Basic C++

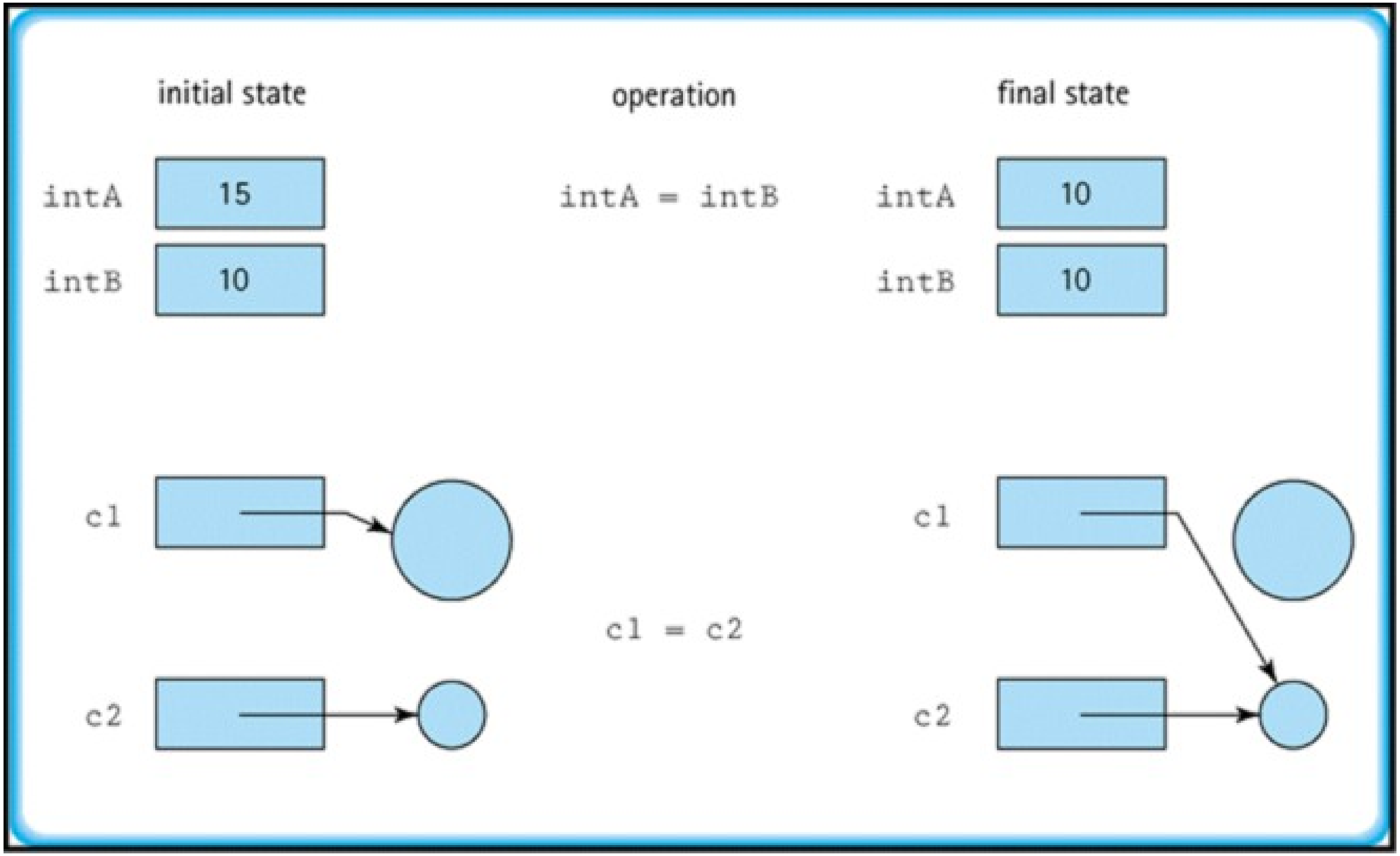
9

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Value vs Reference semantics



Calendar class

```
#include <date.h>
#include <string>

struct Event    // one entry in Calendar
{
    Date        time_;
    std::string descr_;
    Event      *next_ = nullptr;
};
class Calendar    // the Calendar itself
{
public:
    void push_back( Event ptr);       // insert event at the end
    void pop_front();                 // remove oldest event
    Event& find( std::string descr);  // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};
```

Calendar class

```
#include <date.h>
#include <string>

struct Event    // one entry in Calendar
{
    Date        time_;
    std::string descr_;
    Event      *next_ = nullptr;
};
class Calendar    // the Calendar itself
{
public:
    void push_back( Event ptr);      // insert event at the end
    void pop_front();                // remove oldest event
    Event& find( std::string descr); // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
}
```

Important remark!

```cpp
#include <date.h>
#include <string>

struct Event   // one entry in Calendar
{
    Date        time_;
    std::string descr_;
    Event      *next_ = nullptr;
};
class Calendar   // the Calendar itself
{
public:
    void push_back( Event ptr);       // insert event at the end
    void pop_front();                 // remove oldest event
    Event& find( std::string descr);  // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
}
```

Important remark!

```
#include <date.h>
#include <string>

struct Event    // one entry in Calendar
{
    Date        time_;
    std::string descr_;
    Event      *next_ = nullptr;
};
class Calendar   // the Calendar itself
{
public:
    void push_back( Event ptr);        // insert event at the end
    void pop_front();                  // remove oldest event
    Event& find( std::string descr);   // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
}
```

Important remark!

```
#include <date.h>
#include <string>

struct Event    // one entry in Calendar
{
    Date        time_;
    std::string descr_;
};
using Calendar = std::list<Event>;
using Calendar = std::list<std::pair<Date,std::string>>;

using Calendar = std::map<Date,std::string>;
using Calendar = std::multimap<Date,std::string>;

int main()
{
    Calendar c;
    c.push_back( { {2023,8,14}, "training" };     // list
    c[{2023,8,14}] = "training";                   // map
    c.insert({{2023,8,14},"training"});            // multimap
}
```

Calendar class

```
#include <date.h>
#include <string>

struct Event;   // one entry in Calendar

class Calendar   // the Calendar itself
{
public:
    void push_back( Event ptr);       // insert event at the end
    void pop_front();                 // remove oldest event
    Event& find( std::string descr);  // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
    c.push_back( { {2023,8,19}, "travel home" };

    Calendar c2 = c;   // create a copy of c
}
```

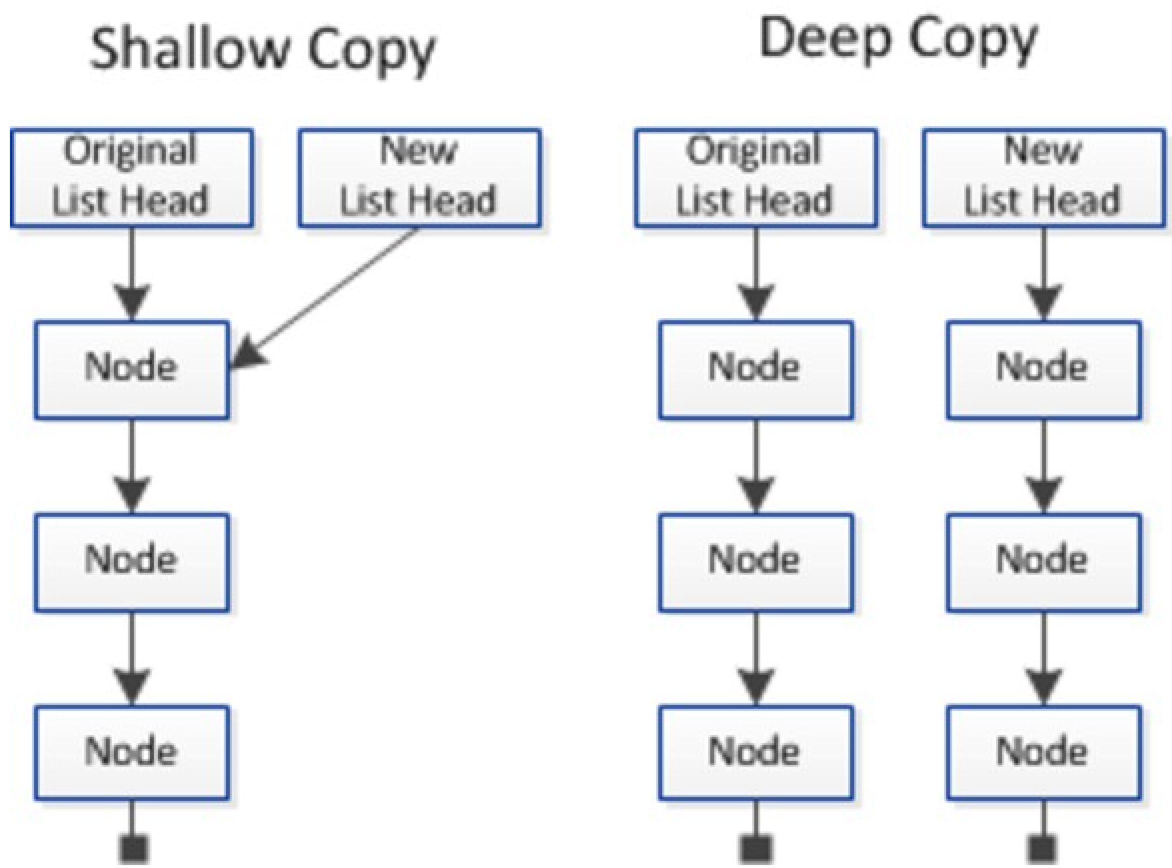
Calendar class

```
#include <date.h>
#include <string>

struct Event;   // one entry in Calendar

class Calendar   // the Calendar itself
{
public:
    void push_back( Event ptr);        // insert event at the end
    void pop_front();                  // remove oldest event
    Event& find( std::string descr);   // get Event connected to description
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
    c.push_back( { {2023,8,19}, "travel home" };

    Calendar c2 = c;   // create a copy of c will copy first and last
}
```

Calendar class

```
#include <date.h>
#include <string>

struct Event;   // one entry in Calendar

class Calendar   // the Calendar itself
{
public:
    void push_back( Event ptr);       // insert event at the end
    void pop_front();                 // remove oldest event
    Event& find( std::string descr);  // get Event connected to description
private:
    Event  *first_ = nullptr;
    Event  *last_  = nullptr;
};

int main()
{
    Calendar c; // empty
    c.push_back( { {2023,8,14}, "training" };
    c.push_back( { {2023,8,19}, "travel home" };

    Calendar c2 = c;   // create a copy of c will copy first and last
    c2.pop_front();    // first element of c will also disappair
}
```

Value vs Reference semantics



Non-Trivially copyable types

- Some classes cannot be handled as “char array”s
- Initializing, copying and destruction requires user defined actions
  - Constructor(s)
  - Copy constructor
  - Assignment operator
  - Destructor
- Do not copy them as char array (e.g., with memcpy, memmove)
- Destructor must never throw exception!

Rule of 3

```
#include <date.h>
#include <string>

class Calendar   // the Calendar itself
{
public:
    Calendar() { }                   // default constructor
                                     //   to initialize object
    Calendar(const Calendar& rhs);   // copy constructor
                                     //   to initialize from object of same type
    Calendar& operator=(const Calendar& rhs); // assignment operator
                                     //   to copy between objects of same type
    ~Calendar();                     // destructor
                                     //   to clean up after end of life


};

int main()
{
    Calendar c1;        // calls default constructor
    Calendar c2 = c1;   // c2.Calendar(c1) = c1   copy constructor
    c1 = c2;            // c1.operator=(c2)  assignment
}                       // c2.~Calendar(), c1.~Calendar() destructors called
```

Rule of ~~3~~ 5

```
#include <date.h>
#include <string>

class Calendar   // the Calendar itself
{
public:
    Calendar() { }                     // default constructor
                                       //   to initialize object
    Calendar(const Calendar& rhs);     // copy constructor
                                       //   to initialize from object of same type
    Calendar& operator=(const Calendar& rhs); // assignment operator
                                       //   to copy between objects of same type
    ~Calendar();                       // destructor

    Calendar(Calendar&& rhs);                // move constructor
    Calendar& operator=(Calendar&& rhs);     // move operator
};

int main()
{
    Calendar c1;        // calls default constructor
    Calendar c2 = c1;   // c2.Calendar(c1) = c1   copy constructor
    c1 = c2;            // c1.operator=(c2)  assignment
}                       // c2.~Calendar(), c1.~Calendar() destructors are called
```

Implementing special operations

```
struct Event    // one entry in Calendar
{
    Date        time_;
    std::string descr_;
    Event       *next_ = nullptr;
};
class Calendar    // the Calendar itself
{
Public:
    Calendar(const Calendar& rhs);
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

Calendar::Calendar(const Calendar& rhs)
{
    // allocate resources and copy from rhs
    Event *p = rhs.first_;
    while ( p )
    {
        push_back({p->time_,p->descr_});
        p = p->next_;
    }
}
```

Implementing special operations

```
class Calendar   // the Calendar itself
{
Public:
    Calendar& operator=(const Calendar& rhs);
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

Calendar& Calendar::operator=(const Calendar& rhs)
{
    // remove my old resources
    Event *p = first_;
    while ( p )
    {
        // ??  we have to allocate the Event objects on the head dynamically
        p = p->next_;
    }
    // allocate resources and copy from rhs
    *p = rhs.first_;
    while ( p )
    {
        push_back({p->time_,p->descr_});
        p = p->next_;
    }
}
```

Implementing special operations

```
class Calendar   // the Calendar itself
{
Public:
    Calendar& operator=(const Calendar& rhs);
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

Calendar& Calendar::operator=(const Calendar& rhs)
{
    // remove my old resources
    Event *p = first_;
    while ( p )
    {
        delete p;       // delete the Event object dynamically
        p = p->next_;  // oops, p points to deleted memory
    }
    // allocate resources and copy from rhs
    *p = rhs.first_;
    while ( p )
    {
        push_back( new Event{p->time_,p->descr_});
        p = p->next_;
    }
}
```

Implementing special operations

```
class Calendar   // the Calendar itself
{
Public:
    Calendar& operator=(const Calendar& rhs);
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

Calendar& Calendar::operator=(const Calendar& rhs)
{
    // remove my old resources
    Event *p = first_;
    while ( p )
    {
        Event *toDelete = p;
        p = p->next_;       // fine p points to next
        delete toDelete;    // delete the Event object dynamically
    }
    // allocate resources and copy from rhs
    *p = rhs.first_;
    while ( p )
    {
        push_back(new Event{p->time_,p->descr_});
        p = p->next_;
    }
    return *this;
}
```

Implementing special operations

```
class Calendar   // the Calendar itself
{
Public:
    ~Calendar();
private:
    Event   *first_ = nullptr;
    Event   *last_  = nullptr;
};

Calendar::~Calendar()
{
    // remove my old resources
    Event *p = first_;
    while ( p )
    {
        Event *toDelete = p;
        p = p->next_;      // fine p points to next
        delete toDelete;   // delete the Event object dynamically
    }
}
```

Implementing special operations

```
void Calendar::release()  // private
{
    // remove my old resources
    Event *p = first_;
    while ( p )
    {
        Event *toDelete = p;
        p = p->next_;      // fine p points to next
        delete toDelete;   // delete the Event object dynamically
    }
}
void Calendar::copy(const Calendar& rhs) // private
{
    *p = rhs.first_;
    while ( p )
    {
        push_back(new Event{p->time_,p->descr_});
        p = p->next_;
    }
}
Calendar::Calendar(const Calendar& rhs) { copy(rhs); }
Calendar::~Calendar() { release(); }
Calendar& operator=(const Calendar& rhs) {copy(rhs); release(); return *this;}
```

Implementing special operations

```
Calendar& operator=(const Calendar& rhs)
{



    release(); // release old resources
    copy(rhs); // allocate and copy new resources

    return *this;
}


int main()
{
    Calendar c1; // put data to c1
    Calendar c2; // put data to c2


    c1 = c2;    // ok op= first delete source (c1) then copy elements from c2

}
```

Implementing special operations

```
Calendar& operator=(const Calendar& rhs)
{



    release(); // release old resources
    copy(rhs); // allocate and copy new resources

    return *this;
}


int main()
{
    Calendar c1; // put data to c1
    Calendar c2; // put data to c2

    Calendar *p = &c1;
    c1 = c2;    // ok op= first delete source (c1) then copy elements from c2
    c1 = *p;    // oops, op= first delete source (c1) then try to copy to c1
}
```

Implementing special operations

```
Calendar& operator=(const Calendar& rhs)
{
    if ( this == &rhs )
        return *this;      // early return to avoid c = c

    release(); // release old resources
    copy(rhs); // allocate and copy new resources

    return *this;
}


int main()
{
    Calendar c1; // put data to c1
    Calendar c2; // put data to c2

    Calendar *p = &c1;
    c1 = c2;    // ok op= first delete source (c1) then copy elements from c2
    c1 = *p;    // ok, early returns avoid to delete source (c1)
}
```

Implementing special operations

```
#include <string>
#include "calendar.h"
#include "computer.h"

struct OfficeTools
{
    std::string    name_plate_;
    Calendar       cal_;
    Computer       comp_;
}


int main()
{
    OfficeTools    m1;          // should we write constructor?
    OfficeTools    m2 = m1;     // should we write copy constructor?

    m1 = m2;                    // should we write operator=?
}                               // should we write destructor?
```

Implementing special operations

```
#include <string>
#include "calendar.h"
#include "computer.h"

struct OfficeTools
{
    std::string    name_plate_;
    Calendar       cal_;
    Computer       comp_;
}


int main()
{
    OfficeTools    m1;          // should we write constructor?
    OfficeTools    m2 = m1;     // should we write copy constructor?

    m1 = m2;                    // should we write operator=?
}                               // should we write destructor?
```

- No! member-wise operations apply automatically
- But we can, if we want to change the behavior.

Copy operations

- Aggregate and trivially copyable types are copied member-wise
- Non-trivially copyable types require
  - Constructor(s)
  - Copy constructor
  - Assignment operator
  - Destructor
  - Do not copy them as char array (e.g., with memcpy, memmove)
- We can forbid copy operations at all if we want

Deleted and default operations

```
class X
{
private:
    X(const X&);              // pre-C++11 way to delete copy operatons
    X& operator=(const X&); // pre-C++11
};

class X : private boost::noncopyable // pre-C++11 way to delete copy operatons
{
    // ...
};

class X
{
    // copy is not to use
    X(const X&) = delete;              // C++11 way to delete copy operatons
    X& operator=(const X&) = delete; // C++11
};

class X
{
    // memberwise copy is required
    X(const X&) = default;      // C++11 to generate defalt copy operations
    X& operator=(const X&) = default; // C++11
};
```

Delegated constructors C++11

```
// C++98
class X
{
   int a;
   validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
   X(int x) { validate(x); }
   X() { validate(42); }
   X(string s) { int x = boost::lexical_cast<int>(s); validate(x); }
   // ...
};

// C++11
class X
{
   int a;
public:
   X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
   X() : X{42} { }
   X(string s) : X{boost::lexical_cast<int>(s)} { }
   // ...
};
```

Inheritance

- Suppose, we want to create a software system to inspect vehicles
- There is a mandatory (bi-)annual test for each vehicle
- But there are different parameters/requirements for different vehicles
  - Personal Cars
  - Buses
  - Trucks
- We might want to extend the system later by new vehicle types

Inheritance: first approach

```
class Owner;   // representing owner, either person or legal entity

class Vehicle
{
public:
    Vehicle( std::string pl, ...);

    bool mot() const;               // annual test of the vehicles

    std::string plate() const { return plate_; } // usual getters/setters
    // Further Vehicle interface ...



private:
    std::string _plate;  // licence plate
    // Further Vehicle attributes ...
};
```

Inheritance: first approach

```
class Owner;   // representing owner, either person or legal entity

class Vehicle
{
public:
    Vehicle( std::string pl, int ax, ...);

    bool mot() const;              // annual test of the vehicles

    std::string plate() const { return plate_; } // usual getters/setters

    double wpax() const { return weight_/axes_; }    // only for trucks
    int passangers() const { return passangers_; }   // only for buses

private:
    std::string plate_;  // licence plate for all
    // Further Vehicle attributes ...

    double  weight_;        // only for trucks
    int     axes_;          // only for trucks
    int     passangers_;    // only for buses
};
```

Inheritance: first approach

```
class Owner;   // representing owner, either person or legal entity

class Vehicle
{
public:
    Vehicle( std::string pl, int ax, ...);    // Contructor

    bool mot() const;              // annual test of the vehicles

    std::string plate() const { return plate_; } // usual getters/setters

    double wpax() const { return weight_/axes_; }    // only for trucks
    int passangers() const { return passangers_; }   // only for buses

private:
    std::string _plate;  // licence plate for all
    enum tag { CAR, TRUCK, BUS} tag_;
    union
    {
        Car     c;
        Truck   t;
        Bus     b;
    };
};
```

Inheritance: first approach

```
bool Vehicle::mot() const      // annual test of the vehicles
{
    switch( tag_ )
    {
    case CAR:   // car specific...
                break;
    case TRUCK: // truc specific...
                break;
    case BUS:   // bus specific...
                break;
    default:    // is this possible? is this valid?
                break;
    };
};
```

- How to extend this kind of code?

Inheritance: second approach

```
class Owner;   // representing owner, either person or legal entity

class Car
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor

    bool mot() const;              // annual test of the vehicles

    std::string plate() const { return plate_; } // usual getters/setters
    double emission() const { return emission_;} // usual getters/setters



private:
    Vehicle    base_;     // common attribute
    double     emission_; // car specific attributes
};
```

Inheritance: second approach

```
class Owner;   // representing owner, either person or legal entity

class Car
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor

    bool mot() const;              // annual test of the vehicles

    std::string plate() const { return plate_; } // no plate in Car
    double emission() const { return emission_;} // usual getters/setters



private:
    Vehicle    base_;     // common attribute
    double     emission_; // car specific attributes
};
```

Inheritance: second approach

```
class Owner;   // representing owner, either person or legal entity

class Car
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor

    bool mot() const;            // annual test of the vehicles

    std::string plate() const { return base_.plate(); } // re-write interface
    double emission() const { return emission_;} // usual getters/setters



private:
    Vehicle    base_;      // common attribute
    double     emission_; // car specific attributes
};
```

Inheritance: second approach

```
bool mot(? v)  // what kind of parameter?
{
    switch( tag_ )     // somehow we should know the type of Vehicle
    {
    case CAR:   return c.mot();    // car specific...
                break;
    case TRUCK: return t.mot();    // truc specific...
                break;
    case BUS:   return b.mot();    // bus specific...
                break;
    default:    // is this possible? is this valid?
                break;
    };
};
```

- How to extend this kind of code?

Inheritance: second approach

```
bool mot(std::variant<Car,Truck,Bus> v)  // what kind of parameter?
{
    switch( tag_ )     // somehow we should know the type of Vehicle
    {
    case CAR:   return v.get<Car>().mot();    // car specific...
                break;
    case TRUCK: return v.get<Truck>().mot();  // truc specific...
                break;
    case BUS:   return v.get<Bus>().mot();    // bus specific...
                break;
    default:    // is this possible? is this valid?
                break;
    };
};
```

- How to extend this kind of code?

Inheritance

```
class Vehicle
{
public:
    Vehicle( std::string pl, ...);

    bool mot() const;             // annual test of the vehicles
    std::string plate() const { return plate_; } // only common getters/setters
    std::string to_string() const { return "[Vehicle: "+plate_+"]"; }
private:
    std::string plate_;  // only common data members
};

class Car : public Vehicle
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor
    bool mot() const;             // annual test for Cars
    double emission() const { return emission_;} // Car specific
    std::string to_string() const {return "[Car: "+plate_+", "+emission_+"]";}
private:
    double     emission_; // car specific attributes
};

class Truck : public Vehicle { ... }; // similar
class Bus   : public Vehicle { ... }; // similar
```

Inheritance

- Inheritance extends base class
- Derived class holds all attributes of base class
- (public) Derived class interface contains base interface
  - Can overwrite with different behavior
  - Extend with new methods
- Derived class cannot access Base private members
  - But they access protected members (transitive relationship)
- Class hierarchy can be incrementally extended

Inheritance

```
int main()
{
    Car     c{"ABC-123", 0.6};     // licence plate, emission
    Truck   t{"EFG-123", 4500, 8}; // licence plate, weight, axes
    Bus     b{"HIJ-123", 50};      // licence plate, passangers

    st::string l1 = c.plate();     // ok, inherited from Vehicle
    double     em = c.emission(); // ok, from Car
    st::string l2 = c.plate();     // ok, inherited from Vehicle
    double     wp = c.wpax();      // ok, from Truck
    st::string l3 = c.plate();     // ok, inherited from Vehicle
    int        ps = c.passangers(); // ok, from Bus
}
```

Inheritance

- Derived classes can access Base public members
- Derived class cannot access Base private members
- Derived classes can access Base protected members (transitively)
- Public inheritance
  - Public members of Base is part of Derived interface
- Protected inheritance
  - Public members of Base act as Derived protected members
- Private inheritance
  - Public members of Base act as Derived private members
  - This is only technically inheritance

Constructors

```
class Vehicle
{
public:
    Vehicle( std::string pl, ...);

    bool mot() const;           // annual test of the vehicles
    std::string plate() const { return plate_; } // only common getters/setters
    std::string to_string() const { return "[Vehicle: "+plate_+"]"; }
private:
    std::string plate_;  // only common data members
};

class Car : public Vehicle
{
public:
    Car( std::string pl, double emission, Owner own) :
        Vehicle( pl, own),     // step 1: pass parameters to base class
        emission_(emission)    // step 2: initialize own attributes
    {
        // step 3: execute Car specific actions
    }
    // ...
};
```

Conversions: upcast

```
int main()
{
    Car     c{"ABC-123", 0.6};     // licence plate, emission
    Truck   t{"EFG-123", 4500, 8}; // licence plate, weight, axes
    Bus     b{"HIJ-123", 50};      // licence plate, passangers

    Vehicle   v = c;   // ok, Car is a Vehicle, but slicing happens
    Vehicle *vp = &c;  // ok, Car is a Vehicle, no slicing
    Vehicle &vr = c;   // ok, Car is a Vehicle, no slicing

    std::string lp = v.plate();        // ok, Vehicle has plate
    int         em = v.emission();     // error, Vehicle has no emission

    std::string lp = vp->plate();      // ok, Vehicle has plate
    int         em = vp->emission();   // error, Vehicle has no emission

    std::string lp = vr.plate();       // ok, Vehicle has plate
    int         em = vr.emission();    // error, Vehicle has no emission

    // similar for Truck and Bus
```

Conversions: downcast

```
int main()
{
    Car     c{"ABC-123", 0.6};     // licence plate, emission
    Truck   t{"EFG-123", 4500, 8}; // licence plate, weight, axes
    Bus     b{"HIJ-123", 50};      // licence plate, passangers

    Vehicle   v = c;   // ok, Car is a Vehicle, but slicing happens
    Vehicle *vp = &c;  // ok, Car is a Vehicle, no slicing
    Vehicle &vr = c;   // ok, Car is a Vehicle, no slicing

    std::string lp = v.plate();        // ok, Vehicle has plate
    int         em = static_cast<Car>(v).emission(); // error, where get data?

    std::string lp = vp->plate();      // ok, Vehicle has plate
    int         em = static_cast<Car>(vp)->emission(); // ok, vp pointed a Car

    std::string lp = vr.plate();       // ok, Vehicle has plate
    int         em = static_cast<Car>(vr).emission(); // ok, vr referred a Car

    // similar for Truck and Bus
}
```

Conversions: downcast

```
int main()
{
    Car     c{"ABC-123", 0.6};      // licence plate, emission
    Truck   t{"EFG-123", 4500, 8}; // licence plate, weight, axes
    Bus     b{"HIJ-123", 50};       // licence plate, passangers

    Vehicle   v = c;   // ok, Car is a Vehicle, but slicing happens
    Vehicle *vp = &c;  // ok, Car is a Vehicle, no slicing
    Vehicle &vr = c;   // ok, Car is a Vehicle, no slicing

    std::string lp = v.plate();        // ok, Vehicle has plate
    int         em = static_cast<Car>(v).emission(); // error, where get data?

    std::string lp = vp->plate();      // ok, Vehicle has plate
    int         em = static_cast<Car>(vp)->emission(); // ok, vp pointed a Car

    std::string lp = vr.plate();       // ok, Vehicle has plate
    int         em = static_cast<Car>(vr).emission(); // ok, vr referred a Car

    // But are you sure that vp and vr really points/referres to a Car?
}
```

Static vs dynamic type

```
#include <vector>


void f()
{
    std::vector<Vehicle> vehicles;

    vl.push_back(Car{"ABC-123", 0.6});
    vl.push_back(Truck{"EFG-123", 4500, 8});
    vl.push_back(Bus{"HIJ-123", 50});


    for ( auto v : vehicles )
    {
        std::cout << v.to_string() << '\n';
    }
}

$ ./a.out
[Vehicle: "ABC-123"]
[Vehicle: "EFG-123"]
```

Static vs dynamic type

```
#include <vector>


void f()
{
    std::vector<Vehicle> vehicles;

    vl.push_back(Car{"ABC-123", 0.6});          // slicing!
    vl.push_back(Truck{"EFG-123", 4500, 8}); // slicing!
    vl.push_back(Bus{"HIJ-123", 50});           // slicing!


    for ( auto v : vehicles )
    {
        std::cout << v.to_string() << '\n';
    }
}

$ ./a.out
[Vehicle: "ABC-123"]
[Vehicle: "EFG-123"]
```

Static vs dynamic type

```
#include <vector>


void f()
{
    std::vector<Vehicle*> vehicles;

    vl.push_back(new Car{"ABC-123", 0.6});        // fine
    vl.push_back(new Truck{"EFG-123", 4500, 8}); // fine
    vl.push_back(new Bus{"HIJ-123", 50});         // fine


    for ( auto vp : vehicles )
    {
        std::cout << vp->to_string() << '\n';
    }
}
```

Static vs dynamic type

```
#include <vector>

void f()
{
    std::vector<Vehicle*> vehicles;

    vl.push_back(new Car{"ABC-123", 0.6});       // fine
    vl.push_back(new Truck{"EFG-123", 4500, 8}); // fine
    vl.push_back(new Bus{"HIJ-123", 50});        // fine


    for ( auto vp : vehicles )
    {
        std::cout << vp->to_string() << '\n';
    }
} // memory leak!
```

Static vs dynamic type

```
#include <vector>
#include <memory>

void f()
{
    std::vector<Vehicle*> vehicles;

    vl.push_back(std::make_unique<Car>("ABC-123", 0.6);          // fine
    vl.push_back(std::make_unique<Truck>("EFG-123", 4500, 8);   // fine
    vl.push_back(std::make_unique<bus>("HIJ-123", 50);           // fine


    for ( auto vp : vehicles )
    {
        std::cout << vp->to_string() << '\n';
    }
} // ok, delete is called for all pointers in vehicles
```

Static vs dynamic type

```
#include <vector>
#include <memory>

void f()
{
    std::vector<Vehicle*> vehicles;   // ponters of Vehicle*

    vl.push_back(std::make_unique<Car>("ABC-123", 0.6);          // fine
    vl.push_back(std::make_unique<Truck>("EFG-123", 4500, 8);   // fine
    vl.push_back(std::make_unique<bus>("HIJ-123", 50);          // fine


    for ( auto vp : vehicles )        // static type of *vp is Vehicle
    {                                 // dynamic type of *vp are Car,Truck,Bus
        std::cout << vp->to_string() << '\n';  // called on static type
    }
}

$ ./a.out
[Vehicle: "ABC-123"]
[Vehicle: "EFG-123"]
[Vehicle: "HIJ-123"]
```

Virtual functions

- Non-virtual functions are called on static type
- Virtual functions are called on the actual dynamic type
- Virtual functions should be declared in the Base class
- Overriding versions should declare in the Derived class(es)
  - with exactly same signature (incl. noexcept, const, volatile)
- Classes with at least one virtual functions are called Polymorphic
- There is an overhead for polymorphic classes
  - Memory (vptr)
  - Run-time (call of the function)
- Pure virtual to mark a class abstract

Polymorphism

```
class Vehicle
{
public:
    Vehicle( std::string pl, ...);

    virtual bool mot() const = 0;           // annual test of the vehicles
    std::string plate() const { return plate_; } // only common getters/setters
    virtual std::string to_string() const = 0 {return "[Vehicle: "+plate_+"]";}
private:
    std::string plate_;  // only common data members
};

class Car : public Vehicle
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor
    bool mot() const override;        // annual test for Cars
    double emission() const { return emission_;} // Car specific
    virtual std::string to_string() const override {
                          return "[Car: "+plate_+", "+emission_+"]";}
private:
    double     emission_; // car specific attributes
};
```

Polymorphism

```
#include <vector>
#include <memory>

void f()
{
    std::vector<Vehicle*> vehicles;   // ponters of Vehicle*

    vl.push_back(std::make_unique<Car>("ABC-123", 0.6);          // fine
    vl.push_back(std::make_unique<Truck>("EFG-123", 4500, 8);   // fine
    vl.push_back(std::make_unique<bus>("HIJ-123", 50);           // fine


    for ( auto vp : vehicles )        // static type of *vp is Vehicle
    {                                 // dynamic type of *vp are Car,Truck,Bus
        std::cout << vp->to_string() << '\n'; // virtual,called on dynamic type
    }
}

$ ./a.out
[Car: "ABC-123", 0.6]
[Truck: "EFG-123", 562.5 ]
[Bus: "HIJ-123", 50]
```

Polymorphism

```
class Vehicle   // abstract base class, one cannot create Vehicle objects
{
public:
    Vehicle( std::string pl, ...);
    virtual ~Vehicle() { }    // to prepare be destroyed via base pointer
    virtual bool mot() const = 0;      // annual test of the vehicles
    std::string plate() const { return plate_; } // only common getters/setters
    virtual std::string to_string() const = 0 {return "[Vehicle: "+plate_+"]";}
private:
    std::string plate_;  // only common data members
};

class Car : public Vehicle
{
public:
    Car( std::string pl, double emission, Owner own);    // Contructor
    bool mot() const override;        // annual test for Cars
    double emission() const { return emission_;} // Car specific
    virtual std::string to_string() const override {
                                  return "[Car: "+plate_+", "+emission_+"]";}
private:
    double     emission_; // car specific attributes
};
```

Cloning – “Virtual” constructors

- Constructors are not virtual
- But sometimes we need “virtual” behavior

```
std::vector<Base*> source;
std::vector<Base*> target;

source.push_back( new Derived1() );
source.push_back( new Derived2() );
source.push_back( new Derived3() );

// should create new instances of the
// corresponding Derived classes and
// place them to target
deep_copy( target, source );
```

Wrong approach

```
void deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source )
{
  for( auto i = source.begin(); i != source.end(); ++i )
  {
    target.push_back( new Base(**i) ); // creates Base
  }
}
```

Wrong approach 2

```
void deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source )
{
  for( auto i = source.begin(); i != source.end(); ++i )
  {
    if ( Derived1 *dp = dynamic_cast<Derived1*>(*i) )
      target.push_back( new Derived1(*dp) );
    else if ( Derived2 *dp = dynamic_cast<Derived2*>(*i) )
      target.push_back( new Derived2(*dp) );
    else if ( Derived3 *dp = dynamic_cast<Derived3*>(*i) )
      target.push_back( new Derived3(*dp) );
  }
}
```

Cloning

```
class Base
{
public:
  virtual Base* clone() const = 0;
};

class Derived : public Base
{
public:
  virtual Derived* clone() const { return new Derived(*this); }
};

deep_copy(std::vector<Base*> &target, const std::vector<Base*> &source)
{
  for( auto i = source.begin(); i != source.end(); ++i )
  {
    target.push_back( (*i)->clone() ); // inserts Derived()
  }
}
```