {
  Circle l_circle( 2.3 );
  Rectangle l_rectangle( 4.2, 1.8, 3 );

  Shape* l_shapes[2];
  l_shapes[0] = &l_circle;
  l_shapes[1] = &l_rectangle;

  for( int l_shapeId = 0; l_shapeId < 2; l_shapeId ++ ) {
    std::cout << l_shapes[l_shapeId]->getColor() << ",␣"
              << l_shapes[l_shapeId]->getSurfaceArea() << std::endl;
  }

  // not allowed for virtual function w/o default implementation
  //Shape l_shape1();
  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual.cpp`

### Description

”Right” virtual functions are called (even if the compiler doesn’t have this knowledge) due to runtime decoding

*How is this working?* ⇒ vtables (upcoming)

## Static and Dynamic Decoding

```cpp
class Base {
  //private:
  public:
    void staticPrint() {
      std::cout << "base" << std::endl;
    }

    virtual void dynamicPrint() {
      std::cout << "base" << std::endl;
    }
};

class Derived: public Base {
  //private:
  public:
    void staticPrint() {
      std::cout << "derived" << std::endl;
    }

    virtual void dynamicPrint() {
      std::cout << "derived" << std::endl;
    }
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/static_dynamic.cpp`

## Static and Dynamic Decoding (2)

```
int main() {
  Base l_base;
  l_base.staticPrint();
  l_base.dynamicPrint();

  Derived l_derived;
  l_derived.staticPrint();
  l_derived.dynamicPrint();

  Base *l_derivedPointer = &l_derived;
  l_derivedPointer->staticPrint();
  l_derivedPointer->dynamicPrint();
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/static_dynamic.cpp`

### Description

”Right” virtual functions are called (even if the compiler doesn’t have this knowledge) due to runtime decoding
*How is this working?* ⇒ vtables (upcoming)

## Virtual Destructors

### Source

```
class Base {
  //private:
  public:
    virtual ~Base() { std::cout << "clean␣up␣after␣Base" << std::endl; }

    // usually bad practive: not virtual for a base class
    //~Base() { std::cout << "clean up after Base" << std::endl; }
};

class Derived: public Base {
  //private:
  public:
    virtual ~Derived() { std::cout << "clean␣up␣after␣Derived" << std::endl; }
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual_destructor.cpp`

## Virtual Destructors (2)

```cpp
    virtual ~Base() { std::cout << "clean␣up␣after␣Base" << std::endl; }

    virtual ~Derived() { std::cout << "clean␣up␣after␣Derived" << std::endl; }

int main() {
  Base l_base;
  Derived l_derived;

  Base    *l_pointerToBase     = new Base;
  Derived *l_pointerToDerived1 = new Derived;
  Base    *l_pointerToDerived2 = new Derived;

  delete l_pointerToBase;
  delete l_pointerToDerived1;
  delete l_pointerToDerived2; // might cause trouble
  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual_destructor.cpp`

### Concept

Virtual destructors are bound to the type the object at creation.

## Virtual Destructors (3)

```
    //~Base() { std::cout << "clean up after Base" << std::endl; }

    virtual ~Derived() { std::cout << "clean␣up␣after␣Derived" ↙
        ↳ << std::endl; }

int main() {
  Base l_base;
  Derived l_derived;

  Base    *l_pointerToBase      = new Base;
  Derived *l_pointerToDerived1 = new Derived;
  Base    *l_pointerToDerived2 = new Derived;

  delete l_pointerToBase;
  delete l_pointerToDerived1;
  delete l_pointerToDerived2; // might cause trouble
  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/virtual_destructor.cpp`

### Concept

Non-virtual destructors are bound to the type of the pointer, which points to them.

## Virtual Destructors: Summary

### In Detail

You need a **virtual destructor** for an object if:

- A class is **derived** from it
- **new** is used to construct objects of your class
- **delete** is called on the resulting pointers, which have the type of a base class

### Rule of Thumb

And much simpler: **Virtual member functions $\Rightarrow$ Virtual destructor**

## Inheritance: Motivation Revised

### Concept

- Usual implementation in C: New Code calls old code; example: *myNewFunction()* calls *printf()*
- Feature due to inheritance and `C++` : Old code calls new code; example *someOldFunction()* calls *myNewFunction()*

### Example

- Graphics library, which draws all objects you throw at it
- It will draw pre-defined objects **and** custom objects (added after the implementation of the library) ⇒ Hands on in the tutorials

## Part II: Overview

**Theory** Inheritance as a Concept

- Understanding ”is-a”
- Interface and Implementation

**Basics in `C++`** How to use inheritance?

- Derived Class
- Virtual Member Functions
- Extended Access Control

**Vtables** Inheritance Under the Hood

- Realization of Virtual Functions
- Extending Vtables for Inheritance
- Discussion of the ”Performance Overhead”

**Type Conversion** Changing the Type of Variables

- Implicit
- Explicit: Old School
- Dynamic, Static, Reinterpret, Const

## Extended Access Control

### Concept

- **protected** members can be accessed by derived classes
- Inheritance itself has access control
  - **Public inheritance** (most commonly used)

| Base | Derived Access | Public Access |
|---|---|---|
| public | ✓ | ✓ |
| protected | ✓ | ✗ |
| private | ✗ | ✗ |

  - **Protected inheritance**

| Base | Derived Access | Public Access |
|---|---|---|
| public | ✓ | ✗ |
| protected | ✓ | ✗ |
| private | ✗ | ✗ |

  - **Private inheritance**: Same as protected, but derived classes can't access anymore

## Extended Access Control: friend

```
class Simple {
  //private:
    int m_private;
  protected:
    int m_protected;
  public:
    int m_public;

     friend void  modify( Simple &io_simple );
     friend class Friend;
     friend class DerivedFriend;
};

void modify( Simple &io_simple ) {
  io_simple.m_private   = 1;
  io_simple.m_protected = 1;
  io_simple.m_public    = 1;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/friend.cpp`

### Keyword

**friend** allows functions or classes to access private and protected members

## Extended Access Control: friend (2)

### Code

```
class Simple {

   friend void  modify( Simple &io_simple );
   friend class Friend;
   friend class DerivedFriend;
};

class Friend {
  public:
    void modify( Simple &io_simple ) {
      io_simple.m_private   = 1;
      io_simple.m_protected = 1;
      io_simple.m_public    = 1;
    }
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/friend.cpp`

### Keyword

**friend** allows functions or classes to access private and protected members

## Extended Access Control: friend (3)

### Code

```
class Simple {

   friend void  modify( Simple &io_simple );
   friend class Friend;
   friend class DerivedFriend;
};

class DerivedFriend: public Simple {
  public:
    void modify() {
      m_private   = 1;
      m_protected = 1;
      m_public    = 1;
    }
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/friend.cpp

### Keyword

**friend** allows functions or classes to access private and protected members

## Part II: Overview

**Theory** Inheritance as a Concept

- Understanding ”is-a”
- Interface and Implementation

**Basics in `C++`** How to use inheritance?

- Derived Class
- Virtual Member Functions
- Extended Access Control

**Vtables** Inheritance Under the Hood

- Realization of Virtual Functions
- Extending Vtables for Inheritance
- Discussion of the ”Performance Overhead”

**Type Conversion** Changing the Type of Variables

- Implicit
- Explicit: Old School
- Dynamic, Static, Reinterpret, Const

## VTables: Inheritance Under the Hood

### Concept

- Implementation of inheritance and runtime decoding of virtual functions is **compiler specific**
- Most compilers use **virtual tables** (short: vtables) and corresponding **virtual pointers** (short: vptrs)
- Virtual table: Usually an array of **function pointers** (sometimes: linked list); One VTable for all objects of a class
- Virtual pointer: Pointer to the virtual table of this class; one pointer per object

### Layout

*Next*: VTable implementation for an example, *afterwards*: performance considerations and *in the tutorials*: low level examples.

## Implementation: Base Class

### Code

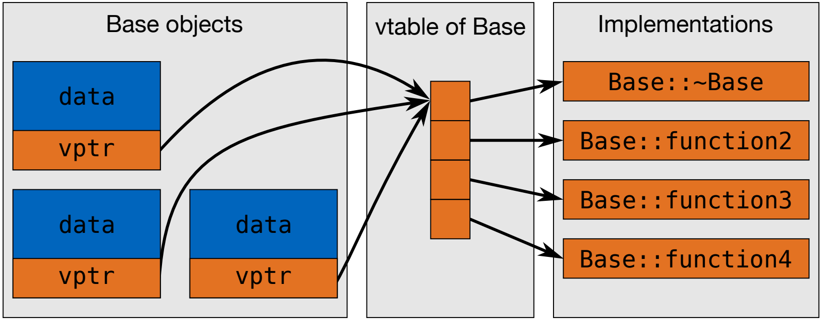
```
class Base {
  public:
    virtual       ~Base()  {}
    void          function1(){}
    virtual void  function2(){}
    virtual void  function3(){}
    virtual void  function4(){}
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/vtables.cpp`

### Summary

*Base* contains four virtual functions (destructor, function 2,3,4) ⇒ Have to be stored in a vtable

## Internal Realization: Base Class



### Concept

- Each *Base* object stores a vpointer to the vtable of class *Base*
- Vtable of *Base* contains function pointers to implementations of virtual functions: *˜Base, function2, function3* and *function4*
- vpointer not necessarily aligned with end/beginning of data

## Implementation: Derived Class

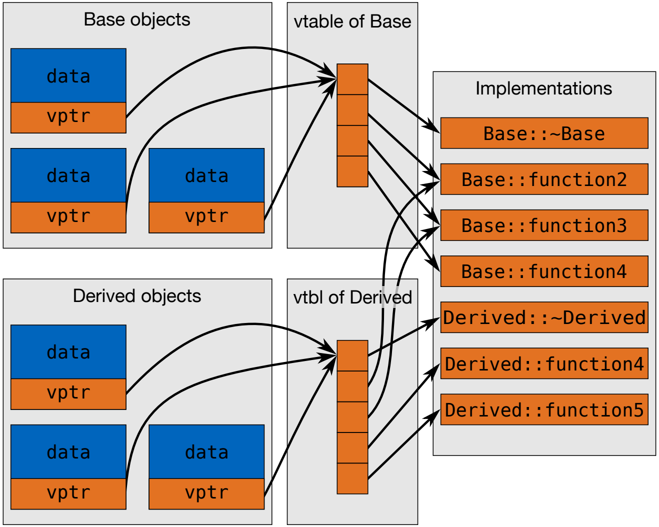
```
class Base {
  public:
    virtual       ~Base() {}
    void          function1(){}
    virtual void  function2(){}
    virtual void  function3(){}
    virtual void  function4(){}
};

class Derived: public Base {
  public:
    virtual       ~Derived() {}
    virtual void  function4(){}
    virtual void  function5(){}
    void          function6(){};
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/inheritance/vtables.cpp`

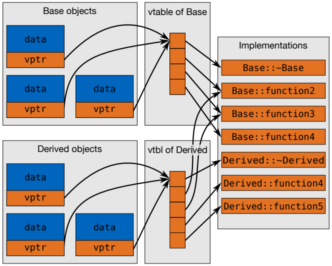
### Summary

*Derived*: Three virtual functions (*˜Derived*,*function 4&5*); *function4* reimplements *Base*s implementation, *˜Derived* the destructor and *function 5* is new

## Internal Realization: Derived Class

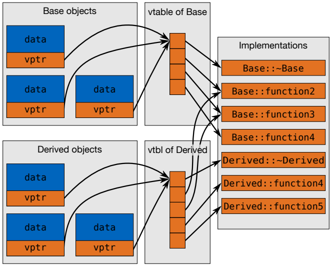


## Vtables: Step-by-step



- Create virtual table & add funtion pointers for virtual functions of *Base*, overwrite pointers for re-implementations in *Derived*
- Add new virtual functions of *Derived*
- Set up virtual pointer to the generated vtable if an object of type *Derived* is generated

## Vtables: Summary



- Vpointers together with vtables ensure that the right functions are called – even if base pointers are used
- Overloading a function is just a different address (compared to base class) stored in the corresponding vtable
- **fetch-fetch-call**-approach instead of the **fetch-call** of non-virtual functions

## Performance: Location of Vtables

- Build process of a program (compile & link) must provide a **single vtable** for all objects of a class across the program
- Two major approaches to solve this challenge:
  - Brute-force: Generate a vtable every object, which could need it. Get rid of copies during linking. ⇒ Reasonable, if compiler == linker
  - Heuristics: Usually place vtable in the first object containing the first non-inline non-pure virtual function, for us: Files containing *˜Base* and *˜Derived*; problematic with lots of inlines

## Performance: Overhead

### Additional Structure

- **One vtable per class** (not per object of the type); contains a function pointer for every virtual function this class can call
  Critical for a large number of classes in the program; Bad software design?
- **One vpointer per object** containing virtual function
  Crititcal for small objects; AoS vs. SoA?
- Fetch-fetch-call: **Follow two pointers instead of one**
  Crtical for virtual functions with minor workloads

### In General

Vtables come with a certain overhead but bring functionality.
Low-level implementations come with an overhead of their own (i.e. if-else statements) to support same functionality ⇒ Use inheritance/vtables if reasonable for your software design

## Part II: Overview

**Theory** Inheritance as a Concept

- Understanding ”is-a”
- Interface and Implementation

**Basics in** `C++` How to use inheritance?

- Derived Class
- Virtual Member Functions
- Extended Access Control

**Vtables** Inheritance Under the Hood

- Realization of Virtual Functions
- Extending Vtables for Inheritance
- Discussion of the ”Performance Overhead”

**Type Conversion** Changing the Type of Variables

- Implicit
- Explicit: Old School
- Dynamic, Static, Reinterpret, Const

## Implicit Type Conversion in C

### Code

```
int main(){
  unsigned int l_uint1 = -1;
  unsigned int l_uint2 = -2;
  unsigned int l_uint3 = -1 + 1;
  int l_int1           = true;
  int l_bool1          = -15;
  double* l_pointer    = false;
  bool l_bool2         = l_pointer;
  int l_int2           = 3.9;
  float l_float1       = 5 / 2;
  float l_float2       = l_float1 * 0.5; // where's the implicit typecast?

};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/implicit.cpp`

### Task

What is the result of the operations above? You have five minutes..

## Implicit Type Conversion: Rules

An expression *e* of a given type is **implicitly converted** if used in one of the following situations:

- Expression *e* is used as an operand of an **arithmetic** or **logical** operation.
- Expression *e* is used as a condition in an **if** statement or an **iteration** statement (such as a for loop). Expression e will be converted to bool (or int in C).
- Expression e is used in a **switch** statement. Expression e will be converted to an integral type.
- ...

[...]

IBM, XL C/C++ (V6.0)

## Implicit Type Conversion: Rules (2)

- Expression e is used in an **initialization**. This includes the following:
    - An **assignment** is made to an lvalue that has a different type than *e*.
    - A **function** is provided an **argument** value of *e* that has a different type than the parameter.
    - Expression *e* is specified in the **return** statement of a function, and *e* has a different type from the defined return type for the function.

The compiler will allow an implicit conversion of an expression e to a type T **if and only if** the compiler would allow the following statement:
*T var = e;*

...

You can perform **explicit** type conversions using one of the **cast operators**, the **function style cast**, or the **C-style cast**.

IBM, XL C/C++ (V6.0)

# C-like and Functional Type Casting

## Code

```cpp
int main(){
  double l_double = 3.9;

  int l_int1 = (int) l_double;  // C-like
  int l_int2 = int  (l_double); // functional

  std::cout << l_double << std::endl;
  std::cout << l_int1   << std::endl;
  std::cout << l_int2   << std::endl;

  return 0;
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/clike_functional.cpp

## Concept

**C-like/functional** type casting is extremely **powerful** (no restrictions), but also **error-prone**: You are circumventing type checks at compile time

## Explicit Type Conversion in C++

### Example Classes

```
class Base {
  //private:
  public:
    virtual ~Base(){};
};

class Derived: public Base {
  //private:
  public:
    virtual ~Derived(){};
};
```

### Concept

C++ adds four less error-prone type casts to your toolbox:

- `dynamic_cast<T>(e)`
- `static_cast<T>(e)`
- `reinterpret_cast<T>(e)`
- `const_cast<T>(e)`

, `e` is the expression and `T` the new type. Important features: *Now*

## Dynamic Cast

```
int main() {
  Base*    l_pointerToDerived1 = new Derived;
  Base*    l_pointerToBase     = new Base;
  // not allowed: implicit down cast
  //Derived* l_pointerToDerived2 = l_pointerToDerived1;
  //Derived* l_pointerToDerived3 = l_pointerToBase;

  Derived* l_pointerToDerived4 = dynamic_cast<Derived*>(  ↙
        ↳  l_pointerToDerived1 );
  Derived* l_pointerToDerived5 = dynamic_cast<Derived*>(  ↙
        ↳  l_pointerToBase     );

}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/dynamic.cpp`

### Description

Works on **references** and **pointers**; Always allows for **upcast** (derived to base), **downcasts** only (base to derived), if save, else: NULL.
*Warning:* **Performance overhead** due to **Run-time type information (RTTI)**.

## Dynamic Cast (2)

### Code

```
  if( l_pointerToDerived4 ) std::cout << "first␣cast␣works" << std::endl;
  else                      std::cout << "first␣cast␣returns␣NULL" << std::endl;

  if( l_pointerToDerived5 ) std::cout << "second␣cast␣works" << std::endl;
  else                      std::cout << "second␣cast␣returns␣NULL" << std::endl;

  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/dynamic.cpp`

### Description

Works on **references** and **pointers**; Always allows for **upcast** (derived to base), **downcasts** only (base to derived), if save, else: NULL.
*Warning:* **Performance overhead** due to **Run-Time Type Information (RTTI)**.

## Static Cast

### Code

```
int main() {
  Base    *l_pointer1 = new Base;
  Derived *l_pointer2 = static_cast<Derived*>(l_pointer1);
  // not allowed for static casts
  //int     *l_pointer3 = static_cast<int*>(l_pointer2);

  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/static.cpp`

### Description

Allows **pointers of related classes** to be converted into each other (up- & downcasts) and all operations allowed via **implicit conversions**

*Warning:* No checks done (NULL result) as in dynamic ⇒ Fast but error-prone

# Reinterpret Cast

## Code

```cpp
int main() {
  Base    *l_pointer1 = new Base;
  Derived *l_pointer2 = reinterpret_cast<Derived*>(l_pointer1);
  int     *l_pointer3 = reinterpret_cast<int*>(l_pointer2);

  std::cout << reinterpret_cast<long>(l_pointer1) << std::endl;
  std::cout << reinterpret_cast<long>(l_pointer2) << std::endl;
  std::cout << reinterpret_cast<long>(l_pointer3) << std::endl;

  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/reinterpret.cpp`

## Description

Allows **any pointer** to be converted **to any other pointer**, **pointer to integral type** and **integral type to pointer**

*Warning:* Low level binary copies are involved ⇒ Fast but error-prone

## Const Cast

### Code

```
void print( int &i ) {
  std::cout << i << std::endl;
}

int main() {
  const int l_int = 27;

  // not allowed: print(..) is allowed to modify l_int
  //print( l_int );

  print( const_cast<int&>(l_int) );

  return 0;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/type_conversion/const.cpp`

### Description

Overwrites the constant status of a variables

## References and Literature

- *Scott Meyers*, Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition
- *Scott Meyers*, More Effective C++: 35 New Ways to Improve Your Programs and Designs
- `http://www.parashift.com/c++-faq/`
- `http://www.cplusplus.com`