Basic C++

6

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

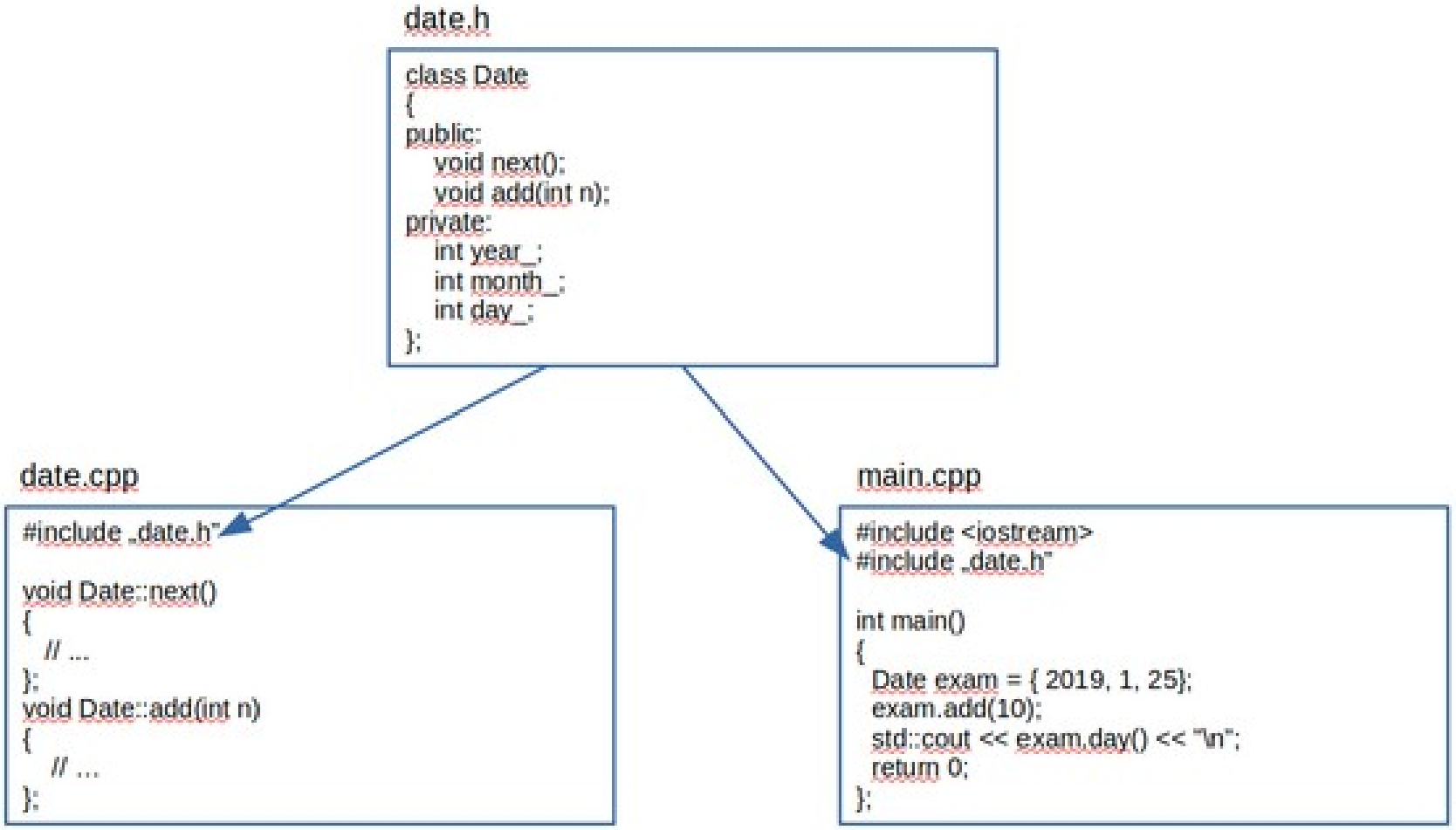
Trivially copyable types

- Scalar types
- Simple classes
- Can be mapped to a consecutive byte sequence
  - Can be copied by memcopy
  - Except living volatile objects
- Separation of interface and implementation

Date class

```
struct date_t   // aggregate
{
    int year_;
    int month_;
    int day_;
};

void f()
{
    date_t training_beg = { 2023, 8, 14};          // aggregate initialization
    date_t training_end = training_beg.day_ += 5; // { 2023, 8, 19}

    training_beg = training_end;  // assignment works by copying bytes
    training_beg = { 2023, 9, 1}; // assignment works by copying bytes

    date_t training2_beg = { 2023, 8, 30};
    date_t training2_end = training_beg2.day_ += 5;  // { 2023, 8, 35}

    date_t invalid1 = { 2023, 13, 1};  // no such month
    date_t invalid2 = { 2023, 12, 35}; // no such day
    date_t invalid3 = { 2023, 2, 30};  // month and day exist, still invalid
 }
```

Implementing C++ classes



Date class

```
#ifndef DATE_H
#define DATE_H

struct date_t   // aggregate
{
    int year_;
    int month_;
    int day_;
};

#endif // DATE_H


#include "date.h"
void f()
{
    date_t training_beg = { 2023, 8, 14};          // aggregate initialization
    date_t training_end = training_beg.day_ += 5; // { 2023, 8, 19}

    training_beg = training_end;  // assignment works by copying bytes
    training_beg = { 2023, 9, 1}; // assignment works by copying bytes

    date_t training2_beg = { 2023, 8, 30};
    date_t training2_end = training_beg2.day_ += 5;  // { 2023, 8, 35}

    date_t invalid1 = { 2023, 13, 1};  // no such month
    date_t invalid2 = { 2023, 12, 35}; // no such day
    date_t invalid3 = { 2023, 2, 30};  // month and day exist, still invalid
}
```

Date class

```
struct date_t   // aggregate
{
    int year_;
    int month_;
    int day_;

    void next();       // increment by one day
    void add(int n);   // add n days
};
void date_t::next()
{
    ++year_;
    // handle overflow ...
}
void date_t::add(int n)
{
    for (int i = 0; i < n; ++i)
        next(); // optimize later...
}
void f()
{
    date_t training_beg = { 2023, 8, 14};  // aggregate initialization
    training_beg.add(30);      // training delayed, still correct date

}
```

Date class

```
struct date_t   // aggregate
{
    int year_;
    int month_;
    int day_;

    void next();       // increment by one day
    void add(int n);   // add n days
};
void date_t::next()
{
    ++year_;
    // handle overflow ...
}
void date_t::add(int n)
{
    for (int i = 0; i < n; ++i)
        next(); // optimize later...
}
void f()
{
    date_t training_beg = { 2023, 8, 14};  // aggregate initialization
    training_beg.add(30);      // training delayed, still correct date
    training_beg.day_ += 40;   // still possible
}
```

Date class

```
struct date_t   // aggregate
{
public:  // public interface
    void next();       // increment by one day
    void add(int n);   // add n days
private: // hidden implementation
    int year_;
    int month_;
    int day_;
};
void date_t::next()
{
    ++year_;
    // handle overflow ...
}
void date_t::add(int n)
{
    for (int i = 0; i < n; ++i)
        next(); // optimize later...
}
void f()
{
    date_t training_beg = { 2023, 8, 14};  // aggregate initialization
    training_beg.add(30);      // training delayed, still correct date

}
```

Date class

```
struct date_t   // aggregate
{
public:  // public interface
    void next();       // increment by one day
    void add(int n);   // add n days
private: // hidden implementation
    int year_;
    int month_;
    int day_;
};
void date_t::next()
{
    ++year_;
    // handle overflow ...
}
void date_t::add(int n)
{
    for (int i = 0; i < n; ++i)
        next(); // optimize later...
}
void f()
{
    date_t training_beg = { 2023, 8, 14};  // aggregate initialization
    training_beg.add(30);      // training delayed, still correct date
    training_beg.day_ += 40;   // compile error, day is private
}
```

Date class

```
struct date_t   // aggregate
{
public:  // public interface
    void next();       // increment by one day
    void add(int n);   // add n days
private: // hidden implementation
    int year_;
    int month_;
    int day_;
};
void date_t::next()
{
    ++year_;
    // handle overflow ...
}
void date_t::add(int n)
{
    for (int i = 0; i < n; ++i)
        next(); // optimize later...
}
void f()
{
    date_t training_beg = { 2023, 8, 14};  // compile error, fields are private
    training_beg.add(30);      // training delayed, still correct date
    training_beg.day_ += 40;   // compile error, day is private
}
```

User-defined types

```
class Date
{
public:
    // constructor to initialize the object
    Date( int year, int month, int day) { year_= ; month_=m; day_=d; }
    // …
    int getYear()      { return year_;  }  // getters
    int getMonth()     { return month_; }
    int getDay()       { return day_;   }
    void setYear(int y)    { year_ = y;  } // setters
    void setMonth(int m)   { month_ = m; }
    void setSay(int d)     { day_ = d;   }
    // …
private:
    int year_;
    int month_;
    int day_;
};
```

User-defined types

```
class Date
{
public:
    // constructor to initialize the object
    Date( int year, int month, int day) { year_= ; month_=m; day_=d; }
    // …
    int getYear()      { return year_;  } // this->year
    int getMonth()     { return month_; } // this->month
    int getDay()       { return day_;   } // this->day
    void setYear(int y)    { year_ = y;  } // this->year
    void setMonth(int m)   { month_ = m; } // this->month
    void setSay(int d)     { day_ = d;   } // this->day
    // …
private:
    int year_;
    int month_;
    int day_;
};
```

User-defined types

```
class Date
{
public:
    // constructor to initialize the object
    Date( int year, int month, int day) { set( year, month, day); }
    // …
    int getYear()      { return year_;  }  // getters
    int getMonth()     { return month_; }
    int getDay()       { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields

    // …
private:
    int year_;
    int month_;
    int day_;
};

void Date::set(int y, int m, int d)   { ... }
```

User-defined types

```
class Date
{
public:
    // constructor to initialize the object
    Date( int year, int month, int day) { set( year, month, day); }
    // …
    int getYear()      { return year_;  }  // getters
    int getMonth()     { return month_; }
    int getDay()       { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields

    // …
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void Date::set(int y, int m, int d)   { ... }
void Date::check(int y, int m, int d) { ... }
```

User-defined types

```
class Date
{
public:
    // constructor to initialize the object
    Date( int year, int month, int day) { set( year, month, day); }
    // …
    int getYear()      { return year_;  }  // getters
    int getMonth()     { return month_; }
    int getDay()       { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void Date::set(int y, int m, int d) { check(y,m,d); year_=y; ... }
void Date::check(int y, int m, int d)
{
    if ( ... )
        throw std::out_of_range{};
}
```

User-defined types

```
class Date
{
public:
    // constructor initializer list
    Date( int year,int month,int day) : year_(y),month_(m),day_(d) { }
    // …
    int getYear()      { return year_;  }  // getters
    int getMonth()     { return month_; }
    int getDay()       { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void Date::set(int y, int m, int d) { check(y,m,d); year_=y; ... }
void Date::check(int y, int m, int d)
{
    if ( ... )
        throw std::out_of_range{};
}
```

User-defined types

```
class Date
{
public:
    // constructor with default parameters and initializer list
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    // …
    int getYear()       { return year_;  }  // getters
    int getMonth()      { return month_; }
    int getDay()        { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void Date::set(int y, int m, int d) { check(y,m,d); year_=y; ... }
void Date::check(int y, int m, int d)
{
    if ( ... )
        throw std::out_of_range{};
}
```

User-defined types

```
class Date
{
public:
    // constructor with default parameters and initializer list
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    // …
    int getYear()       { return year_;  }  // getters
    int getMonth()      { return month_; }
    int getDay()        { return day_;   }

    void set(int y, int m, int d); // check the parameters and set fields
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void f()
{
    Date d1{2023, 8, 17};
    Date d2{2023, 8};  // {2023, 8, 1}
    Date d3{2023};     // {2023, 1, 1}
}
```

Constness on user-defined types

```
class Date
{
public:
    Date( int year, int month=1, int day=1);
    // …
    int getYear();
    int getMonth();
    int getDay();
    void set(int y, int m, int d);
    // …
private:
    int year;
    int month;
    int day;
    void check(int y, int m, int d); // check the parameters
};

const Date my_birthday(1963,11,11); // const can be initialized
      Date curr_date(2015,7,10);

  curr_date = my_birthday; // ok, const can be read
my_birthday = curr_date;   // compile-time error: write const
```

Constness on user-defined types

- ```
  class Date
  {
  public:
      Date( int year, int month=1, int day=1);
      // …
      int getYear();
      int getMonth();
      int getDay();
      void set(int y, int m, int d);
      // …
  private:
      int year;
      int month;
      int day;
      void check(int y, int m, int d); // check the parameters
  };

  const Date my_birthday(1963,11,11);
        Date curr_date(2015,7,10);

  int x = my_birthday.getYear(); // can I read ??
  my_birthday.set(2015,7,10);    // can I write ??
  ```
  ```

Constness on user-defined types

- ```
  class Date
  {
  public:
      Date( int year, int month=1, int day=1);
      // …
      int getYear() const;
      int getMonth() const;
      int getDay() const;
      void set(int y, int m, int d);
      // …
  private:
      int year;
      int month;
      int day;
     void check(int y, int m, int d) const; // check the parameters
  };

  const Date my_birthday(1963,11,11);
        Date curr_date(2015,7,10);

  int x = my_birthday.getYear(); // works
  my_birthday.set(2015,7,10);    // compile error
  ```
  ```

Constness on user-defined types

- ```
  class Date
  {
  public:
      Date( int year, int month=1, int day=1);
      // …
      int getYear() const;
      int getMonth() const;
      int getDay() const;
      void set(int y, int m, int d);
      // …
  private:
      int year;
      int month;
      int day;
     void check(int y, int m, int d) const; // declared as const
  };

  const Date my_birthday(1963,11,11);
        Date curr_date(2015,7,10);

  void Date::check(int y, int m, int d) const // const is part of signature
  {
      if ( ... )
          throw std::out_of_range{};
  }
  ```
  ```

Constness on user-defined types

```
class Date
{
public:
  Date( int year, int month, int day);
  // …
  int getYear() const;          -> _ZNK4Date7getYearEv(const Date* this);
  int getMonth() const;
  int getDay() const;
  void set(int y, int m, int d);-> _ZN4Date3setEiii(Date *this,int y,int m,int d)
  // …
private:
  int year;
  int month;
  int day;
};
const Date my_birthday(1963,11,11);
      Date curr_date(2015,7,10);


int x = my_birthday.getYear(); // ok
my_birthday.set(2015,7,10);    // compile-time error!
```

Overloading on const

```
template <typename T, ... >
class vector
{
public:
    T&       operator[](size_t i);
    const T& operator[](size_t i) const;
    // ...
};
int main()
{
          std::vector<int>   iv;
    const std::vector<int>  civ;
    // ...
    iv[i] = 42;         // non-const
    int i = iv[5];
    int j = civ[5]      // const
    // ...
}
```

Const members

```
class Msg
{
public:
    Msg(const char *t);
    int getId() const { return id; }
private:
    const int id;
    std::string txt;
};

Msg m1("first"), m2("second");
m1.getId() != m2.getId();

MSg::Msg(const char *t)
{
    txt = t;
    id  = getNewId();  // syntax error, id is const
}

//initialization list works
MSg::Msg(const char *t) : id(getNextId()),txt(t) { }
```

Const members

```
class Msg
{
public:
    Msg(const char *t);
    int getId() const { return id; }
private:
    const int id = getNewId(); // since C++11
    std::string txt;
};
```

Mutable members

```
struct Point
{
    void getXY(int& x, int& y) const;
    double xcoord;
    double ycoord;
    mutable int read_cnt;
};

void f()
{
  const Point a;
  ++a.read_cnt;  // ok, Point::read_cnt is mutable
}

void Point::getXY(int& x, int& y) const
{
  // …
  ++read_cnt; // ok, Point::read_cnt is mutable
};
```

Mutexes are usually mutables

```
#include <mutex>

struct Point
{
public:
  void getXY(int& x, int& y) const;
  // …
private:
  double  xcoord;
  double  ycoord;
  mutable std::mutex m;
};

void getXY(int& x, int& y) const // atomic read of point
{
  std::lock_guard< std::mutex > guard(m); // locking of m
  x = xcoord;
  y = ycoord;
} // unlocking m
```

User-defined types

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date next();        // increment
    Date add(int n);    // add n days
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void f()
{
    Date d1{2023, 8, 17};
    Date d2 = d1.next();       // d2 == ?
    Date d3 = d2.add(40);      // d3 == ?
}
```

Clean interface

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date next();        // increment
    Date add(int n);    // add n days
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void f()
{
    Date d1{2023, 8, 17};
    Date d2 = d1.next();       // d2 == ? d1 == ?
    Date d3 = d2.add(40);      // d3 == ? d1 == ?
}
```

Clean interface

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date next();        // increment
    Date add(int n);    // add n days
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters
};

void f()
{
    Date d1{2023, 8, 17};
    Date d2 = ++d1;         // d2 == ? d1 == ?
    d2 += 40;               // d2 == ? d1 == ?
}
```

Clean interface

```
#include <iostream>    // standard header files
#include "date.h"      // date.h for date class

int main()
{
    date d1{2016,4};   // this should be 2016.04.01
    date d2 = d1;      // copy constructor, d2 is 2016.04.01 too

    d2 += 40;          // add 40 days
    d1 = d2;           // assignment, now d1 is 2016.05.11

    std::cout << "d1++ == " << d1++ << '\n'; // 2016.05.11
    std::cout << " d1  == " <<  d1  << '\n'; // 2016.05.12
    std::cout << "++d2 == " << ++d2 << '\n'; // 2016.05.12
    std::cout << " d2  == " <<  d2  << '\n'; // 2016.05.12

    if ( d1 < d2  &&  d3 != d1 )
        d3.set(2016,3,1);

    std::cout << "++d3 == " << ++d3 << '\n'; // 2016.03.02

    return 0;
}
```

How to define operators?

- a + b, a – b, a == b, ...

  a.**operator**+(b)

  operator+(a,b)

- a = b, a[b], a(b1,b2,...), a-> only member

  a.**operator**= (b)

  a.**operator**[ ] (b)

  a.**operator**( ) (b1, b2, …)

  a.**operator**->()

Clean interface

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }

    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields

    Date& operator++() { next(); return *this }      // pre-increment
    Date  operator++(int) { Date old{*this}; add(n); return old; }  // post
    Date& operator+=(int n) { add(n); return *this } // incr. assignment
private:
    int year_;
    int month_;
    int day_;

    void check(int y, int m, int d); // check the parameters, may throw
    void next();   // increment by 1
    void add(n);   // increment by n
};
```

Where to define operators?

- Theory says: data and operations on it have strong binding
  - Member operators
- Some operators can't be members

```
std::ostream& operator<<(std::ostream&,const X&)
```

- Sometimes members creates unwanted dependencies

```
std::getline(std::basic_istream&,std::basic_string&)
```

- Sometime operators should be symmetric

Clean interface

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date& operator++() { next(); return *this }       // pre-increment
    Date  operator++(int) { Date old{*this}; add(n); return old; }  // post
    Date& operator+=(int n) { add(n); return *this } // incr. assignment
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters, may throw
    void next();   // increment by 1
    void add(n);   // increment by n
};
std::ostream& operator<<(std::ostream& os, const date& d)
{
    os <<"["<<d.getYear()<<"."<<d.getMonth()<<"."<<d.getDay()<<"]";
    return os;
}
std::istream& operator>>(std::istream& is, date& d) { ... } // similar
```

Symmetry

```
class Date
{
public:
    Date( int year,int month=1,int day=1);
    // ...
    bool operator<(const Date& rhs) const;
    // ...
};

int main()
{
    Date today{2023,8,16};   // current date


    if ( today < date{2016} )  // works
       // ...
    if ( today < 2016 )        // works?
       // ...


};
```

Symmetry

```
class Date
{
public:
    Date( int year,int month=1,int day=1);
    // ...
    bool operator<(const Date& rhs) const;
    // ...
};

int main()
{
    Date today{2023,8,16};   // current date


    if ( today < date{2016} )  // works
       // ...
    if ( today < 2016 )        // works! today.operator<(2016)
       // ...


};
```

Symmetry

```
class Date
{
public:
    Date( int year,int month=1,int day=1);
    // ...
    bool operator<(const Date& rhs) const;
    // ...
};

int main()
{
    Date today{2023,8,16};   // current date


    if ( today < date{2016} )  // works
       // ...
    if ( today < 2016 )        // works! today.operator<(2016)
       // ...
    if ( 2016 < today )        // compile error! 2016.operator<(today)
       // ...
};
```

Symmetry

- ```
  class Date
  {
  public:
      Date( int year,int month=1,int day=1);
      // ...
  };

  bool operator<(const Date& lhs, const Date& rhs);

  int main()
  {
      Date today{2023,8,16};   // current date


      if ( today < date{2016} )  // works
         // ...
      if ( today < 2016 )        // works! operator<(today,2016)
         // ...
      if ( 2016 < today )        // works! Operator<(2016,today)
         // ...
  };
  ```
  ```

Explicit constructor

```
class Date
{
public:
    explicit Date( int year,int month=1,int day=1);
    // ...
};

bool operator<(const Date& lhs, const Date& rhs);

int main()
{
    Date today{2023,8,16};   // current date


    if ( today < date{2016} )  // works, explicit call of constructor
       // ...
    if ( today < 2016 )        // compile error! No implicit conversion
       // ...
    if ( 2016 < today )        // compile error! No implic
```

Clean interface

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date& operator++() { next(); return *this }      // pre-increment
    Date  operator++(int) { Date old{*this}; add(n); return old; }  // post
    Date& operator+=(int n) { add(n); return *this } // incr. assignment
private:
    // ...
};
std::ostream& operator<<(std::ostream& os, const Date& d);
std::istream& operator>>(std::istream& is, Date& d);

bool operator<(const Date& l, const Date& r);

inline bool operator>(const Date& l, const Date& r)  {return r<l;}
inline bool operator==(const Date& l, const Date& r) {return !(r<l||l<r);}
inline bool operator!=(const Date& l, const Date& r) {return !(l==r);}
// ...
inline int operator-(const Date& l, const Date& r) {return ...;}
```

Member pointers

```
class Date
{
public:
    Date( int year,int month=1,int day=1) : year_(y),month_(m),day_(d) { }
    int getYear() const    { return year_;  }
    int getMonth() const   { return month_; }
    int getDay() const     { return day_;   }
    void set(int y, int m, int d); // check the parameters and set fields
    Date& operator++() { next(); return *this }      // pre-increment
    Date  operator++(int) { Date old{*this}; add(n); return old; }  // post
    Date& operator+=(int n) { add(n); return *this } // incr. assignment
private:
    int year_;
    int month_;
    int day_;
    void check(int y, int m, int d); // check the parameters, may throw
    void next();   // increment by 1
    void add(n);   // increment by n
};
std::ostream& operator<<(std::ostream& os, const date& d) // [2023.8.16]
{
    os <<"["<<d.getYear()<<"."<<d.getMonth()<<"."<<d.getDay()<<"]";
    return os;
}
std::istream& operator>>(std::istream& is, date& d) { ... } // similar
```

Member pointers

- Data Member pointer: Referencing to an offset inside a class
- Member function pointer: Referencing to a (possible virtual) member function of a class
- Works with 2 components: **this + mptr**

```
Type Class::*dmptr;
Type (Class::*fmptr)(P1 par1, P2 par2, …);

Class obj;
Class *ptr = &obj;

obj.*dmptr = …;
ptr->*dmptr = …;

(obj.*fmptr)(par1,par2);
(ptr->*fmptr)(par1,par2);
```

Data member pointers

```
#include <iostream>

class Date
{
public:
  void set (int y, int m, int d);
  int  getYear() const  { return _year; }
  int  getMonth() const { return _month; }
  int  getDay() const   { return _day; }

  void hu();
  void us();
private:
  int year_;
  int month_;
  int day_;

  int Date::*p1 = year_;
  int Date::*p2 = month_;
  int Date::*p3 = day_;

  char sep = '.';
};
```

Data member pointers

```
void Date::hu()
{
  sep = '.';
  p1 = &Date::_year;
  p2 = &Date::_month;
  p3 = &Date::_day;
}

void Date::us()
{
  sep = '/';
  p1 = &Date::_month;
  p2 = &Date::_day;
  p3 = &Date::_year;
}
```

```
int main()
{
  Date d;
  d.set(2017,4,20);
  d.hu();
  std::cout << d << std::endl;
  d.us();
  std::cout << d << std::endl;
}
```

```
2017.4.20
4/20/2017
```

```
std::ostream& operator<<( std::ostream& os, const Date& d)
{
  os << this->*p1 << sep << this->*p2 << sep << this->*p3;
  return os;
}
```

Member function pointers

```
#include <iostream>

class Date
{
public:
  void set (int y, int m, int d);
  int  getYear() const  { return _year; }
  int  getMonth() const { return _month; }
  int  getDay() const   { return _day; }

  void hu();
  void us();
private:
  int year_;
  int month_;
  int day_;

  int (Date::*g1)() const = &Date::getYear;
  int (Date::*g2)() const = &Date::getMonth;
  int (Date::*g3)() const = &Date::getDay;

  char sep = '.';
};
```

Member function pointers

```
void Date::hu()
{
  sep = '.';
  g1 = &Date::getYear;
  g2 = &Date::getMonth;
  g3 = &Date::getDay;
}

void Date::us()
{
  sep = '/';
  g1 = &Date::getYear;
  g2 = &Date::getMonth;
  g3 = &Date::getDay;
}
```

```
int main()
{
  Date d;
  d.set(2017,4,20);
  d.hu();
  std::cout << d << std::endl;
  d.us();
  std::cout << d << std::endl;
}
```

```
2017.4.20
4/20/2017
```

```
std::ostream& operator<<(std::ostream& os, const Date& d)
{
  os << (this->*g1)() << sep << (this->*g2)() << sep << (this->*g3)();
  return os;
}
```

Static members

```
#include <iostream>

class Date
{
public:
  void set (int y, int m, int d);
  int  getYear() const  { return _year; }
  int  getMonth() const { return _month; }
  int  getDay() const   { return _day; }

  void hu();
  void us();
private:
  int year_;
  int month_;
  int day_;

  int (Date::*g1)() const = &Date::getYear;
  int (Date::*g2)() const = &Date::getMonth;
  int (Date::*g3)() const = &Date::getDay;

  char sep = '.';
};
```

Static members

```
#include <iostream>

int (Date::*g1)() const = &Date::getYear;    // accessible by anybody
int (Date::*g2)() const = &Date::getMonth;
int (Date::*g3)() const = &Date::getDay;
char sep = '.';

void hu();                                   // not connected logically
void us();                                   // to class Date

class Date
{
public:
  void set (int y, int m, int d);
  int  getYear() const  { return _year; }
  int  getMonth() const { return _month; }
  int  getDay() const   { return _day; }

private:
  int year_;
  int month_;
  int day_;
};
```

Static members

```
#include <iostream>

class Date
{
public:
  void set (int y, int m, int d);
  int  getYear() const  { return _year; }
  int  getMonth() const { return _month; }
  int  getDay() const   { return _day; }

  static void hu();
  static void us();
private:
  int year_;
  int month_;
  int day_;

  static int (Date::*g1)() const = &Date::getYear;   // private,
  static int (Date::*g2)() const = &Date::getMonth;  // not accessible from
  static int (Date::*g3)() const = &Date::getDay;    // outside of Date class

  static char sep = '.';
};
```

Static members

```
void Date::hu()
{
  sep = '.';
  g1 = &Date::getYear;
  g2 = &Date::getMonth;
  g3 = &Date::getDay;
}

void Date::us()
{
  sep = '/';
  g1 = &Date::getYear;
  g2 = &Date::getMonth;
  g3 = &Date::getDay;
}
```

```
int main()
{
  Date::us(); // can be used without object
  Date d;
  d.set(2017,4,20);
  std::cout << d << std::endl;
  d.hu();  // same as Date::hu()
  std::cout << d << std::endl;
  d::us(); // same as Date::us()
  std::cout << d << std::endl;
}

4/20/2017
2017.4.20
4/20/2017
```

```
std::ostream& operator<<(std::ostream& os, const Date& d)
{
  os << (this->*g1)() << sep << (this->*g2)() << sep << (this->*g3)();
  return os;
}
```

Static const

```
// x.h

class X
{
  static const int  c1 = 7;    // ok,scalar
  static       int  i2 = 8;    // error: not const
  const        int  c3 = 9;    // C++11: ok
  static const int  c4 = f(2); // error: non-const init.
  static const float f = 3.14; // ok,scalar
};


// x.cpp

const int X::c1; // initializer is here xor in-class  DO NOT FORGET!
```

Constexpr in C++14

```
class Point
{
public:
    constexpr Point(double xVal=0, double yVal=0) : x(xVal),y(yVal) noexcept {}
    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }

    // can be constexpr since C++14
    constexpr void setX(double newX) noexcept { x = newX; }
    constexpr void setY(double newY) noexcept { y = newY; }
private:
    double x, y;
};

constexpr Point p1(42.0, -33.33); // fine, constexpr ctor during compilation
constexpr Point p2(25.0, 33.3);   // also fine

constexpr Point midpoint(const Point& p1, const Point& p2) noexcept
{
    return { (p1.xValue() + p2.xValue()) / 2,   // call constexpr
             (p1.yValue() + p2.yValue()) / 2 }; // member funcs
}

constexpr auto mid = midpoint(p1, p2); // init constexpr object with
                                       // result of constexpr
```

Constexpr lambda in C++17

- Constexpr lambda (+ template lambda)
- Closure objects are literal types (as long as captured members are literal types)

```
template <typename I>
constexpr auto adder(I i) {
  //use a lambda in constexpr context
  return [i](auto j){ return i + j; };
}

//constexpr closure object
constexpr auto add5  = adder(5);
constexpr auto add15 = adder(15);

template <unsigned N>
class X{};

int foo()
{
  //use in a constant expression
  X<add5(22)> x27;
  int t25[add15(10)];
}
```