FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Ausbildung von
Mathematisch-Technischen Software-Entwicklern

Programming in C++
Part II

Bernd Mohr

FZJ-JSC-BHB-0155

1. Auflage

(letzte Änderung: 19.09.2003)

Publikationen des JSC stehen in druckbaren Formaten (PDF auf dem WWW-Server des Forschungszentrums unter der URL: <http://www.fz-juelich.de/jsc/files/docs/> zur Verfügung. Eine Übersicht über alle Publikationen des JSC erhalten Sie unter der URL: **<http://www.fz-juelich.de/jsc/docs>** .

Beratung

Tel: +49 2461 61 -nnnn

Auskunft, Nutzer-Management (Dispatch)

Das Dispatch befindet sich am Haupteingang des JSC, Gebäude 16.4, und ist telefonisch erreichbar von

Montag bis Donnerstag	8.00 - 17.00 Uhr
Freitag	8.00 - 16.00 Uhr

Tel. 5642 oder 6400, Fax 2810, E-Mail: **dispatch.jsc@fz-juelich.de**

Supercomputer-Beratung

Tel. 2828, E-Mail: **sc@fz-juelich.de**

Netzwerk-Beratung, IT-Sicherheit

Tel. 6440, E-Mail: **junet-helpdesk@fz-juelich.de**

Rufbereitschaft

Außerhalb der Arbeitszeiten (montags bis donnerstags: 17.00 - 24.00 Uhr, freitags: 16.00 - 24.00 Uhr, samstags: 8.00 - 17.00 Uhr) können Sie dringende Probleme der Rufbereitschaft melden:

Rufbereitschaft Rechnerbetrieb:	Tel. 6400
Rufbereitschaft Netzwerke:	Tel. 6440

An Sonn- und Feiertagen gibt es keine Rufbereitschaft.

Fachberater

Tel. +49 2461 61 -nnnn

Fachgebiet	**Berater**	**Telefon**	***E-Mail***
Auskunft, Nutzer-Management, Dispatch	E. Bielitza	5642	***dispatch.jsc@fz-juelich.de***
Supercomputer	W. Frings	2828	***sc@fz-juelich.de***
JuNet/Internet, Netzwerke, IT-Sicherheit	T. Schmühl	6440	***junet-helpdesk@fz-juelich.de***
Web-Server	Dr. S. Höfler-Thierfeldt	6765	***webmaster@fz-juelich.de***
Backup, Archivierung	U. Schmidt	1817	***backup.jsc@fz-juelich.de***
Fortran	D. Koschmieder	3439	***fortran.jsc@fz-juelich.de***
Mathematische Methoden	Dr. B. Steffen	6431	***mathe-admin@fz-juelich.de***
Statistik	Dr. W. Meyer	6414	***mathe-admin@fz-juelich.de***
Mathematische Software	Dr. B. Körfgen, R. Zimmermann	6761, 4136	***mathe-admin@fz-juelich.de***
Graphik	Ma. Busch, M. Boltes	4100, 6557	***graphik.jsc@fz-juelich.de***
Videokonferenzen	M. Sczimarowsky, R. Grallert	6411, 6421	***vc.jsc@fz-juelich.de***
Oracle-Datenbank	M. Wegmann, J. Kreutz	1463, 1464	***oracle.jsc@fz-juelich.de***

Die jeweils aktuelle Version dieser Tabelle finden Sie unter der URL:
<http://www.fz-juelich.de/jsc/allgemeines/beratung>

Programming in C++

Part II

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Version 19 September 2003

Programming in C++

☆☆☆ Inheritance ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Inheritance — Motivation

- What has `FArray` bought us?
  - The size of the `FArray` need not be constant ✔
  - The `FArray` knows its own size ✔
  - `FArray` assignment: `fa4 = fa6;` ✔
  - Range checking: `cout << fa2[-3];  // still bad!` ✘
- What are our options for `FArray` range checking?
  - Add this capability to `FArray::operator[]()`
    - ➠ Problem: this may be slow, not always desired, e.g.:

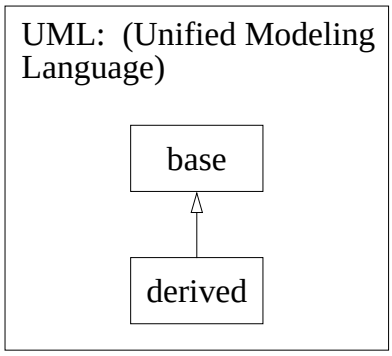
```
for(int i=0; i<fa3.size(); i++) fa3[i]=0.0;   // just fine
```

    vs.

```
cin >> farray_index;              // unsafe!
cout << fa1[farray_index];
```

  - Define a new class `safeFArray`
    - ➠ Problem: this will repeat lots of code, error prone
  - **Inheritance**: Create a subclass!

- Goal: a new, *derived* class (*subclass*) whose objects are both
  - objects of the *base* class (*superclass*), and also
  - specific, specialized objects of the base class



- **single most important rule**: a derived object ***is-a*** base object, but not vice versa
- Inheritance is commonly used in two ways
  - *reuse* mechanism
    - ➠ a way to create a new class that strongly resembles an existing one (like `safeFArray`)
  - *abstraction* mechanism:
    - ➠ a tool to organize classes into hierarchies of specialization
    - ➠ describe *relationships* between user-defined types

- Suppose our triangle class

```
class triangle { private: Coord v1, v2, v3; /*...*/ };
```

- Possible derived classes:
  - particular characteristic: `right_triangle`
  - additional field (`color`): `color_triangle`

➠ they pass the *is-a* test

- Not possible derived classes:
  - particular characteristic: `trianglar_pipe` (*is-a* `pipe`, but not a `triangle`)
  - additional field (`v4`): `rectangle`

➠ they fail the *is-a* test

- Note: `trianglar_pipe` will probably *include* `triangle` data member (*hasA*)

- Derived class *inherits* member functions and data members from base class
  - in addition to its own members
  - **exceptions:** constructors, destructors, `operator=()`, and friends are ***not*** inherited

- Access rules for derived class implementation:

members of base class which are	can be accessed in base class	can be accessed in derived class	compare to: client code
`public`	✔	✔	✔
`private`	✔	✘	✘
`protected`	✔	✔	✘

- General form of derived class definitions:

```
class DerivedClassName : AccessMethod BaseClassname {
   /* ... */
};
```

where *`AccessMethod`* is one of the following:

`public` inherited `public` base class members stay `public` in derived class

(`private` inherited `public` base class members become `private` in derived class)

➠ **does affect access rights of client code and classes which derive from derived classes**

➠ specifying AccessMethod `public` is important as the default is `private`!

- **Note**: the *is-a* test only applies to `public` inheritance!

Public inheritance

```
class b {
private:   int prv;
protected: int prt;
public:    int pub;
   void f() {
   // access to pub, prt, prv
   }
};

class d: public b {
private:   int dprv;
public:    int dpub;
   void df() {
   // pub, prt, dpub, dprv, f
   }
};

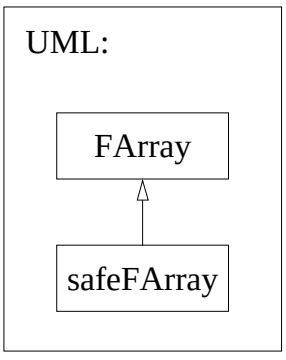
void func() {   //b::pub, b::f
   //d::pub, d::dpub, d::f, d::df
}
```

Private inheritance

```
class b {
private:   int prv;
protected: int prt;
public:    int pub;
   void f() {
   // access to pub, prt, prv
   }
};

class d: private b {
private:   int dprv;
public:    int dpub;
   void df() {
   // pub, prt, dpub, dprv, f
   }
};

void func() {   //b::pub, b::f
   //d::pub, d::dpub, d::f, d::df
}
```

- how about our `safeFArray` class?

```
#include "farray.h"

class safeFArray : public FArray {
public:
   // new constructors needed (not inherited)

   // destructor and operator= not inherited but default OK

   // override definition of operator[]() and operator[]() const
};
```

- Placed in "`safefarray.h`"

- Constructors for the base class are always called first, destructors for base classes last. e.g.,

```
class base { public:
   base()  { cout << "constructing base" << endl; }
   ~base() { cout << "destructing base" << endl; }
};

class derived : public base {
public:
   derived(){ cout << "constructing derived" << endl; }
   ~derived(){ cout << "destructing derived" << endl; }
};

int main(int, char **) {
   derived d;
}
```

will print out

```
constructing base
constructing derived
destructing derived
destructing base
```


- Problem: if not otherwise specified, default constructor of base class is called
  - ➠ for `safeFArray`, we always would end up with `FArray(def_farray_size)`
- Solution: use the following syntax to specify the base constructor to call:
  (special case of general *member initialization list*)

```
class safeFArray : public FArray {

public:
   safeFArray(int size = def_FArray_size) : FArray(size) {}

   safeFArray(const FArray& rhs) : FArray(rhs) {}

   safeFArray(const float* fa, int size) : FArray(fa, size) {}

// ...
};
```

- Note: nothing else needs to be constructed
  - ➠ copy constructor can take `FArray`'s
- Note: only *direct* base class initialization possible

- we like and keep declaration/interface **and** must match return value as well, but
- definition needs reworking, e.g.,

```
#include <assert.h>
#include "safefarray.h"

float& safeFArray::operator[](int index) {
   assert(index >= 0 && index < size());       // exit on failure
   return FArray::operator[](index); // reuse base implementation
}

const float& safeFArray::operator[](int index) const {
   // same as above...
}
```

- both versions of `operator[]()`, const and non-const, must be overridden
- if possible, reuse base function definition
- Placed in "`safefarray.cpp`"
- `safeFArray::operator[]()` replaces `FArray::operator[]() !!!` (unlike constructors/destructors)

- Rule: derived objects, pointers to derived objects, and references to derived objects are automatically converted to corresponding base objects if necessary:

```
class base { /*...*/ };

class derived : public base {
   /*...*/
};

void base_func (base& b);
void devd_func (derived& d);

base b;
derived d;

base_func(b);
base_func(d);   // d is-a base
devd_func(d);
devd_func(b);   // error!
```

```
class car { /*...*/ };

class rv : public car {
   /*...*/
};

void drive (car& c);
void live_in (rv& r);

car c;
rv r;

drive(c);
drive(r);       // r is-a car
live_in(r);
live_in(c);     // error!
```

➠ C++ permissions follow *is-a* model.

- if a inherited member function is overridden by a derived class and called through a *pointer* or *reference*, the type of the *pointer/reference object itself* determines what function gets called:

```
class base {
   void foo(void);
};
class derived : public base {
   void foo(void);
};

derived d, *pd=&d, &rd2=d;
base b, *pb=&b, &rb=b, &rd1=d;

pb->foo();   // base::foo()
rb.foo();
pd->foo();   // derived::foo()
rd2.foo();

pb = pd;     // or: pb = &d;
pb->foo();   // base::foo()!!!
rd1.foo();
```

```
class car {
   void park(void);
};
class rv : public car {
   void park(void);
};

rv r, *pr=&r, &rr2=r;
car c, *pc=&c, &rc=c, &rr1=r;

pc->park();  // parks car
rc.park();
pr->park();  // parks rv
rr2.park();

pc = pr;     // rv is-a car
pc->park();  // parks rv like
rr1.park();  // a car (ouch!)
```

➠ *static binding*, i.e., type of member function determined at compile time

- if a **`virtual`** member function is overridden by a derived class and called through a *pointer* or *reference*, the type of the *pointed to/referenced object* determines what function gets called:

```
class base {
   virtual void foo(void);
};
class derived : public base {
   virtual void foo(void);
};

derived d, *pd=&d, &rd2=d;
base b, *pb=&b, &rb=b, &rd1=d;

pb->foo();   // base::foo()
rb.foo();
pd->foo();   // derived::foo()
rd2.foo();

pb = pd;     // or: pb = &d;
pb->foo();   // derived::foo()
rd1.foo();   // good!
```

```
class car {
   virtual void park(void);
};
class rv : public car {
   virtual void park(void);
};

rv r, *pr=&r, &rr2=r;
car c, *pc=&c, &rc=c, &rr1=r;

pc->park();  // parks car
rc.park();
pr->park();  // parks rv
rr2.park();

pc = pr;     // rv is-a car
pc->park();  // parks rv like
rr1.park();  // a rv (good!)
```

➠ *dynamic binding*, i.e., type of member function determined at run time

- Rule: Never redefine an inherited non-virtual function!
- Derived class by adding fields to subclass:
  - likely `operator=()` and `operator==()` (at least) will change
  - make them `virtual` in base class
- Destructor called through pointer when using `new` / `delete`
  - static / dynamic binding rules applies

```
base *pb; derived *pd = new derived;

// ... later
pb = pd; delete pb;  // only calls base::~base()!!!
```

  - make destructors `virtual` in base class (if base has >1 virtual member function; if not, shouldn't be base class in the first place)
- But: `virtual` functions cause *run-time* (indirect call) and *space* (virtual table) overhead
- If function needs to be `virtual`
  - another reason to use member vs. global function
- **Virtual functions supply subclasses with default definitions**

- What if there is no (useful) default definition?

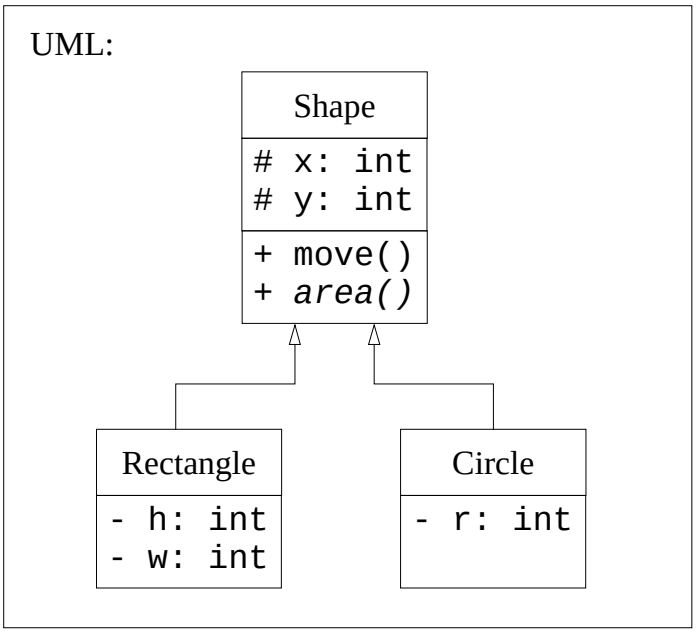
```
class Shape {
public:
   virtual double area(void) const = 0;           // pure virtual
   // ...
};

class Rectangle : public Shape { /* ... */ };

class Circle : public Shape { /* ... */ };
```

- effects of ≥ 1 pure virtual function(s)
  - `Shape` is an *abstract base class* (ABC)
  - cannot create objects of class `Shape`
  - but can build subclasses on top of `Shape`
  - subclasses (`Rectangle` and `Circle`) must
    - implement pure virtual functions (and become *concrete* subclasses), or
    - inherit them, and become an ABC themselves

- Example:
  Define framework for geometrical shapes like rectangles and circles.
  Each shape has a center (x, y) and methods to move the object and to calculate its area.



\+ → public member
\# → protected member
\- → private member

abstract member

- shape.h:

```
class Shape {
protected: int x_, y_;  // center
public:    Shape(int x, int y) : x_(x), y_(y) {}
           virtual double area() const = 0;
           void move(int dx, int dy) { x_ += dx; y_ += dy; }
};

class Rectangle : public Shape {
private:   int h_, w_;
public:    Rectangle(int x, int y, int h, int w)
                              : Shape(x, y), h_(h), w_(w) {}
           virtual double area() const { return h_*w_; }
};

class Circle : public Shape {
private:   int r_;
public:    Circle(int x, int y, int rad) : Shape(x, y), r_(rad) {}
           virtual double area() const { return r_*r_*3.1415926; }
};
```

- Main program:

```
#include <iostream>
using namespace std;
#include "shape.h"

double area(Shape *s[], int n) {
   double sum = 0.0;
   for (int i=0; i<n; ++i) sum += s[i]->area();
   return sum;
}

int main(int, char**) {
   Shape *shapes[3];
   shapes[0] = new Rectangle(2,6,2,10);
   shapes[1] = new Circle(0,0,1);
   shapes[2] = new Rectangle(3,4,5,5);

   cout << "area: " << area(shapes, 3) << endl;
   for (int i=0; i<3; ++i) shapes[i]->move(10, -4);
}
```

- prints: `area: 48.1416`


```
class Shape {
public:    enum kind { REC, CIR };
           Shape(int x, int y, int h, int w)
              : x_(x), y_(y), h_(h), w_(w), t_(REC), r_(0) {}
           Shape(int x, int y, int r)
              : x_(x), y_(y), h_(0), w_(0), t_(CIR), r_(r) {}
           void move(int dx, int dy) { x_ += dx; y_ += dy; }
           double area() {
              switch (t_) {
                 case REC:  return h_*w_;
                 case CIR:  return r_*r_*3.1415926;
                 default:   return 0.0;
              }
           };
private:   int x_, y_, h_, w_, r_;
           kind t_;
};
```

- Adding new shape requires changes everywhere `kind` is used (don't forget one!)
- Recompilation of all source files necessary which depend on `Shape`!

Summary: for `public` inheritance (derived *is-a* base), the following rules apply:

- pure virtual function
  - function interface only is inherited
  - concrete subclasses ***must*** supply their own implementation
- simple (non-pure) virtual function
  - function interface and default implementation is inherited
  - concrete subclasses ***can*** supply their own implementation
- non-virtual function
  - function interface and mandatory implementation is inherited
  - concrete subclasses ***should not*** supply their own implementation

- **Function virtuality is an important C++ feature**
- **But: polymorphism is not the solution to every programming problem (KISS principle!)**

- Only member functions can be `virtual`
- Problem: how to make class related global functions (e.g., `operator<<()`) behave correctly for inheritance
- introduce `public virtual` helper method (e.g., `print()`)
- derived classes can redefine helper method if necessary

```
class foo {
public:
   virtual ostream& print(ostream& ostr) {
      // usual implementation of output operator here
      ...
   }
   ...
};
ostream& operator<<(ostream& ostr, const foo& f) {
   return f.print(ostr);
}
```

```
class base {
public:
   virtual void func() const { cout << "base::func" << endl; }
};
class drvd : public base {
public:
   virtual void func() const { cout << "drvd::func" << endl; }
};
void gen_func(base b) { b.func(); }
derived d; gen_func(d);              // prints: base::func !!!
```

- Problem: argument `b` of `gen_func` is passed by *value*
  - copy constructor is invoked
  - `b` is not a reference or pointer ➟ *static* binding ➟ copy constructor of `base` is invoked
  - copies only `base` part (*slicing*)
- **Another reason to pass class objects by** *reference:*

```
void gen_func(const base& b) { b.func(); }
```


- Recall rules:
  - Assign to all members
  - Check for assignment to self
  - Return a reference to `*this`
- New rule: derived class's assignment operator must also handle base class members!

```
class base {
private:
 int x;
public:
 base(int i) : x(i) {}
};
```

```
class derived : public base{
private:
 int *y;
public:
 derived(int i) : base(i), y(0) {}
 derived& operator=(const derived& rhs);
};
```

```
derived& operator=(const derived& rhs) {
        if (this == &rhs) return *this;
        base::operator=(rhs);      // don't forget this
        if (y) { *y = *(rhs.y); } else { y = new int(*(rhs.y)); }
        return *this;
}
```

Rules for overloading base class functions in derived classes:

```
class base {
public:
   virtual void f(int) {}
   virtual void f(double) {}
   virtual void g(int i = 10) {}
};
```

```
class Derived: public Base {
public:
   void f(Complex) {}

   void g(int i = 20) {}
};
```

```
Base b;  Derived d;  Base *pb = new Derived;
```

- A derived function with the same name but *without* a matching signature **hides** all base class functions of the same name

```
d.f(1.0);                    // Derived::f(Complex) !!!
```

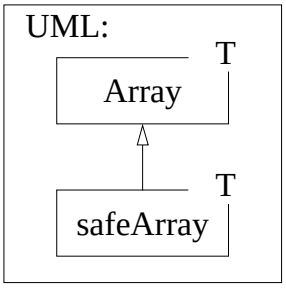
  - use "using" declarations to bring them into scope

- Never change the default argument values of overridden inherited functions

```
b.g();                       // Base::g(int = 10)
d.g();                       // Derived::g(int = 20)
pb->g();                     // Derived::g(int = 10) !!!
```

  - default taken from base class function because compiler does it at compile time

- We had an `Array<T>` template; how about `safeArray<T>`?




```
#include <assert.h>
#include "array.h"

template<class T>
class safeArray : public Array<T> {

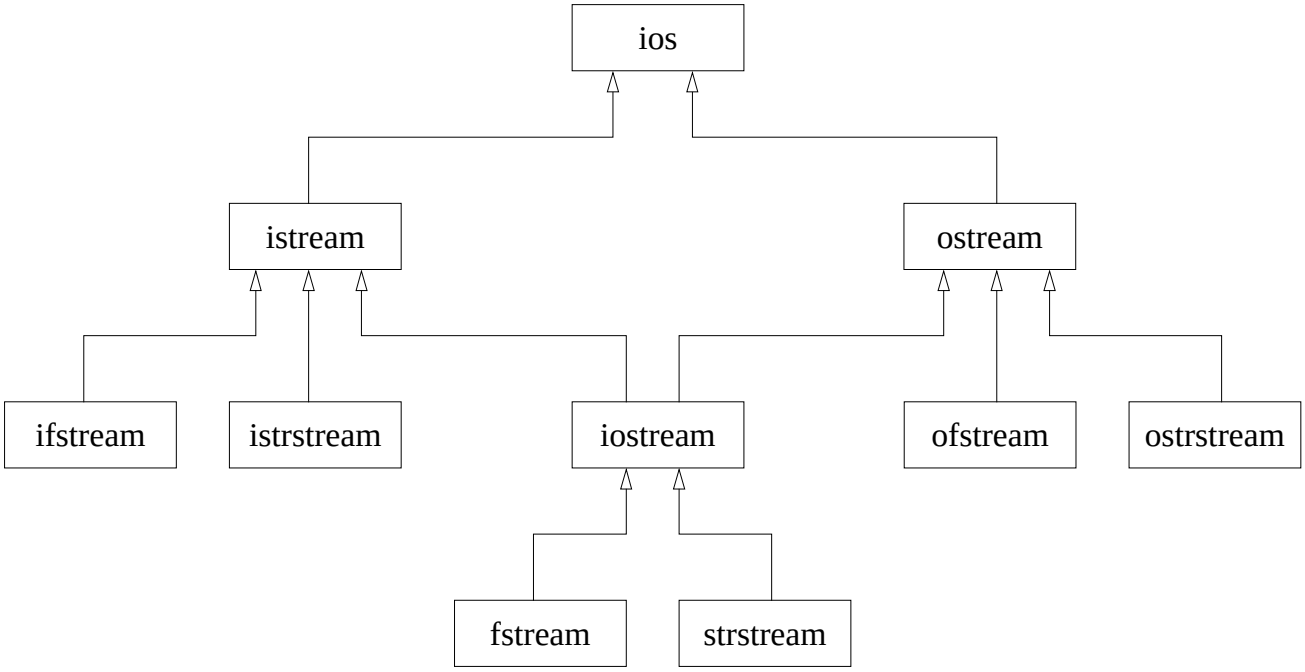
public:
   safeArray(int size = def_array_size) : Array<T>(size) {}
   safeArray(const Array<T>& rhs) : Array<T>(rhs) {}
   safeArray(const T* ay, int size) : Array<T>(ay, size) {}

   T& operator[](int index) {
      assert(index >= 0 && index < Array<T>::size());
      return Array<T>::operator[](index);
   }
};
```

- Instantiation: `safeArray<float> ssa(14);` causes base and derived class generation
- Note: `remove_hot()` should also be re-written into a function template

```
template<class T> void remove_hot(Array<T>& day_temp);
```

- User-defined input and output operators work for all IOStreams because IOStream classes actually form inheritance hierarchy!



- Allow arrays with arbitrary bounds (not just `0` to `n-1`):

```
template<class T>                                          // boundarray.h
class boundArray : public safeArray<T> {
public:
   boundArray(int lowIdx, int highIdx)
            : safeArray<T>(highIdx-lowIdx+1), low(lowIdx) {}
   boundArray(const boundArray<T>& rhs)
            : safeArray<T>(rhs), low(rhs.low) {}
   boundArray(int lowIdx, int highIdx, const T* ay)
            : safeArray<T>(ay, highIdx-lowIdx+1), low(lowIdx) {}

   boundArray<T>& operator=(const boundArray<T>& rhs);

   T& operator[](int index);
   const T& operator[](int index) const;

   int lowerBound() const { return low; }
   int upperBound() const { return low+safeArray<T>::size()-1; }

private:
   int low;
};
```

```
                                                    // boundarray.cpp
template<class T>
boundArray<T>& boundArray<T>::operator=(const boundArray<T>& rhs)
{
   if (this == &rhs) return *this;
   safeArray<T>::operator=(rhs);
   low = rhs.low;
   return *this;
}

template<class T>
T& boundArray<T>::operator[](int index) {
   return safeArray<T>::operator[](index - low);
}

template<class T>
const T& boundArray<T>::operator[](int index) const {
   return safeArray<T>::operator[](index - low);
}
```

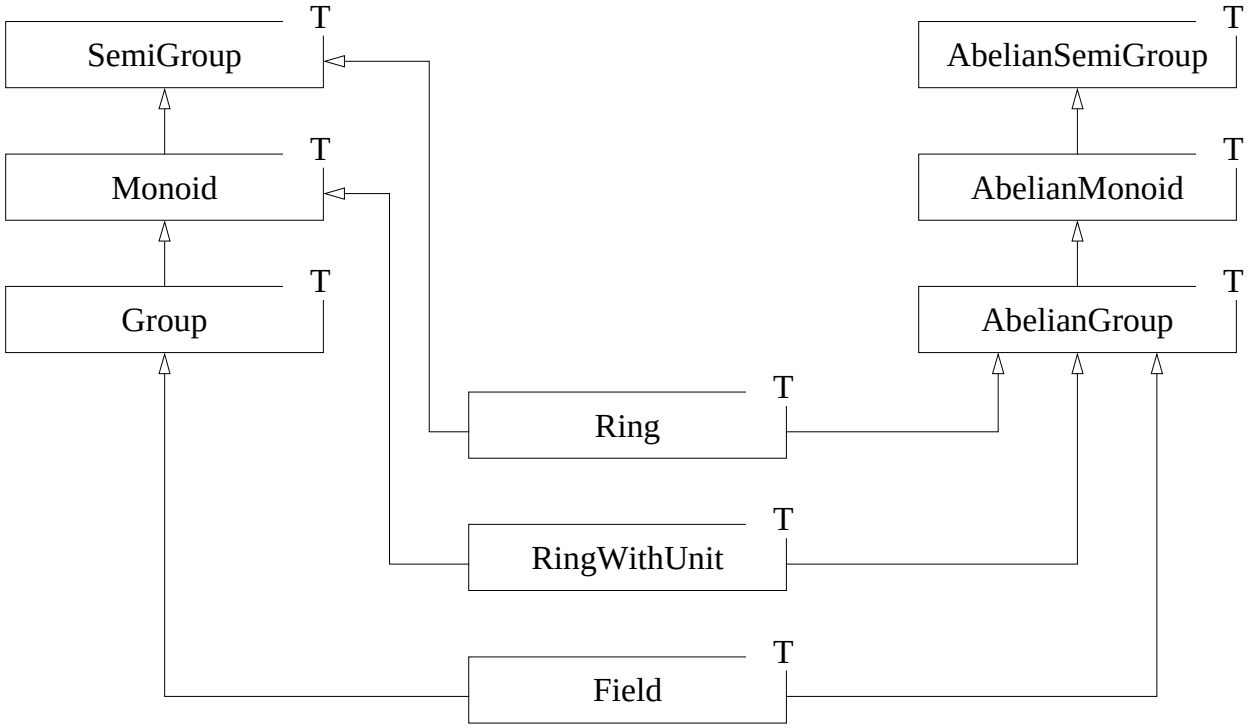

- Better solution for automatically defining mathematical operators:

```
template<class T> class Ring {       // incomplete
public:
   // inline friend: defines global function within class
   friend const T operator*(const T& lhs, const T& rhs) {
      return T(lhs) *= rhs;
   }
   friend const T operator+(const T& lhs, const T& rhs) {
      return T(lhs) += rhs;
   }
   T& operator++() { return ((T&)*this) += T(1); }
   const T operator++(int) {
      T old((T&)*this); ++(*this); return old;
   }
};
```

- Now, number classes only define base operators `+=`, `*=` and `T(1)`, then derive from `Ring<T>`:

```
class Complex : public Ring<Complex> { ... };
```

- Why stop with `Ring<T>`? ➠ define classes for other algebraic structures as well



- **Not** every *is-a* relationship in our natural environment is a candidate for inheritance
- Do **not** use inheritance
  - if not all methods of the base class make sense for the derived class
  - if methods would have a different semantic in the derived class
  - if the meaning of components would change in the derived class
  - if values or properties of components would be restricted in the derived class

  Example: A `Square` is a `Rectangle` but the method `change_width()` or `change_height()` would not make sense for square!

- For every possible derived object, it must make sense to assign it to a object of the corresponding base class. In doing so, new (additional) components of the derived class are ignored.
- Do not mistake *is-like-a* for *is-a*!
- Use inheritance carefully!

- *Multiple inheritance* := inherit from more than one base class

```
class derived : public base1, public base2 { /* ... */ };
```

- Experts are disagreeing whether multiple inheritance is useful or not
  - It is essential to the natural modeling of real-world problems
  - It is slow, difficult to implement, and no more powerful than single inheritance
- Object-orient Languages:
  - C++, Eiffel, Common LISP Object System (CLOS) offer multiple inheritance
  - Smalltalk, Objective C, Object Pascal do not
  - Java does not, but allows the implementation of multiple interfaces
- Fact: multiple inheritance in C++ opens up a Pandora's box of complexities

➠ **Use multiple inheritance judiciously**

- If a derived class inherits a member name from more than one class, any usage of that name is inherently ambiguous
  - ➠ you must explicitly say which member you mean

```
class Lottery {
public:
   virtual int draw();
};
```

```
class GraphicObj {
public:
   virtual int draw();
};
```

```
class LotterySimulation: public Lottery, public GraphicObj {...};

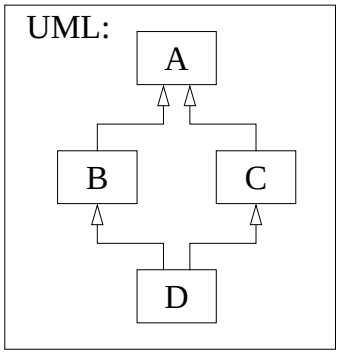
LotterySimulation *pls = new LotterySimulation;
pls->draw();                // error! ambiguous
pls->Lottery::draw();       // OK
pls->GraphicObj::draw();    // OK
```

- By explicitly qualifying a virtual function, it doesn't act virtual anymore
- It is impossible that `LotterySimulation` can redefine both of the `draw()` functions

Multiple Inheritance — Ambiguity^2: the Diamond

- Sooner or later, you come across the following inheritance relationship:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };
```



- `D` includes two copies of `A`
  - unnecessary duplication
  - no syntactic means of distinguishing them
- Solution: *virtual base classes*

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* ... */ };
```

  - only first copy of A is used
  - Problem: specification has to be done of class designer (user/client may not have access)

Multiple Inheritance — Ambiguity^2: the Diamond

- Another problem: an object of class `D` may have *up to five different* addresses:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };

D d;

D* pd1 = &d;    // pd1 is address of D part of D
B* pd2 = &d;    // pd2 is address of B part of D
C* pd3 = &d;    // pd3 is address of C part of D
A* pd4 = (B*) &d;      // pd4 is address of A part of B part of D
A* pd5 = (C*) &d;      // pd5 is address of A part of C part of D
```

- our simple self-test for `operator=()` fails (by checking the address of the object):

```
if (&rhs == this) return;
```

- almost impossible to come up with efficient solution
  (best solution: implement unique object identificator)

- Passing constructor arguments to virtual base classes
  - normally, classes at level `n` of the hierarchy pass arguments to the classes at level `n-1`
  - For virtual base classes, arguments are specified in the classes *most derived* from the base
- Dominance of virtual functions

```
class A { virtual void mf(); /* ... */ };
class B : public A { virtual void mf(); /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };

D *pd = new D;
pd->mf();       // A::mf() or B::mf()?
```

  - ➠ normally ambiguous, but `B::mf()` *dominates* if `A` is virtual base class for both `B` and `C`
- Casting restrictions
  - C++ prohibits to cast down from a pointer/reference to a virtual base class to a derived class

➠ **There are cases where multiple inheritance might be useful**

➠ **Rule: try to avoid virtual base classes (diamond-shaped class hierarchy)**

Programming in C++

☆☆☆ More on Arrays ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

More on Arrays — Motivation

- What have we got so far with our `array` redesign?
  - size of array need not be constant and known at compile-time
  - `array` assignment and re-assignment
  - optional range checking through derived `safearray`
  - generic usage through templates
- What else could be done?
  - adding mathematical operators to get a class `vector`
    - ➠ `vector` can easily implemented by inheriting from `array` or `safearry`
  - what about multi-dimensional arrays like `matrix, array3d, ...?`
    - ➠ implementation?
  - performance of `vector` or `matrix` operations compared to hand-coded loop expressions?

```
for (int i=0; i<size; i++) a[i] = b[i] * c + d[i];
```

  - Functions for selecting parts of an `array` or `vector` (e.g., projections)
  - Sparse arrays, BLAS, LAPACK, `...`

- Derive `vector` from `array` and add mathematical operations

```
#include "array.h"
#include <assert.h>

template<class T> class vector : public array<T> {
public:
   vector(int size = def_array_size) : array<T>(size) {}
   vector(const array<T>& rhs) : array<T>(rhs) {}
   vector(const T* ay, int size) : array<T>(ay, size) {}

   vector<T>& operator+=(const vector<T>& rhs) { // elem. add
      assert (sz == rhs.sz);   // same size?
      for (int i=0; i<sz; ++i) ia[i] += rhs.ia[i];
      return *this;
   }
   vector<T>& operator+=(T rhs) {  // elementwise scalar add
      for (int i=0; i<sz; ++i) ia[i] += rhs;
      return *this;
   }
   // ...
};
```


- Define elementwise addition out of `operator+=` as usual:

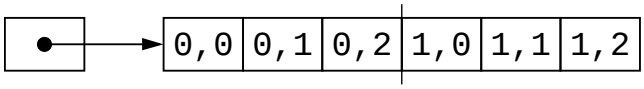
```
template<class T>
vector<T> vector<T>::operator+(const vector<T>& rhs) {
   return vector<T>(*this) += rhs;
}

template<class T>
vector<T> vector<T>::operator+(T rhs) {
   return vector<T>(*this) += rhs;
}
```

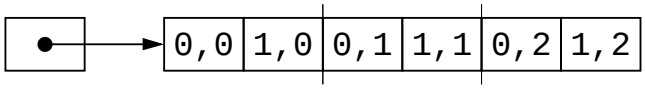
- Global definition of `operator+` to get conversion on both arguments not helpful here, as conversion not defined (for `T=int` even disastrous as `vector` of zeroes of size *n* is added!)

- To complete simple `vector` implementation also implement
  - elementwise substraction, multiplication, division, `...`
  - elementwise trigonometrical (`sin`, `cos`, `...`) and other mathematical functions (`abs`, `...`)
  - dot product, norm, minimum and maximum value or index
  - `...`

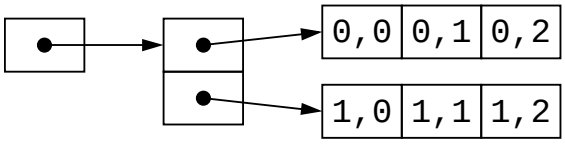
- Implementations of `matrix(2,3);` = $\begin{bmatrix}(0,0) & (0,1) & (0,2)\\(1,0) & (1,1) & (1,2)\end{bmatrix}$



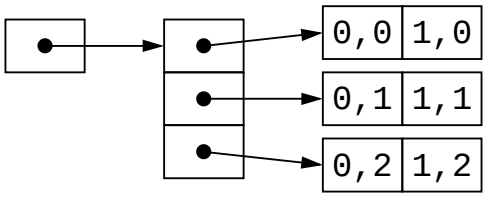
row-major order (C), contiguous



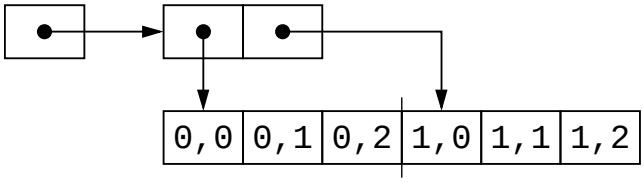
column-major order (Fortran), contiguous



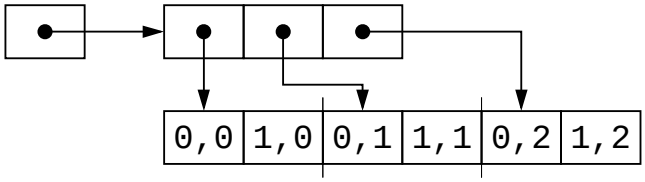
non-contiguous, row vectors



non-contiguous, column vectors



contiguous, row vectors



contiguous, column vectors

➠ even more layouts (sparse, symmetric, packed, `...` representations)

- overloaded versions of `operator[]` not possible, e.g., `operator[](int,int)`
- but we can use overloaded versions of `operator()` to specify subarray selections
  - `operator()(int)` is just a replicated version of `operator[]`

```
const T& operator()(int index) const { return ia[index]; }
```

  - `operator()(int,int)` can be used to select a subarray `[start, end]`

```
array<T> operator()(int start, int end) const {
   return array<T>(ia+start, end-start+1);
}
```

  - `operator()(int,int,int)` then for strided subarrays `[start, end, stride]`

```
array<T> operator()(int start, int end, int stride) const {
   array<T> r((end-start+stride)/stride);
   for (int i=start, j=0; i<=end; i+=stride) r.ia[j++] = ia[i];
   return r;
}
```

- **Not recommended:** could use `operator[]` for indexing with range check, `operator()` without or vice versa (can you remember which is which?)

- Overloaded `operator()` for subarray selection is only sub-optimal solution
  - ➠ e.g., how do you implement it for class `matrix`?
- Better solution:
  - use overloaded `operator()` for indexing in different dimensions
  - use `Range` objects (`Region` for `matrix`, ...) for subarray selection

```
class Range {
public:
   Range (int base);
   Range (int base, int end, int stride = 1);
   ...
};

template <class T>
array<T> array<T>::operator()(const Range&) { /*...*/ }

array<double> a(100), b(100);
a = b(Range(4, 9));
```


- Mathematical relationships between vector, matrix, `...` calls for systematic design
- but multi-dimensional array design can be done in a variety of ways
  - ➠ different uses of arrays require different trade-offs among time, space efficiency, compatibility with existing code (e.g. Fortran libraries), and flexibility
  - ➠ no single array class can cope with the wide range of requirements in practice
- As an example, we have a look on the design used in [Barton and Nackman]
  They distinguish two levels of flexibility:
  - *Rigid arrays*: dimensionality and shape fixed at compile time
  - *Formed arrays*: dimensionality fixed, but shape determined/changeable at runtime

  And they distinguish two kinds based on the way clients can use them:
  - *Interfaced arrays*: shares common *interface base classes* with other array classes; all classes with the same interface can be used interchangeably in client functions
    - ➠ more efficient in code space and programmer time
  - *Concrete arrays*: no common interface base class; no virtual function call overhead
    - ➠ more efficient in function-call time (runtime)

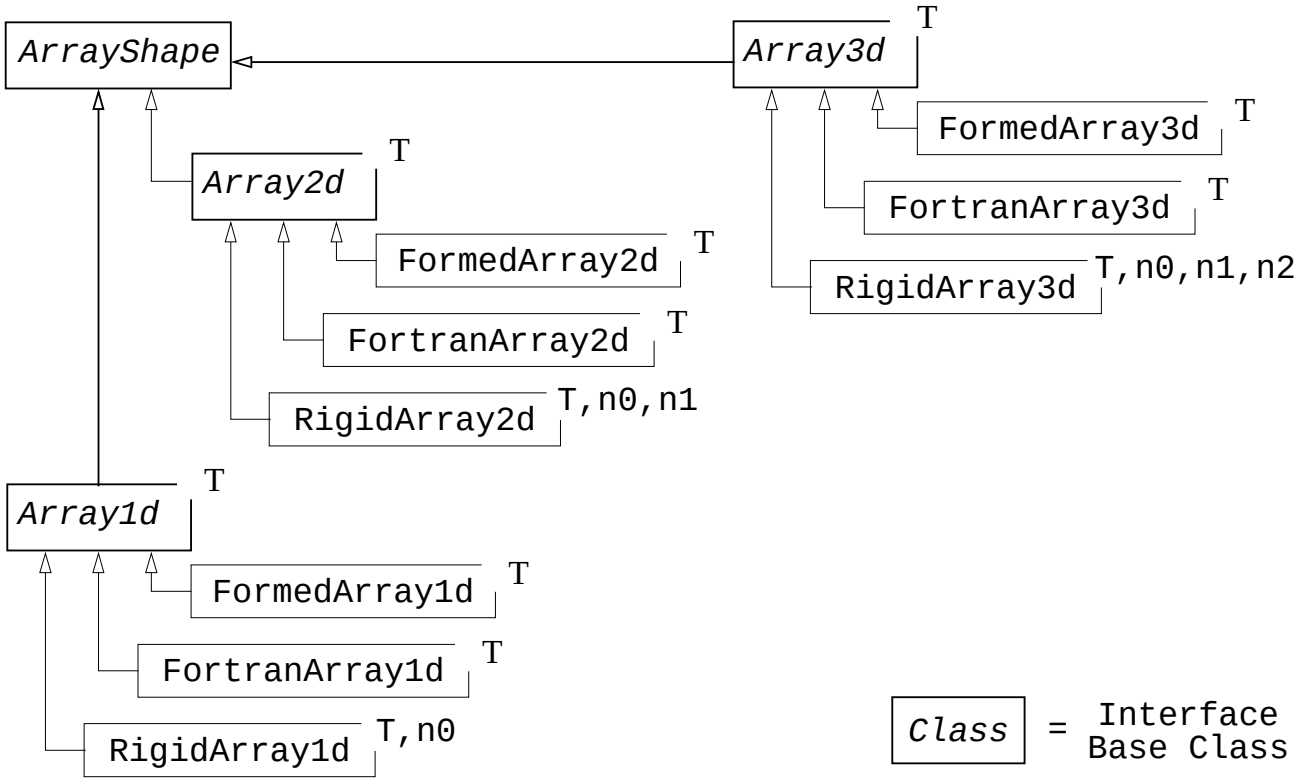
- How can we implement rigid arrays using templates?
  - Non-type template arguments

```
template<int N, class T>
class array {
private:
   T data[N];
   // ...
};
Array<100,int>  a;                    // like int a[100];
```

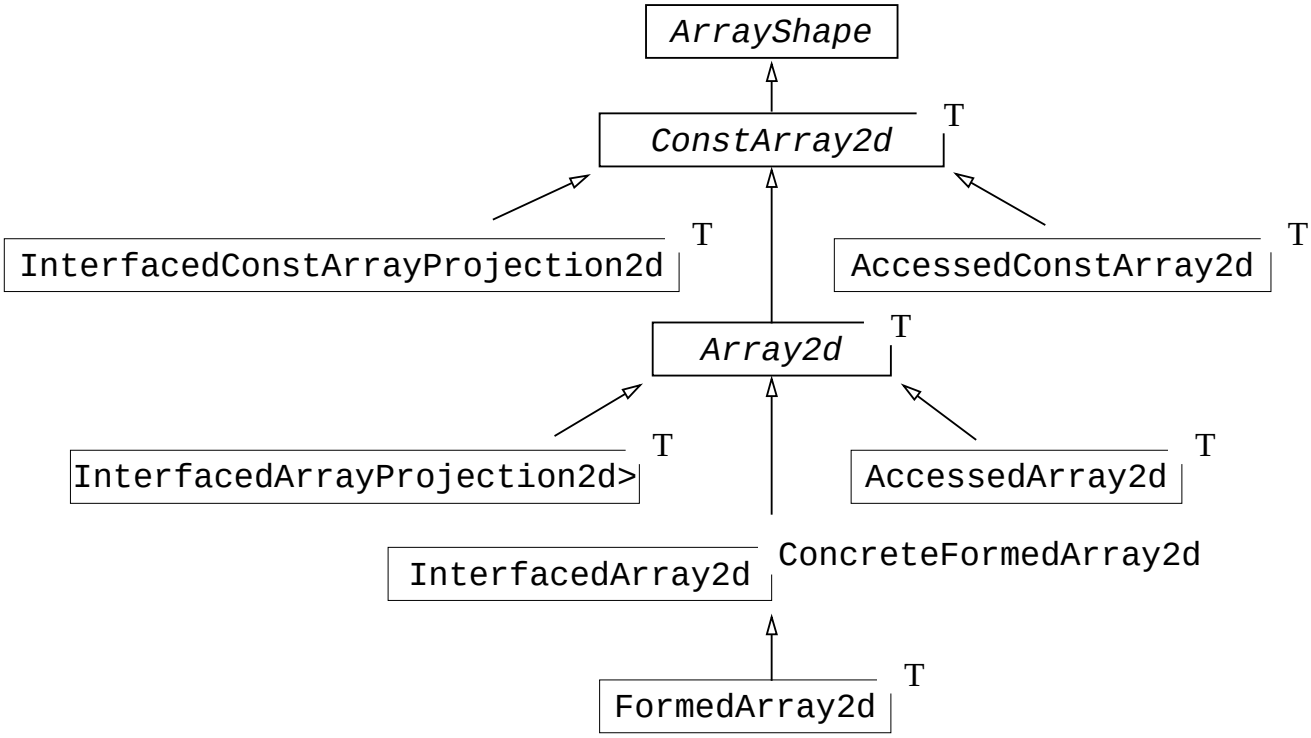

- Example concrete array classes indented for use by client functions

Class name	Storage layout
`ConcreteFormedArray1d<T>`	Row major, contiguous
`ConcreteRigidArray1d<T,n0>`	Row major, contiguous
`ConcreteFortranArray1d<T>`	Column major, contiguous
`ConcreteFormedArray2d<T>`	Row major, contiguous
`ConcreteRigidArray2d<T,n0,n1>`	Row major, contiguous
`ConcreteFortranArray2d<T>`	Column major, contiguous
`ConcreteFortranSymPackedArray2d<T>`	Upper triangular, packed
`ConcreteFormedArray3d<T>`	Row major, contiguous
`ConcreteRigidArray3d<T,n0,n1,n2>`	Row major, contiguous
`ConcreteFortranArray3d<T>`	Column major, contiguous

- Interfaced array class hierarchy overview



- Class hierarchy details for `FormedArray2d<T>` showing 2-dimensional interfaces, projections and accessors



- `vector` class definition

```
template <class T, int N>
class vector {
   private:   T *begin, *end;
    public:   vector();
              vector(const vector<T,N>& rhs);
              ~vector();
              T& operator[](int n);
              vector<T,N>& operator=(const vector<T,N>& rhs);
              vector<T,N> operator+(const vector<T,N>& rhs);
...
};
```

- Constructor: allocate data space and set end pointer

```
template <class T, int N>
vector<T,N>::vector() : begin(new T[N]), end(begin+N) {}
```

- Destructor: free data space

```
template <class T, int N>
vector<T,N>::~vector() { delete [] begin; }
```


- Copy Constructor: allocate data space and copy data over

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
   : begin(new T[N]), end(begin+N) {
   T* dest = begin;
   T* src  = rhs.begin;
   while (src != rhs.end) *dest++ = *src++;
}
```

- Assignment Operator: after self-test, copy data over
  - ➠ can reuse data space as we know that the vectors have same length

```
template <class T, int N>
vector<T,N>& vector<T,N>::operator=(const vector<T,N>& rhs) {
   if (&rhs == this) return *this;
   T* dest = begin;
   T* src  = rhs.begin;
   while (src != rhs.end) *dest++ = *src++;
   return *this;
}
```

- Addition Operators: elementwise addition, define `operator+()` out of `operator+=()`

```
template <class T, int N>
vector<T,N>& vector<T,N>::operator+=(const vector<T,N>& rhs) {
   T* dest = begin;
   T* src  = rhs.begin;
   while (dest!=end) *(dest++) += *(src++);
   return *this;
}
```

```
template <class T, int N>
vector<T,N> vector<T,N>::operator+(const vector<T,N>& rhs) {
   return vector<T,N>(*this) += rhs;
}
```

- Index Operator: nothing new here

```
template <class T, int N>
T& vector<T,N>::operator[](int n) { return begin[n]; }
```

➠ Rest of class methods left as exercise for the reader!

Problem: Overhead for short `vector` lengths

➠ Template member function specialization using manual loop unrolling and inlining

```
inline vector<double,3>&
vector<double,3>::operator=(const vector<double,3>& rhs) {
   if (&rhs == this) return *this;
   double *dest = begin;              // or:
   double *src  = rhs.begin;          //  begin[0] = rhs.begin[0];
   *dest++ = *src++;                  //  begin[1] = rhs.begin[1];
   *dest++ = *src++;                  //  begin[2] = rhs.begin[2];
   *dest = *src;
   return *this;
}
```

side note: partial specialization would be better (double ➠ T)
but not yet supported by mainstream compilers

```
template<class T> inline vector<T,3>&
vector<T,3>::operator=(const vector<T,3>& rhs) { /*...*/ }
```

❑ Example: elementwise `vector` addition

```
vector<T,N> u, v, w;

// later...                  // typical C (or Fortran-like version)
u = v + w;                   for (int i=0; i<N; ++i) u[i]=v[i]+w[i];
```

❑ Generated code (`GHOST` vector not generated if compiler features "*Return Value Optimization*")

```
v.operator+(w)
➟    vector<T,N> NoName(v);
     ➟    NoName.begin = new T[N]; loop NoName[i] ← v[i];
     NoName.operator+=(w);
     ➟    loop NoName[i] += w[i];
     return NoName;
     ➟    vector<T,N> GHOST(NoName);
          ➟    GHOST.begin = new T[N];
          ➟    loop GHOST[i] ← NoName[i];
          NoName.~vector();
u.operator=(GHOST);
➟    loop u[i] ← GHOST[i];
GHOST.~vector();
```


Problem: Too many temporary objects; e.g. `u = v + w;` ⇒ 2 `new` / `delete` calls!

➟ implement own memory allocation:
`delete` puts object in freelist; `new` uses freelist objects first

❑ Implementation using `Allocate` class modelled after STL:

```
template <class T, int N>
class vector {
private:  T *begin, *end;
          static Allocator<T,N> allocator_data;
public:
   vector() : begin(allocator_data.allocate()), end(begin+N) {}
   ~vector() { allocator_data.deallocate(begin); }
    ...
};

// Definition
template <class T, int N>
Allocator<T,N> vector<T,N>::allocator_data;

// with preallocation: ... ::allocator_data(100);
```

- Class definition

```
template <class T, int N>
class Allocator {
   private:  union {                  // data used either for
                T content[N];         // - vector storage
                Allocator<T,N>* next; // - pointer in freelist
             };
   public:   Allocator();             // default constructor
             Allocator(int n);        // ctor with preallocation
             T *allocate(void);       // "new"
             void deallocate(T* s);   // "delete"
             ~Allocator();
};
```

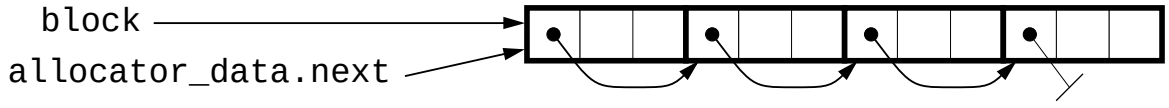
- Default Constructor: setup empty freelist

```
template <class T, int N>
inline Allocator<T,N>::Allocator() { next=0; }
```


- Constructor with Preallocation: allocate block of memory and setup freelist

```
template <class T, int N>
inline Allocator<T,N>::Allocator(int n) {
   Allocator<T,N>* block = new Allocator<T,N>[n];
   for (int i=0; i<n-1; ++i) block[i].next = &(block[i+1]);
   block[n-1].next = 0;
   next = block;
}
```

Example: `Allocator<double,3> vector<double,3>::allocator_data(4);`

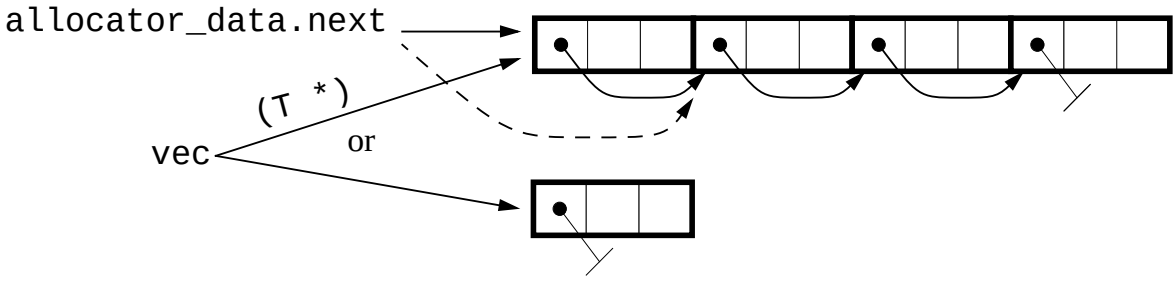


- Destructor

```
template <class T, int N>
inline Allocator<T,N>::~Allocator() {
   if (next) delete next;    // recursive !
}
```

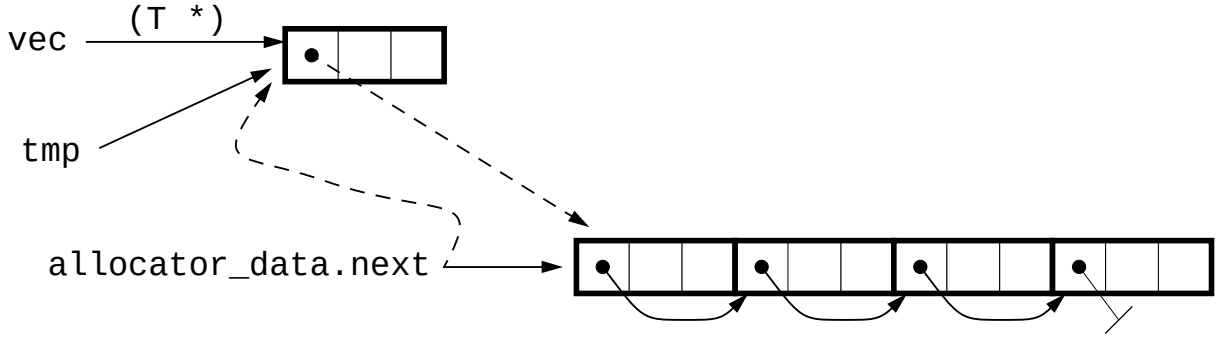
- Allocate Function ("new")

```
template <class T, int N>
inline T* Allocator<T,N>::allocate(void) {
   if (next) {
      T* vec = next->content;
      next = next->next;
      return vec;
   } else {
      Allocator<T,N>* tmp = new Allocator<T,N>;
      return tmp->content;
   }
}
```



- Deallocate Function ("delete")

```
template <class T, int N>
inline void Allocator<T,N>::deallocate(T* vec) {
   Allocator<T,N> *tmp = (Allocator<T,N>*) vec;
   tmp->next = next;
   next = tmp;
}
```



Problem: copy loops

➠ avoid unnecessary copying using *"copy-on-write"* idiom (implemented by *Reference Counting)*

❑ Implementation this time without using *handle/body class* idiom

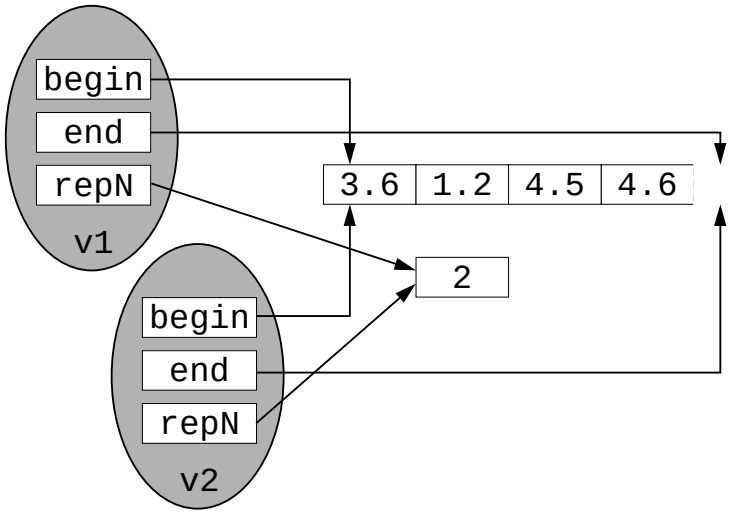
➠ each reference-counted object now has pointer to (shared) reference counter

➠ reference counter also managed by optimized memory managment

```
vector<double,4> v1, v2;
v1[0] = 3.6;
v1[1] = 1.2;

...

v2 = v1;
```



❑ Class `vector` definition

```
template <class T, int N>
class vector {

private:
   T *begin, *end;
   unsigned int *repN;       // reference counter pointer (new)
   static Allocator<T,N>              allocator_data;
   static Allocator<unsigned int, 1> allocator_repN;     // (new)

public:
   vector();
   vector(const vector<T,N>& rhs);
   ~vector();
   T& operator[](int n);

   // ...
};
```

- Constructor now also allocates reference counter and sets it to `1`

```
template <class T, int N>
vector<T, N>::vector(void) : begin(allocator_data.allocate()),
                 end(begin+N), repN(allocator_repN.allocate()) {
  *repN = 1;
}
```

- Copy constructor: shallow copy plus increment of the counter

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
  : begin(rhs.begin), end(rhs.end), repN(rhs.repN) {
  (*repN)++;
}
```

- Destructor only destructs if reference counter goes down to zero

```
template <class T, int N> vector<T, N>::~vector() {
   if (!(--(*repN))) {
      allocator_repN.deallocate(repN);
      allocator_data.deallocate(begin); }
}
```


- Assignment operator: shallow copy plus increment of the counter

```
template <class T, int N>
vector<T, N>& vector<T, N>::operator=(const vector<T, N>& rhs) {
   // self-test
   if (this == &rhs) return *this;

   // destroy lhs if no longer referenced
   if (!(--(*repN))) {
      allocator_repN.deallocate(repN);
      allocator_data.deallocate(begin);
   }

   // increment counter
   (*rhs.repN)++;

   // shallow copy of members
   repN = rhs.repN; begin = rhs.begin; end = rhs.end;
   return *this;
}
```

- Index operator: if vector is referenced more than once, make copy

```
template <class T, int N>
T& vector<T, N>::operator[](int n) {
   if ((*repN) == 1)
      return begin[n];
   else {
      (*repN)--;                          // count down old
      repN = allocator_repN.allocate(); // create new ref counter
      *repN = 1;

      // create new data space and copy over
      T* temp = allocator_data.allocate();
      memcpy((void*) temp, (void*) begin, size_t(N*sizeof(T)) );
      begin = temp;
      end   = begin+N;
      return begin[n];
   }
}
```

➠ reference counter especially effective for large arrays

- *Temporary Base Class Idiom* = "Reference Counting without counting"
- **Idea:** shallow/deep copy flag encoded in types
- temporary vectors are represented by class `Tvector`

```
template <class T, int N>
class Tvector {
private:
   T *begin, *end;
   static Allocator<T,N> allocator_data;

public:
```

- "Standard" Constructor

```
   Tvector() : begin(allocator_data.allocate()), end(begin+N) {}
```

- Copy constructor does "shallow" copy only

```
   Tvector(const Tvector<T,N>& rhs)
      : begin(rhs.begin), end(rhs.end) {}
```

- Extra constructor for `vector` copy constructor

```
   Tvector(T* b) : begin(b), end(b+N) {}
```

- Destructor empty as destruction is all handled by class `vector` (`virtual` because `vector` will be derived from `Tvector`)

```
virtual ~Tvector() {}
```

- "Standard" mathematical operators

```
Tvector<T, N> operator+(const Tvector<T, N>& rhs);
// ...
```

- Optimization (see below)

```
Tvector<T, N> operator+(const vector<T, N>& rhs);
```

- We also need to declare class `vector` a friend:

```
friend class vector<T, N>;
```

If your compiler doesn't yet support friends of templates, we need to use the following "hack"

```
//protected:  T* start() const { return begin; }
};
```


- `vector` class now subclass of `Tvector`

```
template <class T, int N>
class vector : public Tvector<T,N> {
public:
```

- Default constructor empty as construction is done by `Tvector()`

```
vector() {}
```

- Deallocate here, as `Tvector` doesn't delete anything!

```
~vector() { allocator_data.deallocate(begin); }
```

- Copy constructor (see below)

```
vector(const vector<T,N>& rhs);
```

- Assignment and Indexing handled here

```
T& operator[](int n) { return begin[n]; }
vector<T,N>& operator=(const vector<T,N>& rhs);
vector<T,N>& operator=(const Tvector<T,N>& rhs);
Tvector<T,N> operator+(const vector<T,N>& rhs);
};
```

- Copy constructor uses special `Tvector` constructor and does deep copy

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
   : Tvector<T,N>(allocator_data.allocate()) {
   T* dest = begin; T* src  = rhs.begin;
   while (dest != end) *(dest++) = *(src++);
}
```

- temporary variables as well as return type of mathematical operators are now of type `Tvector`

```
template <class T, int N> inline Tvector<T,N>
vector<T,N>::operator+ (const vector<T,N>& rhs) {
   Tvector<T,N> t;
   T* sum1 = begin;
   T* sum2 = rhs.begin;
   T* dest = t.begin;        // can use t.start() if
                             // friend templates don't work
   while (sum1 != end) *(dest++) = *(sum1++) + *(sum2++);
   return t;
}
```


- special assignment operator turn a `Tvector` into a `vector` and handles "`delete`"

```
template <class T, int N>
inline vector<T,N>&
vector<T,N>::operator= (const Tvector<T,N>& rhs) {
   allocator_data.deallocate(begin);
   begin = rhs.begin; // or if necessary: begin = rhs.start();
   end   = rhs.end;   //                  end   = begin+N;
}
```

- special optimization for mathematical `Tvector` operations if `this` is already temporary vector (e.g. `(v1+v2)` if `u = v1 + v2 + v3`)

```
template <class T, int N>
inline Tvector<T,N>
Tvector<T,N>::operator+ (const vector<T,N>& rhs) {
   T* sum1 = begin;
   T* sum2 = rhs.begin;
   while (sum1 != end) *(sum1++) += *(sum2++);
   return *this;
}
```

- *Chained Expression Objects*
  - Overloaded operators don't perform the operation but build an expression stack at runtime
  - Assignment operator matches expression stack with library of tuned expression kernels

    `x = v + w;` will execute as
    ```
    remember expr. "vector + vector"
    execute "add(x,v,w)"
    ```

- *Expression templates*
  - Avoid temporary objects in the first place by automatically transform

    ```
    u = v + w;        // vector u, v, w;
    ```

    at compile time (more-or-less) into

    ```
    for (int i=0; i<u.length(); ++i) {
       u[i] = v[i] + w[i];
    }
    ```

    using *Template Meta-Programming* (or *"Compile-Time Programs"*)
  - See Blitz++ (`http://oonumerics.org/blitz/`) and PETE (`http://www.acl.lanl.gov/pete/`)

Programming in C++

☆☆☆ The C++ Standard Library and Generic Programming ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

- Namespaces
  - provide a way to partition the global scope
  - a namespace is NOT a module; but it supports techniques related to modularity

```
namespace Chrono {
   class Date {
      /* ... */
   };
   enum Month { Jan, Feb, Mar, Apr, May, Jun,
                Jul, Aug, Sep, Oct, Nov, Dec };
   bool leapyear(int y);
   Date next_weekday(Date d);
}
```

- A `namespace` may only be defined
  - at file scope
  - or nested directly in another `namespace`

- Names declared inside a namespace can be accessed from outside the namespace by using the fully qualified name

```
Chrono::Date d(15, Chrono::May, 1998);
```

  - ➠ allows mix'n'match between components from different libraries

```
AnotherPackage::Date d2(1998, 5, 15);
```

- A namespace can be "unlocked" for the purpose of name lookup with a *using-directive*

```
using namespace Chrono;
Date d(15, May, 1998);
```

  - ➠ "if name lookup reaches file scope before the name has been found
    - ☆ search all unlocked namespaces
    - ☆ as well as file scope (which is considered a "special-name" namespace)"

- An individual name can be imported into the current scope with a *using-declaration*

```
using Chrono::Date;
Date d(15, Chrono::May, 1998);
```


The new C++ standard has now an extensive standard library

- Support for the standard C library (but slightly changed to fit C++'s stricter type checking)
- Support for strings (`string`)
- Support for localization and internationalization
- Support for I/O ➠ standardized I/O streams
- Support for numeric applications
  - complex numbers: `complex`
  - special array classes: `valarray`
- Support for general-purpose containers and algorithms
  - Standard Template Library (STL)

The name *Standard Template Library* is not particularly descriptive because almost everything in the C++ *Standard Library* is a template

- Standard `string` class is actually

```
template < class charT,
           class traits = string_char_traits<charT>,
           class Allocator = allocator>
class basic_string;

typedef basic_string<char> string;
```

- Class `complex` is actually

```
template<class T> class complex;
class complex<float>;
class complex<double>;
class complex<long double>;
```

- Same is true for `valarray`, `ios`, `istream`, `ostream`, **...**
- ➠ Note: no longer possible to use a forward declaration of these types:

```
class ostream;                // does no longer work!
```


- Standrad C++ library headers use a "new" style (note the missing `.h`!):

```
<iostream>      <string>     <vector>
<fstream>       <complex>    <list>       ...
```

  - ➠ Do not mix pre-standard old headers and new style headers! (e.g., `<iostream.h>` and `<vector>`)

- New C++ headers for C library facilities (not available yet for many compilers)

```
<cassert>   <ciso646>   <csetjmp>   <cstdio>   <ctime>
<cctype>    <climits>   <csignal>   <cstdlib>  <cwchar>
<cerrno>    <clocale>   <cstdarg>   <cstring>  <cwctype>
<cfloat>    <cmath>     <cstddef>
```

- All C++ standard library objects (including the C library interface) are now in namespace `std` !!!
- Compiler vendors often supply migration headers:

  e.g. `<vector.h>`:

```
#include <vector>
using std::vector;
```

- Generic Programming is NOT about object-oriented programming
- Although first research papers appeared 1975, first experimental generic software was not implemented before 1989
  - ➠ not many textbooks on generic programming exist
- First larger example of generic software to become important outside research groups was

  STL (“*Standard Template Library*”)
  - designed by Alexander Stepanov and Meng Lee of HP
  - added to C++ standard in 1994
  - available as public domain software first from HP and now from SGI

- New programming techniques were always based on new *abstractions*
  - often used sequences of instructions ➠ *subroutines*
  - data + interface ➠ *abstract data types*
  - hierarchies of polymorphic ADT’s ➠ *inheritance*
- For *generic programming*:
  - set of requirements on data types ➠ *concept*
- Generic algorithm has
  - generic instructions describing the steps of the algorithm
  - set of requirements specifying the properties the algorithm arguments must satisfiy

➠ only first part can be expressed in C++

➠ templates

- STL is based on four fundamental concepts
  - *containers* hold collections of objects
    - `bitset`, `vector`, `list`, `deque`, `queue`, `stack`, `set`, `map`, ...
  - *iterators* are pointer-like objects to walk through STL containers
  - *algorithms* are functions that work on STL containers
    - `find`, `copy`, `count`, `fill`, `remove`, `search`, ...
  - *functors* are function objects used by the algorithms

- To understand these concepts, consider a function to find a value in an `int` array:

```
int find(int array[], int len, int value) {
   int idx;
   for (idx=0; idx<len; ++idx) if (array[idx]==value) return idx;
   return -1;
}
```

  - can be improved by using pointers to specify begin of search, end of search, and result
  - also allows for search in subarrays

- To understand the basic principle of STL, look at the rules of C++ (and C) arrays and pointers:
  - a pointer to an array can legitimately point to any element of the array
  - or it can point to one element beyond the end of the array (but then it can only be compared to other pointers to the array and it cannot be dereferenced)

  and rewrite `find` to search in the *range* [`begin`, `end`):

```
int* find(int* begin, int* end, int value) {
   while (begin != end && *begin != value) ++begin;
   return begin;   // begin==end if not found
}
```

- You could use `find` function like this:

```
int values[50];
int *firstFive = find(values, values+50, 5);

if (firstFive!=values+50) { //5 found...} else { //not found...}
```

- You can also use `find` to search subranges of the array:

```
int *five = find(values+10, values+20, 5);
```

- Nothing inherent in the `find` function that limits it to array of `int`, so should be `template`:

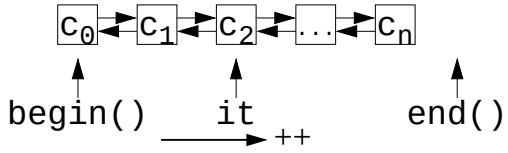
```
template<class T>
T* find(T* begin, T* end, const T& value) {
   while (begin != end && *begin != value) ++begin;
   return begin;
}
```

- Nice, but still to limiting: look at the operations on `begin` and `end`
  - inequality operator, dereferencing, prefix increment, copying (for result!)
  - Why restrict `begin` and `end` to pointers?
  - ➠ allow any object which supports the operations above: *iterators*

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value) {
   while (begin != end && *begin != value) ++begin;
   return begin;
}
```

- ➠ This version is the actual implementation of `find` in the STL!

- We can apply `find` to every STL container, e.g. `list`




```
list<char> cList;

// fill cList with values...

list<char>::iterator it = find(cList.begin(), cList.end(), 'x');
if (it != cList.end()) { // found 'x' ... }
```

  - ➠ `begin()` and `end()` are STL container member functions which return iterators pointing to the beginning and end of the container

- Furthermore, C++ pointers *are* STL iterators, so e.g.,

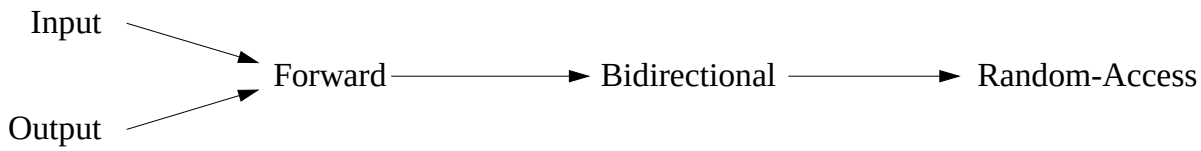
```
int values[50];

// fill values with actual values ...

int *firstFive = find(values, values+50, 5);  // calls STL find
if (firstFive != values+50) { // found 5 ... }
```

- A set of requirements (e.g., like for *iterator*) is called a *concept*
- A *concept* can be defined in two (equivalent) ways
  - list of type requirements
  - a set of types
- A *type* `T` is a *model* of a *concept* `C` if
  - `T` satisfies all of `C`'s type requirements
  - `T` belongs to the defining set of types
- *Concept requirements* can**not** fully be described as needed set of member functions (➠ classes)
  - ➠ e.g., `char*` is model of concept iterator but has no member functions
  - ➠ but can be seen as *list of valid expressions*

- Iterator concepts: categorize iterators based on their functionality



Functionality needed:

```
++i  *i  i=j  i==j  i!=j          --i          i[]  i+n  i-n
                                                 i-j  i<j
```

Input iterator `I`: `*I` may only be read once
Output iterator `O`: `*O` may only be written once
Forward iterator `F`: `*F` may be read or written multiple times

- Relation `C1` ⟶ `C2` is called a *refinement* if concept `C2` satisfies all of `C1` requirements and possibly additional requirements
  - ➠ all *models* of `C2` are also models of `C1`
- *Modeling* is relationship between a type and a concept
  Refinement is relationship between two concepts
  (*inheritance* is relationship between two types)

- **STL is just a collection of class and function templates that adhere to a set of conventions**
  - Basic idea behind STL is **simple**
  - STL is **extensible**: you can add your own collections, algorithms, or iterators as long you follow the STL conventions
  - STL is **efficient**: there is no big inheritance hierarchy, no virtual functions, ...
    - STL: *C* containers + *A* algorithms
    - Traditional Library: *T* types × *C* × *A*
  - STL is **portable**

- Disadvantages
  - no error checking (use "safe-STL"!)
  - unusual programming interface

- Using traditional libraries is better than using no library!
  e.g., RogueWave's `tools.h++` (Cray: CCtoollib), `math.h++`, `lapack.h++` (CCmathlib)

- Containers store *collections* of objects (or pointers to objects)
- All STL containers have a standardized interface. Apart from minor differences they provide the same constructors, member functions, and public types
- STL containers are grouped into four categories
  - *sequence* containers
    - linear unsorted collections (`vector`, `deque`, `list`)
    - insert position depends on time / place not value
  - *sorted associative* containers
    - rapid insertion and deletion based on keys (`set`, `map`, `multiset`, `multimap`)
    - insert position depends on value
  - *"almost"* containers (`string`, built-in arrays, `bitset`, `valarray`)
  - container *adapters* (`stack`, `queue`, `priority_queue`)
- There are no *hashed* containers in the C++ Standard STL
  - public domain SGI STL provides them as an extension

- Every STL container declares at least these public types as nested `typedef`s:

Typename	Description
`value_type`	Type of element stored in container (typically `T`)
`size_type`	Signed type for subscripts, element counts, ...
`difference_type`	Unsigned type of distances between iterators
`iterator`	Type of iterator (behaves like `value_type*`)
`const_iterator`	Type of iterator for constant collections (behaves like `const value_type*`)
`reverse_iterator`	For traversing container in reverse order
`const_reverse_iterator`	Same for constant collections
`reference`	Type of reference to element (behaves like `value_type&`)
`const_reference`	Behaves like `const value_type&`

Associative containers provide also:

`key_type`	Type of key
`mapped_type`	Type of mapped value
`key_compare`	Type of comparison criterion


```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<float> x(5);

   // Get iterators positioned at first and behind "last" element
   vector<float>::iterator first=x.begin(), last=x.end();

   // Get reference to first element and change it
   vector<float>::reference z = x[0];
   z = 8.0;
   cout << "x[0] = " << x[0] << endl;

   // Find the size of the collections
   vector<float>::size_type size=last - first;
   cout << "Size is " << size << endl;
}
```

- Output:

```
x[0] = 8
Size is 5
```

- Every container `Container` provides the following constructors:
  - Default constructor to create empty container

    ```
    Container();
    ```

  - Copy constructor: initialize elements from container of same type

    ```
    Container(const Container&);
    ```

  - Initialize container from container of other type through iterators
    - ➠ element types must be assignment compatible

    ```
    template<class Iterator>
    Container(Iterator first, Iterator last);
    ```

  - Destructor: destroy container and all of its elements

    ```
    ~Container();
    ```

- Examples

```
vector<int> a;                          // calls default ctor
vector<int> b(a);                       // calls copy ctor
list<double> c(a.begin(), a.end());  // calls ctor(iter,iter)
```


- Every STL container provides the following member functions to obtain iterators:

Method	Points to	Associated type
`begin()`	first element	`Container::iterator`
`end()`	one-past-last element	
`rbegin()`	first element of reverse sequence	`Container::reverse_iterator`
`rend()`	one-past-last element of reverse sequence	

- For `const` containers, iterator type is `const_iterator` or `const_reverse_iterator`
- `vector`, `deque`, `string`, `valarray`, and built-in arrays have *random-access* iterators
- `list`, `map`, `multimap`, `set`, and `multiset` have *bidirectional* iterators
- Example: output all values of a vector

```
int data[5] = { 23, 42, 666, -89, 5 };
vector<int> x(data, data+5);
for(vector<int>::iterator it=x.begin(); it!=x.end(); ++it)
   cout << *it << endl;
```

not: `it<x.end()`

➠ would require random access!

note: `++it` more efficfient than `it++`

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main() {
   vector<const char *> mathies(3);
   mathies[0] = "Alan Turing";
   mathies[1] = "Paul Erdos";
   mathies[2] = "Emmy Noether";

   list<const char *> mathies2(mathies.begin(), mathies.end());

   for(list<const char *>::iterator it=mathies2.begin();
                                    it!=mathies2.end(); ++it)
      cout << *it << endl;
}
```

- Output:

```
Alan Turing
Paul Erdos
Emmy Noether
```


- STL containers can be compared using the usual comparison operators
- Equality operator `operator==()`
  - ➠ returns `true` if the containers are the same size
  - ➠ **and** the corresponding elements of each container are equal

  Container elements are compared using `T::operator==()`
- Inequality is defined accordingly to determine if two containers are different
- The other comparison operators
  - `operator<()`
  - `operator<=()`
  - `operator>()`
  - `operator>=()`
  - ➠ return `true` if first container is *lexicographically* less/less or equal/greater/greater or equal than second container
  - ➠ Containers are compared element by element; first element which differs determines which container is less or greater

```
Container a, b;
// fill a and b ...
```

- Assign the contents of one container to the other

```
a = b;
```

- Exchange the contents of the two containers

```
a.swap(b);
```

- Get the current number of elements stored in container

```
Container::size_type n = a.size();
```

- Get the size of the largest possible container

```
Container::size_type maxn = a.max_size();
```

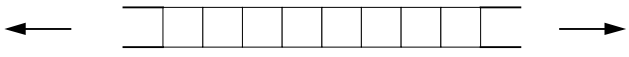
- Check whether container is empty (has no elements)

```
bool isEmpty = a.empty();
```

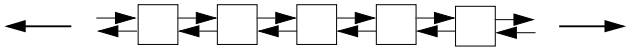

STL Sequence Containers — Overview

- *Sequence* containers provide
  - linear (➠ elements have a predecessor and successor), unsorted collections of elements
  - insert position depends on time / place not value
- STL provides three sequence containers:
  - `vector<T>`

    - ➠ dynamic array
    - ➠ fast random access, fast insert and delete *at the end*
  - `deque<T>`



    - ➠ "double-ended queue"
    - ➠ fast random access, fast insert and delete *at both the beginning and end*
  - `list<T>`



    - ➠ doubly-linked list
    - ➠ *slow* random access, fast insert and delete *anywhere*

- In addition to the general constructors for all STL containers
  - `Container();`
  - `Container(const Container&);`
  - ```
    template<class Iterator>
    Container(Iterator first, Iterator last);
    ```

  sequence containers `SeqContainer` also provide
  - Create container with `n` default values

    `SeqContainer(size_type n);`
  - Create container with `n` copies of an element `x`

    `SeqContainer(size_type n, T x);`

- Examples

```
vector<int> x(5);           // => { 0, 0, 0, 0, 0 }
list<double> y(3, 3.14);    // => { 3.14, 3.14, 3.14 }
```



```
SeqContainer a;               // Fill a ...
SeqContainer::iterator it, r; // Point somewhere into a
SeqContainer::iterator f, l;  // Two more iterators
SeqContainer::value_type val; // Some value to insert
```

- Insert element `val` right before the iterator `it`

```
r = a.insert(it, val);        // r points to inserted val
```

- Insert `n` copies of element `val` right before the iterator `it`

```
Container::size_type n;
a.insert(it, n, val);         // void
```

- Inserts elements described by iterator range [`f`, `l`) right before the iterator `it`

```
a.insert(it, f, l);           // void
```

- Remove element pointed to by iterator `it` or range of elements described by [`f`, `l`)

```
r = a.erase(it);              // r = ++it
r = a.erase(f, l);            // r = l
```

- Remove all elements

```
a.clear();                    // == a.erase(a.begin(), a.end());
```
```

STL Sequence Containers — Assignment

- *Sequence* containers also provide overloaded forms of a member function `assign`
- `x.assign(...)` has the same effects as `x.clear(); x.insert(x.begin(), ...)` but more efficient
- Assign `n` default values to container

```
assign(size_type n);
```

- Assign `n` copies of an element `x` to container

```
assign(size_type n, T x);
```

- Assign elements from container of other type described by iterators to container
  - ➠ element types must be assignment compatible

```
template<class Iterator>
assign(Iterator first, Iterator last);
```

- Example

```
vector<int> x;
x.assign(3, 42);            // => { 42, 42, 42 }
```


STL Sequence Containers — Optional Operations

- These operations are provided only for the sequence containers for which they take constant time:

Expression	Semantics	Container
Getting the first or last element:		
`a.front()`	`*a.begin()`	`vector, list, deque`
`a.back()`	`*--a.end()`	`vector, list, deque`
Adding (`push`) or deleting (`pop`) elements at the beginning or end:		
`a.push_front(x)`	`a.insert(a.begin(),x)`	`list, deque`
`a.push_back(x)`	`a.insert(a.end(),x)`	`vector, list, deque`
`a.pop_front()`	`a.erase(a.begin())`	`list, deque`
`a.pop_back()`	`a.erase(--a.end())`	`vector, list, deque`
Random access to elements:		
`a[n]`	`*(a.begin() + n)`	`vector, deque`
`a.at(n)`	`*(a.begin() + n)`	`vector, deque`

- `at()` provides *bounds-checked* access ➠ throws `out_of_range` if `n >= a.size()`

- Sequence container `vector<T>` provides fast random access to elements
- `vector<T>` can be seen as C style array with a C++ wrapper
  - but will automatically resize itself when necessary
  - adheres to the STL conventions (i.e., defines public types, iterators, ...)
- Usage:

```
#include <vector>
using std::vector;

vector<float> x(10);
```

- `vector<T>` provides *random-access* iterators
  - ➠ **all** STL algorithms may be applied to them

- `vector<T>` also replacement for built-in (especially) dynamic arrays

```
vector<int> x(len);          //instead of:  int *x = new int[len];
                                            ...; delete[] x;

for (int i=0; i<x.size(); ++i) {
   x[i] = 1 + int(6.0*rand()/(RAND_MAX+1.0));  //throw dice
}
```

- Advantages:
  - Size does not have to be constant
  - `vector<T>` knows its size
  - STL compliant, e.g., STL algorithms and iterator access works too:

```
for (vector<int>::iterator it=x.begin(), it!=x.end(); ++it) {
   cout << " " << *it;
}
```

  - can dynamically increase size (automatically with `insert()` or `push_back()`, not `[]`!!)
  - `vector<T>` objects can be assigned
  - bounds checking possible using `at()`

STL deque<T> — Basics

- Sequence container `deque<T>`
  - provides fast random access to elements very much like `vector<T>`
  - can be seen as a vector which can grow dynamically on both ends
  - ➠ Unlike `vector<T>`, `deque<T>` supports also constant time insert and erase operations at the beginning **or** end
- As with `vector<T>`, memory allocation and resizing is handled automatically **but** `deque<T>` does **not** provide methods `capacity()` and `reserve()`
- Usage:

```
#include <deque>
using std::deque;

deque<float> x(10);
```

- `deque<T>` provides *random-access* iterators
  - ➠ **all** STL algorithms may be applied to them

STL list<T> — Basics

- Sequence container `list<T>`
  - provides constant time insert and erase anywhere in the container
    - ➠ unlike `vector<T>` or `deque<T>`, insert or erase operations never invalidate pointers or iterators
  - has automatic storage management
  - does **not** support random access
- `list<T>` is typically implemented as a doubly-linked list
  - ➠ additional storage overhead *per node* (at least two pointers)
- Usage:

```
#include <list>
using std::list;

list<float> x(10);
```

- `list<T>` provides *bidirectional* iterators
  - ➠ **not** all STL algorithms may be applied to them (e.g. `sort()`!)

```
#include <iostream>
#include <list>
using namespace std;

int main() {
   list<const char *> mammals;
   mammals.push_back("cat");
   mammals.push_back("dog");

   list<const char *>::iterator it = mammals.begin();
   ++it;        // 'it' is now at "dog"

   mammals.insert(it, "cow");           // 'it' is still at "dog"

   const char *others[] = { "horse", "pig", "rabbit" };
   mammals.insert(it, others, others+3);// 'it' is still at "dog"

   for(it=mammals.begin(); it!=mammals.end(); ++it)
      cout << " " << *it;
}
```

- Output:

```
cat cow horse pig rabbit dog
```


- STL generic sorting algorithms require *random-access* iterators (which `list<T>` doesn't have)
  - ➠ `list<T>` provides its own efficient `sort()` member function
- Example

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main() {
   const char *places[] = { "Paris", "Rom", "London", "Juelich" };

   list<string> placeList(places, places+4);
   placeList.sort();

   for(list<string>::iterator it=placeList.begin();
                              it!=placeList.end(); ++it)
      cout << " " << *it;
}
```

- Output:

```
Juelich London Paris Rom
```

- What sequence container should be used?
  - Use `vector<T>` when
    - fast, random-access is needed to the contents
    - insertions and deletions usually occur at the end of the sequence
    - size of the collection is reasonably constant
      or occasional resizing can be tolerated
      or size is known in advance
  - If frequent insertions and deletions at the beginning of the sequence are also required, consider using `deque<T>`
  - If frequent insertions and deletions are necessary everywhere in the sequence, use `list<T>`

- STL provides *adaptors* which turn *sequence* containers into containers with *restricted interface*
  - In particular, no iterators provided; can only be used through the specialized interface
  - No "real" STL containers (STL algorithms cannot be used with the adapted containers)

  There are `stack<T>`, `queue<T>`, and `priority_queue<T>`

STL Container Adaptors `stack<T>`

```
template <class T, class Container = deque<T> >  class stack {
public: typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;

        explicit stack(const Container& = Container());
        bool empty() const;
        size_type size() const;
        value_type& top();              // get top element
        const value_type& top() const;
        void push(const value_type& x);  // add element on top
        void pop();                      // remove top element
};
```

- `Container` can be any STL container with `empty()`, `size()`, `back()`, `pop_back()`, and `push_back()`
- Example:

```
#include <stack>
using std::stack;

stack<char> s1;                 // uses deque<char> for storage
stack<int, vector<int> > s2;    // uses vector<int>
```

STL Container Adaptors — queue<T>

```
template <class T, class Container = deque<T> >  class queue {
public: // the same typedefs as stack

        explicit queue(const Container& = Container());
        bool empty() const;
        size_type size() const;
        value_type& front();             // get front element
        const value_type& front() const;
        value_type& back();              // get back element
        const value_type& back() const;
        void push(const value_type& x);  // add element to back
        void pop();                      // remove front element
};
```

- `Container` can be any STL container with `empty()`, `size()`, `front()`, `back()`, `push_back()`, and `pop_front()` ➠ `vector` cannot be used!
- Example:

```
#include <queue>
using std::queue;

queue<char> q1;
```


STL Container Adaptors — priority_queue<T>

```
template <class T, class Ctr = vector<T>, class Cmp =
         less<typename Ctr::value_type> >  class priority_queue {
public:
   // the same typedefs as stack

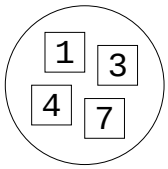
   explicit priority_queue(const Cmp&=Cmp(), const Ctr&=Ctr());
   bool empty() const;
   size_type size() const;
   const value_type& top() const;    // get "top" element
   void push(const value_type& x);   // add element
   void pop();                       // remove "top" element
};
```

- *"top"* element := element with *highest priority* ➠ *largest* element based on `Cmp`
- Container `Ctr` must provide *random-access* iterators ➠ `list` cannot be used!
- Example:

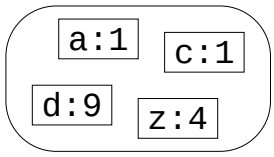
```
#include <queue>
using std::priority_queue;

priority_queue<char> pq1;
```

- *Associative* containers
  - store the elements based on *keys*
  - are sorted (based on keys)

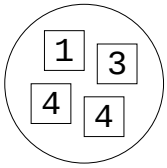


set<int>

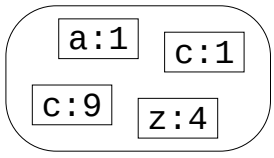


map<char,int>

- STL provides four associative containers:
  - `map<Key, T>`
    - ➠ collection of (key, value) pairs
    - ➠ fast retrieval based on keys
    - ➠ each key in map is *unique* ➠ one value per key
  - `set<T>`
    - ➠ `map` where key is value itself
  - `multimap<Key,T>` and `multiset<T>`
    - ➠ versions of `map` and `set` where keys must not be unique
    - ➠ more than one value per key



multiset<int>



multimap<char,int>

- maps called *associative array* or *table*, multimap *dictionary*, and multiset *bag* in other languages

- Associative containers are *sorted*
  - based on `key_type::operator<()` or
  - a user-specified predicate `Cmp`
    - ➠ `Cmp` must define *strict weak ordering* (`[1]`+`[2]` *is partial ordering*)

```
[1]  Cmp(x,x) is false                              // irreflective
[2]  If Cmp(x,y) and Cmp(y,z), then Cmp(x,z)        // transitive
[3]  If equiv(x,y) and equiv(y,z), then equiv(x,z)
```

- Associative containers have an optional extra template parameter specifying the sorting criterion ➠ default is `key_type::operator<()`
- Equality of keys is tested using `key_type::operator==()`
  - ➠ with user-specified sorting criterion `Cmp`

  `equiv(x,y)` is defined as `!(Cmp(x,y) || Cmp(y,x))`

- **Note**: `operator<()` for C style strings (`char *`) compares pointers not the contents!
  - ➠ use:

```
bool strLess(const char *p1, const char *p2)
                            { return strcmp(p1, p2) < 0; }
```

- The member functions return iterator `end()` to indicate "*not found*"
- There are five member functions for locating a key `k` in an associative container `ac`
  - Return iterator to element with key `k`

    ```
    iterator it = ac.find(k);
    ```
  - Find number of elements with key `k`

    ```
    size_type c = ac.count(k);
    ```
  - Find *first* element with key *greater equal* than `k`

    ```
    iterator it = ac.lower_bound(k);
    ```
  - Find *first* element with key *greater* than `k`

    ```
    iterator it = ac.upper_bound(k);
    ```
  - Get subsequence [`lower_bound`, `upper_bound`) at once

    ```
    pair<iterator,iterator> p = ac.equal_range(k);
    ```
- Of course, the last three member functions are more useful for `multiset` and `multimap`

STL `set<T>` and `multiset<T>` Basics

- Associative container `set<T>` keeps a *sorted* collection of *unique* values
- Associative container `multiset<T>` keeps a *sorted* collection of values
- ➠ for `set` and `multiset`, key is value itself!
- ➠ `T` == `value_type` == `key_type`
- Usage:

```
#include <set>
using std::set;

set<float> x;                          // sorted by <
set<int, greater<int> > y;             // sorted by >

std::multiset<double> z;
```

- `set<T>` and `multiset<T>` provide *bidirectional* iterators
  - ➠ **not** all STL algorithms may be applied to them

STL `set<T>` and `multiset<T>` Inserting and Deleting Elements

- The position of elements in an *associative* container are determined by the sorting criterion
  - insertion member functions do **not** have iterator parameter specifying position for insert

- Two more forms of `insert` are provided for a `set` or `multiset` container `SetContainer`
  - Insert value `val`

```
SetContainer c;
SetContainer::iterator it = c.insert(val);
```

  - Add elements from other container described by iterator range [`f`, `l`)

```
c.insert(f, l);                          // void
```

- For `set`, keys are only inserted if they do not already occur in the container.
  Also, instead of returning an iterator, `insert(val)` returns a `pair<iterator,bool>` where the second part is `false` when the value was already present and was not inserted.

- In addition to the usual ways of deleting elements (`erase(it)`, `erase(f,l)`, and `clear()`), associative containers also allow deletion based on keys (== values for sets):

```
c.erase(key);
```


STL `set<T>` Example

```
#include <iostream>
#include <set>
using namespace std;

int main() {
   set<int> x;

   x.insert(42);
   x.insert(496);
   x.insert(6);
   x.insert(42);
   x.insert(-5);

   for(set<int>::iterator it=x.begin(); it!=x.end(); ++it)
      cout << " " << *it;
   cout << endl;
}
```

- Output:

```
 -5 6 42 496
```

STL `multiset<T>` Example

```
#include <iostream>
#include <set>
using namespace std;

int main() {
   multiset<int> x;

   x.insert(42);
   x.insert(496);
   x.insert(6);
   x.insert(42);
   x.insert(-5);

   for(multiset<int>::iterator it=x.begin(); it!=x.end(); ++it)
      cout << " " << *it;
   cout << endl;
}
```

- Output:

```
 -5 6 42 42 496
```


STL `map<Key,T>` and `multimap<key,T>` Basics

- Associative container `map<Key,T>` keeps a *sorted* collection of *unique* (key, value) *pairs*
- Associative container `multimap<Key,T>` keeps a *sorted* collection of (key, value) *pairs*
- `map` and `multimap` have the same forms of `insert` (and `erase`) like `set`s, **but** their `value_type` is defined as `pair<const Key,T>`
  - `insert` member functions take parameter of type `pair<const Key,T>`
  - iterators point to `pair<const Key,T>`

  (`const Key` because changing keys could destroy sorting)
- Usage:

```
#include <map>
using std::map;

map<int,int> m;                          // int -> int
std::multimap<char*,int,strLess> mm;     // char* -> int
```

- `map<Key,T>` and `multimap<Key,T>` provide *bidirectional* iterators
  - **not** all STL algorithms may be applied to them

STL `pair<T1,T2>`

- STL provides small utility class `pair<T1,T2>` which essentially looks like this:

```
template<class T1, class T2> struct pair {
   T1 first;
   T2 second;
   pair(const T1& f, const T2& s) : first(f), second(s) {}
};
```

- Example of `map insert`

```
map<string,float> numbers;
numbers.insert(pair<string,float>("Pi", 3.141592653589));
numbers.insert(pair<string,float>("e",  2.718281828459));
```

- To make `pair`s slightly less ugly, STL provides `make_pair()` function:

```
numbers.insert(make_pair("Pi", 3.141592653589));
```

- In addition, `map` (but not `multimap`) provides an overloaded `operator[]`:

```
numbers["Pi"] = 3.141592653589;
```

  - lookup key and return reference to value; if not found, *insert* `pair(key,T())`!
  - For lookup without modifying map use `find()` or `count()`!

STL `map<Key,T>` Example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
   map<string,double> prices;

   prices["sugar"] = 2.99;
   prices["salt"]  = 1.87;
   prices["flour"] = 2.49;

   cout << "Sugar costs " << prices["sugar"] << endl;

   map<string,double>::iterator h = prices.find("salt");
   cout << "Salt costs " << h->second << endl;
}
```

- Output:

```
Sugar costs 2.99
Salt costs 1.87
```

STL `multimap<Key,T>` Example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
typedef pair<string,string> Pair;
typedef multimap<string,string> strMap;
typedef pair<strMap::iterator, strMap::iterator> iterPair;

int main() {
   strMap phonebook;
   phonebook.insert(Pair("Hans", "0123 / 454545"));
   phonebook.insert(Pair("Lisa", "0999 / 12345"));
   phonebook.insert(Pair("Moni", "0180 / 999999"));
   phonebook.insert(Pair("Hans", "0771 / 16528"));

   iterPair ip = phonebook.equal_range("Hans");
   for(strMap::iterator it=ip.first; it!=ip.second; ++it)
      cout << "Hans's phone number is " << it->second << endl;
}
```

- Output:
```
Hans's phone number is 0123 / 454545
Hans's phone number is 0771 / 16528
```


STL Algorithms

- STL provides a large assortment of algorithms which can be applied to *any* data structure which adheres to the STL conventions (especially provides iterators)
- Defined in the header file `<algorithm>`
- STL algorithms can be grouped into four categories:
  - *Nonmutating sequence algorithms* operate on containers *without* making changes to their contents (e.g., `find`)
    - ➠ can also be applied to `const` container
  - *Mutating sequence algorithms* operate on containers *changing* their contents (e.g., `fill`). Often, mutating algorithms come in two versions *`foo`*`()` and *`foo`*`_copy()`:
    - ☆ *In-place version*: replaces original contents of container with results
    - ☆ *Copying version*: places the result in a new container
  - *Sorting-related algorithms* sort containers or operate only on *sorted* containers
    - ➠ `set<T>`, `map<T>`, sorted `vector<T>`s, ...
  - *Generalized numeric algorithms* defined in header `<numeric>`

- Some STL algorithms have a version which takes an extra function parameter.
- For example, one version of `sort()` has the following prototype:

```
template<class RandomIter, class Compare>
void sort(RandomIter first, RandomIter last, Compare cmp);
```

`cmp` is a function parameter specifying the sorting criterion for sorting the range [`first`, `last`)

- For example, suppose we have a very simple class `Person`:

```
struct Person {
   Person(const string& first, const string& last) :
                              firstName(first), lastName(last) {}
   string firstName, lastName;
};
ostream& operator<<(ostream& ostr, const vector<Person>& folks) {
   vector<Person>::const_iterator it;
   for (it=folks.begin(); it!=folks.end(); ++it)
      ostr << it->firstName << " " << it->lastName << endl;
   return ostr;
}
```


- We now want to be able to sort a vector of `Person` according to either the first or last name
- Function parameters can be expressed in two ways:
  - as *pointer to functions*:

```
bool compareFirstName(const Person& p1, const Person& p2) {
   return p1.firstName < p2.firstName;
}
```

  - as *function objects* or *functors* := objects which behave like functions
    - ➠ object with function call operator `operator()` defined

```
struct compareLastName {
   bool operator()(const Person& p1, const Person& p2) {
      return p1.lastName < p2.lastName;
   }
};
```

    - ➠ biggest advantage: function objects can have local *state*
    - ➠ also: more efficient because can be inlined (no indirect function calls!)
- Non-modifying *Functions* or *functors* which return `bool` are also called *predicates*

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() {
   vector<Person> folks;

   folks.push_back(Person("Hans", "Lustig"));
   folks.push_back(Person("Lisa", "Sommer"));
   folks.push_back(Person("Moni", "Hauser"));
   folks.push_back(Person("Herbert", "Hauser"));

   // -- sort with function pointer
   sort(folks.begin(), folks.end(), compareFirstName);
   cout << "Sorted by first name:" << endl << folks << endl;

   // -- sort with functor; note the () to call default ctor to get an object!
   sort(folks.begin(), folks.end(), compareLastName());
   cout << "Sorted by last name:" << endl << folks << endl;
}
```


- To keep the following description of the STL algorithms compact
  - the `template<...>` specification was omitted
  - secondary forms of the same algorithm are not described in full detail (e.g., the `_copy` version or versions with an additional optional parameter)
  - the template parameter names are significant:

Name	Meaning
`In`	input iterator
`Out`	output iterator
`For`	forward iterator
`Bi`	bidirectional iterator
`Ran`	random-access iterator
`Op, BinOp`	unary and binary operation (function or functor)
`Pred, BinPred`	unary and binary predicate
`Cmp`	sorting criterion with strict weak ordering

Linear Search

- Find first element equal to `value`; return `last` when not found

```
In find(In first, In last, const T& value);
```

- Find first element for which unary predicate `pred` is true

```
In find_if(In first, In last, Pred pred);
```

- Example: use `find_if` to find all persons whose last name matches a string (using `string::find()`!)
  - define predicate functor

```
struct lastNameMatch {
   lastNameMatch(const string& pattern) : patt(pattern) {}

   bool operator()(const Person& p) {
      return p.lastName.find(patt) != string::npos;
   }

private:
   string patt;
};
```



```
int main() {
   vector<Person> folks;

   folks.push_back(Person("Hans", "Lustig"));
   folks.push_back(Person("Lisa", "Sommer"));
   folks.push_back(Person("Moni", "Hauser"));
   folks.push_back(Person("Herbert", "Hauser"));

   vector<Person>::iterator it = folks.begin();

   while (true) {
      it = find_if(it, folks.end(), lastNameMatch("use"));
      if (it == folks.end()) break;
      cout << it->firstName << " " << it->lastName << endl;
      ++it;
   }
}
```

- Output:

```
Moni Hauser
Herbert Hauser
```

Linear Search (cont.)

- Find first element out of any values in [first2, last2) in the range [first1, last1)

```
For find_first_of(For first1, For last1, For first2, For last2);
```

- Find first consecutive duplicate element

```
For adjacent_find(For first1, For last1);
```

- Find *first* occurrence of subsequence [first2, last2) in [first1, last1)

```
For search(For first1, For last1, For first2, For last2);
```

- Find *last* occurrence of subsequence [first2, last2) in [first1, last1)

```
For find_end(For first1, For last1, For first2, For last2);
```

- Find first occurrence of subsequence of count consecutive copies of value

```
For search_n(For first1, For last1, Size count, const T& value);
```

- All return last1 for indicating not found
- These five algorithms are also available with an additional parameter pred specifying a binary predicate to be used instead of operator==() to check for equality

Counting Elements

- Count number of elements equal to value

```
difference_type count(In first, In last, const T& value);
```

- Count number of elements that satisfy predicate pred

```
difference_type count_if(In first, In last, Pred pred);
```

Minimum and Maximum

- Return the minimum or maximum of two elements

```
const T& min(const T& a, const T& b);
const T& max(const T& a, const T& b);
```

- Find smallest and largest element

```
For min_element(For first, For last);
For max_element(For first, For last);
```

- These four functions are also available with an additional parameter cmp specifying a sorting criterion to be used instead of operator<() for comparing elements

Comparing Two Ranges

- Is range1 equal to range2 (i.e., same elements in same order)?

```
bool equal(In first1, In last1, In first2);
```

- Find first position in each range where the ranges differ

```
pair<InIt1, InIt2> mismatch(In first1, In last1, In first2);
```

- ➠ Second range starting at `first2` must be at least as long as range1 [`first1`, `last1`)
- ➠ Both also available with additional parameter specifying binary predicate `pred`
- Is range1 lexicographically less than range2?

```
bool lexicographical_compare(In first1, In last1,
                             In first2, In last2);
```

- ➠ Also available with additional parameter specifying sorting criterion `cmp`

for_each

- Apply unary function or function object `f` to each element in the sequence
  Return `f` (useful for functors which keep / calculate state information)

```
Op for_each(In first, In last, Op f);
```


Copying Ranges

- Copy range1 [`first`, `last`) to range2 starting at `result`

```
Out copy(In first, In last, Out result);
```

- Copy range1 [`first`, `last`) to range2 ending at `result` backwards
  - ➠ use this version if range1 and range2 *overlap* (start of range2 is between `first` and `last`)

```
Bi copy_backward(Bi first, Bi last, Bi result);
```

- ➠ The result range must have enough elements to store result!
  (the elements are *assigned* the result values, they are not *inserted*)

- Example:

```
block<int,6> x = { 1, 2, 3 4, 5, 6 };
vector<int> not_ok, ok(x.size());

copy(x.begin(), x.end(), not_ok.begin());     // undefined!
copy(x.begin(), x.end(), ok.begin());         // OK!
```

Transforming Elements

- Apply unary operation `op()` to elements in range [`first`, `last`) and write to `res`

```
Out transform(In first, In last, Out res, Op op);
```

- Apply binary operation `op()` to corresponding elements in range1 [`first1`, `last1`) and range2 starting at `first2` and write to range starting at `res`

```
Out transform(In first1, In last1, In first2, Out res, BinOp op);
```

- Example:

```
inline int Square(int z) { return z*z; }

block<int,6> x = { 1, 2, 3 4, 5, 6 };
vector<int> res(x.size());

transform(x.begin(), x.end(), res.begin(), Square);

for (vector<int>::iterator it=res.begin(); it!=res.end(); ++it)
   cout << *it << " ";
cout << endl;
```

Output:

```
1 4 9 16 25 36
```


Replacing Elements

- Replace all elements with value `old` in-place with value `new`

```
void replace(For first, For last, const T& old, const T& new);
```

- Replace all elements satisfying predicate `pred` in-place with value `new`

```
void replace_if(For first, For last, Pred pred, const T& new);
```

- ➠ Also available as `replace_copy` and `replace_copy_if` with additional third parameter describing result range starting at output iterator `out`

Filling Ranges

- Assign `value` to all elements in range [`first`, `last`) or [`first`, `first+n`)

```
void fill(For first, For last, const T& value);
void fill_n(Out first, Size n, const T& value);
```

- Assign result of calling `gen()` to all elements in range [`first`, `last`) or [`first`, `first+n`)

```
void generate(For first, For last, Generator gen);
void generate_n(Out first, Size n, Generator gen);
```

Removing Elements

- "Remove" all elements equal to `value`

```
For remove(For first, For last, const T& value);
```

- "Remove" all elements which satisfy predicate `pred`

```
For remove_if(For first, For last, Pred pred);
```

- "Remove" all consecutive duplicate elements (using `operator==()` or predicate `pred`)

```
For unique(For first, For last);
For unique(For first, For last, BinPred pred);
```

- ➠ Also available as `remove_copy`, `remove_copy_if`, and `unique_copy` with additional third parameter describing result range starting at output iterator `out`
- ➠ **Note**: These functions do **not** really remove the elements from the sequence but move them to the end returning the position where the "removed" elements start
- ➠ To really delete the elements from a Container `C` use `erase()`, e.g., for `remove`:

```
C.erase(remove(C.begin(), C.end(), x), C.end());
```


Permuting Algorithms

- Reverse range in-place or copy reverse range to range starting at `res`

```
void reverse(Bi first, Bi last);
Out reverse_copy(Bi first, Bi last, Out res);
```

- Exchange [`first`, `middle`) and [`middle`, `last`) in-place or copy to `res`

```
void rotate(For first, For middle, For last);
Out rotate_copy(For first, For middle, For last, Out res);
```

- Transform range into next or previous lexicographical permutation

```
bool next_permutation(Bi first, Bi last);
bool prev_permutation(Bi first, Bi last);
```

  - ➠ also available with additional parameter specifying sorting criterion `cmp`

- Randomly re-arrange elements (using the internal or an user-specified random-number generator)

```
void random_shuffle(Ran first, Ran last);
void random_shuffle(Ran first, Ran last, RandomNumberGen& rand);
```

Partitions

- Reorder the elements so that all elements satisfying `pred` precede the elements failing it

```
Bi partition(Bi first, Bi last, Pred pred);
```

- Same as partition but preserves relative order of elements

```
Bi stable_partition(Bi first, Bi last, Pred pred);
```

Swapping Elements

- Exchange values of elements `a` and `b`

```
void swap(T& a, T& b);
```

- Exchange values of elements pointed to by `it1` and `it2` (`swap(*it1, *it2)`)

```
void iter_swap(For it1, For it2);
```

- Swap the corresponding elements of range1 [`first1`, `last1`) and range2 starting at `first2`

```
For swap_ranges(For first1, For last1, For first2);
```


Sorting Ranges

- Sort elements in ascending order (smallest element first)

```
void sort(Ran first, Ran last);
```

- Like `sort()` but preserves the relative order between equivalent elements

```
void stable_sort(Ran first, Ran last);
```

- Put smallest `middle-first` elements into [`first`, `middle`) in sorted order

```
void partial_sort(Ran first, Ran middle, Ran last);
```

- Put smallest `rlast-rfirst` elements of [`first`, `last`) into [`rfirst`, `rfirst+N`) in sorted order; `N` is minimum of `last-first` and `rlast-rfirst`

```
Ran partial_sort_copy(In first, In last, Ran rfirst, Ran rlast);
```

- "Sort" sequence so that element pointed to by `nth` is at correct place
  - elements in [`first`, `nth`) are smaller than the elements in [`nth`, `last`)
  - good for calculating medians or other quantiles

```
void nth_element(Ran first, Ran nth, Ran last);
```

- All functions also available with additional parameter specifying sorting criterion `cmp`

Binary Search

- Determine whether `value` is in sequence [`first`, `last`) using binary search

```
bool binary_search(For first, For last, const T& value);
```

- Returns first / last position where element equal to `value` could be inserted without destroying ordering

```
For lower_bound(For first, For last, const T& value);
For upper_bound(For first, For last, const T& value);
```

- Return `pair<lower_bound, upper_bound>`

```
pair<For, For> equal_range(For first, For last, const T& value);
```

Merging Two Sorted Ranges

- Combine sorted ranges [`first1`, `last1`) and [`first2`, `last2`) into `res`

```
Out merge(In first1, In last1, In first2, In last2, Out res);
```

- Combine the two consecutive sorted ranges [`first`, `middle`) and [`middle`, `last`) in-place

```
void inplace_merge(Bi first, Bi middle, Bi last);
```

➠ All functions on page also available with additional parameter specifying sorting criterion `cmp`

Set Operations

- Is every element in [`first1`, `last1`) included in range [`first2`, `last2`)?

```
bool includes(In first1, In last1, In first2, In last2);
```

- Determine set-like union, intersection, difference, and symmetric difference of the ranges [`first1`, `last1`) and [`first2`, `last2`) and write to `res`

```
Out set_union(In first1, In last1,
              In first2, In last2, Out res);

Out set_intersection(In first1, In last1,
                     In first2, In last2, Out res);

Out set_difference(In first1, In last1, In first2,
                   In last2, Out res);

Out set_symmetric_difference(In first1, In last1,
                             In first2, In last2, Out res);
```

➠ Input ranges need only be *sorted*, they must be no real *sets* (duplicate elements allowed)

➠ All functions also available with additional parameter specifying sorting criterion `cmp`

Heap Operations

➠ *heap* (here) is a tree represented as a sequential range where each node is less than or equal to its parent node ➠ `*first` is largest element

❑ Turn range [`first`, `last`) into heap order

```
void make_heap(Ran first, Ran last);
```

❑ Add element at `last-1` to heap [`first`, `last-1`)

```
void push_heap(Ran first, Ran last);
```

❑ "Remove" (move to the end) largest element

```
void pop_heap(Ran first, Ran last);
```

❑ Turn heap into sorted range

```
void sort_heap(Ran first, Ran last);
```

➠ All functions also available with additional parameter specifying sorting criterion `cmp`

❑ Calculate the sum of `initval` plus all the elements in [`first`, `last`) using `T::operator+()` or binary function `op`

```
T accumulate(In first, In last, T initval);
T accumulate(In first, In last, T initval, BinOp op);
```

❑ Calculate the sum of `initval` plus the results of `first1[i]*first2[i]` for the range [`first1`, `last1`) using `operator+()`, `operator*()` or binary functions `op1`, `op2`

```
T inner_product(Input1 first1, Input1 last1,
                Input2 first2, T initval);
T inner_product(Input1 first1, Input1 last1,
                Input2 first2, T initval, BinOp op1, BinOp op2);
```

❑ Calculate running sum of all elements (or running result of `op()`)

```
Out partial_sum(In first, In last, Out result);
Out partial_sum(In first, In last, Out result, BinOp op);
```

❑ Calculate differences x_i-x_{i-1} for elements in [`first+1`, `last`) (or use `op()` instead of `-`)

```
Out adjacent_difference(In first, In last, Out result);
Out adjacent_difference(In first, In last, Out result, BinOp op);
```

➠ These functions are defined in header `<numeric>`

- block: minimal container which adheres to STL container convention (like built-in array)

```
#include <stddef.h>

template <class T, size_t N>
struct block {
   T data[N];                          // public data

   // -- public types --
   typedef T value_type;
   typedef T& reference;
   typedef const T& const_reference;
   typedef ptrdiff_t difference_type;
   typedef size_t size_type;

   // -- iterators --
   typedef T* iterator;
   typedef const T* const_iterator;

   iterator begin() { return data; }
   const_iterator begin() const { return data; }

   iterator end() { return data+N; }
   const_iterator end() const { return data+N; }
```



```
   // -- member access --
   reference operator[](int i) { return data[i]; }
   const_reference operator[](int i) const { return data[i]; }

   // -- other member functions --
   size_type size() const { return N; }
   size_type max_size() const { return N; }
   bool empty() const { return N==0; }
   void swap(block& x) {
      for (size_t n = 0; n < N; ++n)
         std::swap(data[n], x.data[n]);
   }

   operator T*() { return data; }   // provided for compatibility
                                    // with built-in arrays
};
```

- Because block is defined as a POD (Plain Old Data) type, initialization syntax still works!

```
block<int, 6> x = { 1, 4, 7, 3, 8, 4 };
```

- We still need to define global comparison operators `operator==()` and `operator<()`

```
template <class T, size_t N>
bool operator==(const block<T,N>& lhs, const block<T,N>& rhs) {
   for (size_t n = 0; n < N; ++n)
      if (lhs.data[n] != rhs.data[n])
         return false;
   return true;
}
```

```
template <class T, size_t N>
bool operator<(const block<T,N>& lhs, const block<T,N>& rhs) {
   for (size_t n = 0; n < N; ++n)
      if (lhs.data[n] < rhs.data[n])
         return true;
      else if (rhs.data[n] < lhs.data[n])
         return false;
   return false;
}
```

- STL contains *algorithm adaptors* which define `operator!=()`, `operator<=()`, `operator>()`, and `operator>=()` out of the two operators above

- If we want `block` to be a *reversible* container, we need to add also

```
#include <iterator>
```

```
template <class T, size_t N>
struct block {
   // ...
   typedef std::reverse_iterator<iterator>      reverse_iterator;
   typedef std::reverse_iterator<const_iterator>
                                          const_reverse_iterator;

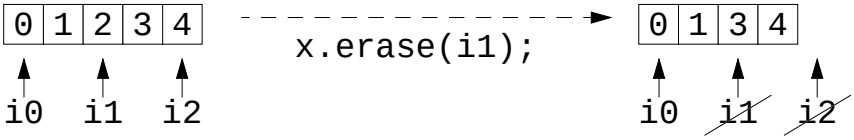
   reverse_iterator rbegin() { return reverse_iterator(end()); }
   const_reverse_iterator rbegin() const {
      return const_reverse_iterator(end());
   }

   reverse_iterator rend() { return reverse_iterator(begin()); }
   const_reverse_iterator rend() const {
      return const_reverse_iterator(begin());
   }

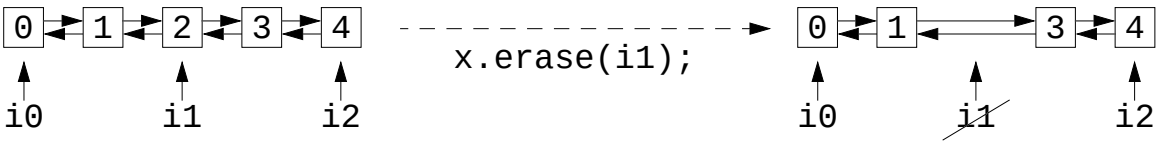
   // ...
};
```

- `vector<T>` and `deque<T>` member functions which *modify* the container (`insert`, `erase`) ***can*** invalidate iterators and pointers into the container
- *invalidated* iterator := the iterator points to another element or becomes invalid
- This is not true for `list<T>` with the exception of a iterator/pointer to a deleted element
- Example:

```
vector<int> x(5);
vector<int>::iterator i0=x.begin(), i1=x.begin()+2,
                      i2=x.begin()+4;
```




```
list<int> x(5);
list<int>::iterator i0=x.begin(), i1=i0, i2;
++i1; ++i1; i2=i1; ++i2; ++i2;
```



- When adding elements to a full `vector<T>`, automatic *resizing* occurs
- How this resizing occurs is implementation dependent
  - ➠ resizing vector one by one element would be horribly inefficient
  - ➠ usual methods are to double the size of the vector or to add fixed-sized blocks and copy over old elements
  - ➠ `vector<T>` has two counts associated with it
    - its *size* := the number of elements currently stored in the vector
    - its *capacity* := the number of elements which could be stored without resizing
    - ➠ always: size <= capacity
- The following member functions related to resizing are available:
  - Get the size of a vector

    ```
    vector<T>::size_type size();
    ```

  - Get the capacity of a vector

    ```
    vector<T>::size_type capacity();
    ```

- Resize the vector to be of length n

```
resize(size_type n, T initValue = T());
```

  - ➠ if n < size() elements at the end are removed
  - ➠ if n > size() additional elements initialized with initValue are added to the end

- Ensure that the *capacity* is at least n

```
reserve(size_type n);
```

  - ➠ can be used to pre-allocate vector memory if approximate size if known in advance
  - ➠ saves memory allocations and copies!

❑ Example

```
const int n = 9000000;
vector<float> numbers;
numbers.reserve(n);
for (int i = 0; i < n; ++i) number.push_back(random());
```

➠ without reserve, several dozens of resizes would occur. With the last resize, vector would probably allocate a 64Mbyte buffer, copy over the 32Mbyte old values, and 29Mbyte would be wasted.

❑ A list<T> has the advantage that it can be reordered by changing links

➠ list reorder functions don't *copy* elements like insert() rather they modify the list data structures that refer to the elements

❑ The following reorder member functions are provided:

- Move contents of list2 into list1 just before iter, leave list2 empty

```
list<T> list1, list2;
list<T>::iterator iter;              // points into list1
list1.splice(iter, list2);
```

- Move element pointed to by iter2 from list2 into list1 just before iter. Element is removed from list2. list1 and list2 may be the same list

```
list<T>::iterator iter2;             // points into list2
list1.splice(iter, list2, iter2);
```

- Move the range [i2, j2) from list2 into list1 just before iter. The range is removed from list2

```
list<T>::iterator i2, j2;            // point into list2
list1.splice(iter, list2, i2, j2);
```

- Combine two *sorted* lists by moving elements from `list2` into `list1` while preserving order.

```
list1.merge(list2);
```

  - ➠ if one of the list is **not** sorted, `merge` still produces one list (however unsorted)

- There are also `sort` and `merge` with a second argument specifying the ordering criterion *cmp*

```
list1.sort(cmp);
list1.merge(list2, cmp);
```

- (Really) remove duplicates that appear consecutively or elements that appear consecutively and both satisfy the predicate *pred*

```
list1.unique();
list1.unique(pred);
```

- (Really) remove all elements with the value *val* or that satisfy predicate *pred*

```
list1.remove(val);
list1.remove_if(pred);
```

- Reverse all elements in the list

```
list1.reverse();
```


- C-style built-in arrays, `string`s, `valarray`s, and `bitset`s are also containers
  - ➠ however, each lacks some aspect or the other of the STL standard containers
  - ➠ these "almost" containers are not completely interchangeable with STL containers
- Built-in arrays
  - supplies subscripting and random-access iterators in the form of ordinary pointers
  - provides no public types and doesn't know its size (like ➠ `block<T>`)
- `std::string` defined in `<string>`
  - provides subscripting, random-access iterators, and most of STL conventions
  - but implementation is optimized for use as a string of characters
- `std::bitset<N>` defined in `<bitset>`
  - like a `vector<bool>` but provides operations to manipulate bits, is of fixed size `N`, and is optimized for space
- `std::valarray` defined in `<valarray>`
  - is a (badly defined) vector for optimized numeric computation

- STL library also provides some predefined predicates and functors in header `<functional>`
  - ➠ unnecessary to invent tiny functions just to implement trivial function objects
- The following basic predicates and arithmetic functors are defined:

Predicate	#Args	Op
`equal_to`	2	`==`
`not_equal_to`	2	`!=`
`greater`	2	`>`
`less`	2	`<`
`greater_equal`	2	`>=`
`less_equal`	2	`<=`
`logical_and`	2	`&&`
`logical_or`	2	`||`
`logical_not`	1	`!`

Arith.Func.	#Args	Op
`plus`	2	`+`
`minus`	2	`-`
`multiplies`	2	`*`
`divides`	2	`/`
`modulus`	2	`%`
`negate`	1	`-`

- These basic predicates and functors are all templates with one parameter specifying the base type
  - ➠ `greater<foo>` is a functor behaving like `foo::operator>()`

- more complex functors can be composed out of trivial ones with the help of *functor adaptors*
- The following adaptors are provided by the STL:

Function	#Args	Action of Generated Functor
`bind2nd(f,y)`	1	Call binary STL functor `f` with `y` as 2nd argument
`bind1st(f,x)`	1	Call binary STL functor `f` with `x` as 1st argument
`not1(f)`	1	Negate unary predicate `f`
`not2(f)`	2	Negate binary predicate `f`
`mem_fun(f)`	0 or 1	Transform 0- or 1-argument member function `f` into functor (call through pointer)
`mem_fun_ref(f)`	0 or 1	Transform 0- or 1-argument member function `f` into functor (call through reference)
`ptr_fun(f)`	1 or 2	Transform pointer to unary or binary function `f` into functor

➠ adapters are simple forms of a *higher-order function* `:=` takes a function argument and produces a new function from it

- Example: write function that deletes all numbers smaller and including 1000 in a sorted vector
- First, we could write our usual *predicate* function:

```
bool bigger1000(int n) { return n > 1000; }
```

- Then, we use STL algorithms, to *find* and *erase* elements in the vector:

```
void g1(vector<int>& numbers) {
   vector<int>::iterator vi =
        find_if(numbers.begin(), numbers.end(), bigger1000);
   numbers.erase(numbers.begin(), vi);
}
```

- Can even do away with the local variable `vi`

```
void g2(vector<int>& numbers) {
   numbers.erase(numbers.begin(),
        find_if(numbers.begin(), numbers.end(), bigger1000));
}
```


- Use binary functor "*greater*" `greater<T>`

```
greater<int> gt;              // gt(3,4) => false
```

- Need adapter functor "*apply with 2nd argument bound to value*" `bind2nd()` to fix value 1000

```
(bind2nd(gt,1000)) (999)     // => false
(bind2nd(gt,1000)) (1001)    // => true
```

- Now we can replace `bigger1000` by predicate function:

```
void g3(vector<int>& numbers) {
   greater<int> gt;
   numbers.erase(numbers.begin(),
      find_if(numbers.begin(), numbers.end(), bind2nd(gt,1000)));
}
```

- Of course, default constructor call `greater<int>()` can replace local variable `gt`:

```
void g4(vector<int>& numbers) {
   numbers.erase(numbers.begin(),
        find_if(numbers.begin(), numbers.end(),
             bind2nd(greater<int>(), 1000)));
}
```

- Read persons from standard input and store them sorted by first name in a list
  - First, define appropriate comparison function

```
bool cmpPtrFirstName(const Person* p1, const Person* p2) {
   return p1->firstName < p2->firstName;
}
```

  - Next, use it to find the right place to insert the new person in the list

```
list<Person*> folks;
string name, fname;

while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   list<Person*>::iterator it = folks.begin();
   while ( it != folks.end() ) {
      if ( cmpPtrFirstName(p, *it) ) break;
      ++it;
   }
   folks.insert(it, p);
}
```

- How about using STL algorithms (and predefined functors)?

- Use STL `find_if` algorithm and `bind1st` and `ptr_fun` functor adaptors to find the right place in the list

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   list<Person*>::iterator it
     = find_if(folks.begin(), folks.end(),
               bind1st(ptr_fun(cmpPtrFirstName), p));
   folks.insert(it, p);
}
```

- Can even do without local variable `it`

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   folks.insert(find_if(folks.begin(), folks.end(),
                bind1st(ptr_fun(cmpPtrFirstName), p)), p);
}
```

- Is it possible to avoid the need for functor adaptor `ptr_fun`?

- Write STL compatible functor by deriving from `binary_function<arg1_type, arg2_type, ret_type>` helper class:

```
struct cmpPtrFirstNameFtor :
   public binary_function<const Person*, const Person*, bool> {

   bool operator()(const Person* p1, const Person* p2) const {
      return p1->firstName < p2->firstName;
   }
};
```

- Then:

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   folks.insert(find_if(folks.begin(), folks.end(),
                bind1st(cmpPtrFirstNameFtor(), p)), p);
}
```

- Can we do without functor adaptor `bind1st`?

- Implement unary functor with state (remembering the person to compare to):

```
struct cmpPtrWithFirstName :
   public unary_function<const Person*, bool> {

   cmpPtrWithFirstName(const Person *pers) : ref(pers) {}

   bool operator()(const Person *other) const {
      return ref->firstName < other->firstName;
   }

   const Person* ref;
};
```

- Usage:

```
folks.insert(find_if(folks.begin(), folks.end(),
             cmpPtrWithFirstName(p)), p);
```

- Note! Probably more efficient to simply insert persons into the list and sort them once:

```
while ( cin >> fname >> name )
   folks.push_back(new Person(fname, name));
folks.sort(cmpPtrFirstName);
```

The STL header `<iterator>` provides

- iterator *primitives*: utility classes and functions to simplify the task of defining new iterators
- iterator *operations*:
  - Increments (or decrements for negative `n`) iterator `i` by `n`

    ```
    void advance(InputIter& i, Distance n);
    ```

  - Returns the number of increments or decrements needed to get from `first` to `last`

    ```
    difference_type distance(InputIter first, InputIter last);
    ```

- *predefined* iterators:
  - Insert iterators ➠ assignment to iterator inserts value in container

    ```
    back_inserter(Container& x);
    front_inserter(Container& x);
    inserter(Container& x, Iterator i);
    ```

  - Stream iterators ➠ stepping iterator means reading/writing values from/to stream

    ```
    istream_iterator(istream& s);
    ostream_iterator(ostream& s, const char* delim);
    ```

  - Reverse iterators (see `block<T>` example)

- `ostream_iterator`s useful for printing container (`cont<T> C`) if `operator<<` for type of container element `T` exists. Instead of

  ```
  for (cont<T>::iterator it=C.begin(); it!=C.end(); ++it)
     cout << *it << "\n";
  ```

  you can use

  ```
  copy(C.begin(), C.end(), ostream_iterator<T>(cout, "\n"));
  ```

- Also useful for directly printing results of mutating sequence algorithms:

  ```
  inline int Square(int z) { return z*z; }

  block<int,6> x = { 1, 2, 3, 4, 5, 6 };
  ostream_iterator<int> ot(cout, " ");

  transform(x.rbegin(), x.rend(), ot, Square);
  cout << "--- ";
  transform(x.begin(), x.end(), x.rbegin(), ot, multiplies<int>());
  cout << endl;
  ```

  Output:

  ```
  36 25 16 9 4 1 --- 6 10 12 12 10 6
  ```

- Result range of mutating sequence algorithms (e.g., `transform`) must have enough elements to store result (because the elements are ***assigned*** the result values, they are not ***inserted***)

➠ Make sure result container is large enough:

```
block<int,6> x1 = { 1, 2, 3, 4, 5, 6 };
vector<int> res1(x1.size());

copy(x1.begin(), x1.end(), res1.begin());     // OK!
```

➠ res1: {1, 2, 3, 4, 5, 6}

```
copy(x.begin(), x.end(), res1.rbegin());      // reversed!
```

➠ res1: {6, 5, 4, 3, 2, 1}

➠ Or use iterator adaptors:

```
block<int,6> x2 = { 1, 2, 3, 4, 5, 6 };
list<int> res2;

copy(x2.begin(), x2.end(), back_inserter(res2));  // OK!
```

➠ res2: {1, 2, 3, 4, 5, 6}

```
copy(x2.begin(), x2.end(), front_inserter(res2)); // reversed!
```

➠ res2: {6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6}


```
#include <fstream>           // get the I/O facilities
#include <vector>            // get the vector facilities
#include <algorithm>         // get the operations on containers
#include <iterator>          // get the iterator facilities
using namespace std;

int main (int argc, char *argv[]) {
  ifstream ii(argv[1]);  // setup input
  ofstream oo(argv[2]);  // setup output
  vector<int> buf;       // vector used for buffering

  // initialize buffer from input
  copy(istream_iterator<int>(ii),
       istream_iterator<int>(),     // def ctor => EOF iterator
       back_inserter(buf));

  // sort the buffer
  sort(buf.begin(), buf.end());

  // copy to output
  copy(buf.begin(), buf.end(), ostream_iterator<int>(oo,"\n"));

}
```

- STL Containers always contain ***copies*** of objects
  - adding objects to or getting objects from a container means copying
  - moving objects in a sequence container (because of `insert` or `erase`) means copying
  - changing order in a sequence container (because of `sort`, `reverse`, ...) means copying

- make sure copying objects works correctly
  - pointer members ➠ deep copy?!
  - derived objects ➠ slicing!

- for “heavy” objects or in the presence of inheritance:
- use containers of pointers to objects
- better: use *smart pointers* for automatic memory management


```
class P {
public:
   P(const string& _n) : n(_n), i(1) { print(cout, "Construct"); }
   P(const P& p) : n(p.n), i(p.i+1) { print(cout, "Copy"); }
   P& operator=(const P& p) {
      print(cout, "Delete");
      n = p.n;  i = p.i+1;
      print(cout, "Copy Assign");
      return *this;
   }
   ~P() { print(cout, "Delete"); }
   ostream& print(ostream& ostr, const char* action) const {
      return ostr << action << "(" << n << i << ")\n";
   }
private:
   string n;     // My name is "n"
   int i;        // I am the i-th copy
};
```

```
ostream& operator<<(
  ostream& ostr,
  const P& p) {
  return p.print(ostr, "Out");
}

vector<P> vec1, vec2;

vec1.push_back( P("Bob") );
vec1.push_back( P("Joe") );
vec1.push_back( P("Sue") );

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<P>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<P>(cout));
```

Possible output (depends on resize behavior):

```
Construct(Bob1)
Copy(Bob2)       //passing by value
Delete(Bob1)     //-
Construct(Joe1)
Copy(Joe2)       //passing by value
Delete(Joe1)     //-
Construct(Sue1)
Copy(Bob3)       //vector resize
Copy(Joe3)       //vector resize
Copy(Sue2)       //passing by value
Delete(Bob2)     //-
Delete(Joe2)     //-
Delete(Sue1)     //-
Copy(Bob4)
Delete(Bob4)
Copy Assign(Sue3)
Out(Bob3)
Out(Joe3)
Out(Sue2)
Out(Sue3)
Delete(Sue3)
Delete(Bob3)
Delete(Joe3)
Delete(Sue2)
```



```
ostream& operator<<(
  ostream& ostr,
  const P* p) {
  return p->print(ostr, "Out");
}

vector<P*> vec1, vec2;

vec1.push_back( new P("Bob") );
vec1.push_back( new P("Joe") );
vec1.push_back( new P("Sue") );

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<P*>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<P*>(cout));
```

- Output:

```
Construct(Bob1)
Construct(Joe1)
Construct(Sue1)
Out(Bob1)
Out(Joe1)
Out(Sue1)
Out(Sue1)

//no Delete?!
```

- Can do manual `delete` but might get complicated in larger program!
- Use smart pointer!

```
typedef shared_ptr<P> PPtr;

ostream& operator<<(
  ostream& ostr,
  const PPtr& p) {
  return p->print(ostr, "Out");
}

vector<PPtr> vec1, vec2;

vec1.push_back(PPtr(new P("Bob")));
vec1.push_back(PPtr(new P("Joe")));
vec1.push_back(PPtr(new P("Sue")));

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<PPtr>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<PPtr>(cout));
```

- Output:

```
Construct(Bob1)
Construct(Joe1)
Construct(Sue1)
Out(Bob1)
Out(Joe1)
Out(Sue1)
Out(Sue1)
Delete(Bob1)
Delete(Joe1)
Delete(Sue1)
```

- Note! `Sue1` gets (correctly) deleted just once!

- Idea: Implement pointer-like class which automatically deletes content if necessary
- Example: Shared ownership based on reference counting

```
template<typename T>
class shared_ptr {

private:
   T*     px;     // contained pointer
   long*  pn;     // pointer to reference counter

public:
   explicit shared_ptr(T* p = 0) : px(p), pn(new long(1)) {}

   ~shared_ptr() { if (--*pn == 0) { delete px; delete pn; } }

   shared_ptr(const shared_ptr& r) : px(r.px) { ++*(pn = r.pn); }

   shared_ptr& operator=(const shared_ptr& r);

   T& operator*() const  { return *px; }
   T* operator->() const { return px; }
   T* get() const        { return px; }
   bool unique() const   { return *pn == 1; }
};
```

```
template<typename T>
shared_ptr<T>& shared_ptr<T>::operator=(const shared_ptr<T>& r) {
   if (pn != r.pn) {
      // increment new reference counter
      ++*(r.pn);
      // decrement old reference counter, delete if last
      if (--*pn == 0) { delete px; delete pn; }
      // copy pointer and counter
      px = r.px;
      pn = r.pn;
   }
   return *this;
}
```

- Example smart pointer too simple for professional use
  - other ownership models (no copy allowed, copy transfers ownership, shared, ...)
  - const and inheritance issues
  - multi-threading?
- ➠ better: see smart pointer collection at `boost.org`

- To eliminate all objects in a container `C` that have a particular value `val`:
  - `vector`, `deque`, `string`: use `erase/remove` idiom

    ```
    C.erase(remove(C.begin(), C.end(), val), C.end());
    ```
  - `list`: use `list` method `remove(val)`
  - associative container: use container method `erase(key)`
- To eliminate all objects in a container that satisfy a particular predicate `pred`:
  - `vector`, `deque`, `string`: use `erase/remove_if` idiom
  - `list`: use `list` method `remove_if(pred)`
  - associative container: use `remove_copy_if` and then `swap` elements

    ```
    AssocCtr C, OK;
    remove_copy_if(C.begin(), C.end(),
                   inserter(OK, OK.end()), pred);
    C.swap(OK);
    ```

- Which method should be used to find objects in STL containers?
- ➠ the faster and simpler, the better!
  - unsorted containers: can only use linear-time algorithms `find`, `find_if`, `count`, and `count_if`
  - sorted sequence containers: can use faster `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` ***algorithms***
  - associative containers: can use faster `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` ***methods***

- ➠ Note! the linear-time algorithms use equality to test whether two objects are the same, the others equivalence!

  equality ⇔ `x1 == x2`

  equivalence ⇔ `! ( x1 < x2 ) && ! ( x2 < x1 )`

- Summary:

What You Want to Know	Algorithm to Use		Member Function to Use	
	Unsorted Range	Sorted Range	`set` or `map`	`multiset` or `multimap`
Does desired value exist?	`find`	`binary_search`	`count`	`find`
Where is first object of desired value?	`find`	`equal_range`	`find`	`find` or `lower_bound`
Where is first object with a value not preceding desired value?	`find_if`	`lower_bound`	`lower_bound`	`lower_bound`
Where is first object with a value succeeding desired value?	`find_if`	`upper_bound`	`upper_bound`	`upper_bound`
How many objects have desired value?	`count`	`equal_range`	`count`	`count`
Where are all objects with desired value?	`find` (iteratively)	`equal_range`	`equal_range`	`equal_range`

- Use `vector<T>` instead of built-in arrays and `string` instead of `char*`!
- What about calling old code?

- Consider, we need to call an old C function

```
void doSomething(const int *pInts, size_t numInts);
```

but we have

```
vector<int> v;
```

  - pass address of first element and size (But don't forget empty vectors!)

```
if ( !v.empty() ) { doSomething(&v[0], v.size()); }
```

- What about

```
void doSomething(const char *pString);
string s;
```

  - Use string method `c_str()`

```
doSomething(s.c_str());
```


Books on STL

- Austern, *Generic Programming and the STL*,
  Addison-Wesley, 1998, ISBN 0-201-30956-4.
  - Most up-to-date and complete book on STL, good tutorial and reference
- Josuttis, *The C++ Standard Library – A Tutorial and Reference*
  Addison-Wesley, 1999, ISBN 0-201-37926-0.
  - Most up-to-date and complete book on **whole** C++ standard library
    (including iostream, string, complex, ...)
- Meyers, *Effective STL*,
  Addison-Wesley, 2001, 0-201-74962-9.
  - 50 specific ways to improve your use of the standard template library

General C++ Books (but cover STL very well)

- Stroustrup, *The C++ Programming Language*, **Third Edition**
- Lippman and Lajoie, *C++ Primer*, **Third Edition**

WWW STL Information

- SGI STL (public-domain implementation plus well-organized on-line documentation)
  `http://www.sgi.com/tech/stl/`
- Adapted SGI STL (value-added and more portable version of SGI STL)
  `http://www.stlport.org`
- BOOST: free, peer-reviewed, portable C++ libraries
  `http://www.boost.org`

- Watch out for differences between original public domain STL from HP and the C++ standard:

	HP (SGI) STL	**C++ Standard STL**
container adaptors `stack`, `queue`, ...	`Adaptor<Container<T> >`	`Adaptor<T>` `Adaptor<T, Container<T> >`
iterator access	`(*iter).field`	`iter->field` or `(*iter).field`
scope of items	global scope	in namespace `std`

headers	`<algo.h>`
	`<function.h>`
	`<stack.h>`
	`<map.h>, <multimap.h>`
	`<set.h>, <multiset.h>`
		`<algorithm>, <numeric>`
		`<functional>`
		`<stack>, <queue>`
		`<map>`
		`<set>`

`*` functor	`times<T>`	`multiplies<T>`
algorithms: `count`, `count_if`, `distance`	`int n = 0;` `func(..., n);`	`int n;` `n = func(...);`
I/O iterators (`ostream` analog)	`istream_iterator` `<T,Distance>`	`istream_iterator` `<T,charT,traits,Distance>`

Programming in C++

☆☆☆ Advanced C++ ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Advanced C++ — Private Inheritance: Access Specifiers

- Private inheritance allows to implement a class in terms of another one:
  - e.g. try to implement `Set` as `List` without `head`/`tail` and new `insert` through *derivation with (compile-time) cancellation* (**but**: not allowed in ARM C++!!!)

```
class List {
public:
   elem* head();
   elem* tail();
   int size();
   bool has(elem&);
   void insert(elem&);
};
```

```
class Set: public List {
public:
   void insert(elem&);
private:
   List::head;  // error! in ARM C++
   List::tail;  // error! in ARM C++
};
```

- However, it works the other way:

```
class Set: private List {
public:
   void insert(elem&);
   List::size;
   List::has;
};
```

- In new C++ Standard:
  - derivation with cancellation now allowed
  - *"using declaration*" should be used instead of access specifiers:

```
using List::size;
using List::has;
```

- But reuse through private inheritance has a severe limitation:
  - the following is illegal because no class may appear as a direct base class twice

```
class Arm { ... };
class Leg { ... };

class Robot: Arm, Arm, Leg, Leg {    // illegal!
private:
   // Robot specific members
public:
   Robot();
};
```

- Solution: use *class members* or *layering* (nested classes)

```
class Robot {
private:
   Arm leftarm, rightarm;
   Leg leftleg, rightleg;
   ...
```

- **avoid private inheritance if possible**

- Sometimes, the design of a class requires an auxiliary class
  - Solution: nested classes

```
class Stack {
private:
   class StackNode {
      T data;
      StackNode *next;
    public:
      StackNode( const T& d,
                 StackNode *n)
         : data(d), next(n) {}
   };
   StackNode *top;

public:
   Stack();
   ~Stack();
   void push(const T& data);
   T pop();
};
```

  - if compiler doesn't support nested classes use private global class with friend

```
class StackNode {
private:
   T data;
   StackNode *next;
   StackNode( const T& d,
              StackNode *n)
        : data(d), next(n) {}
   friend class Stack;
};

class Stack {
public:
   Stack();
   ~Stack();
   // ...
};
```

- Nested classes can now be forward declared just like other classes:

```
class Outer {
   class Inner;
public:
   Inner* getCookie();
private:
   class Inner { /* ... */ };
};
```

- The definition can now also be outside the class:

```
class Outer {
   class Inner;
public:
   Inner* getCookie();
};

class Outer::Inner { /* ... */ };
```


- C programmers are already familiar with the C style cast syntax:

```
(Type)Expression            // double d = (double) 5;
```

- C++ introduced the functional style cast syntax

```
Type(Expression)            // double d = double(5);
```

- In specific circumstances, casts are useful (and necessary)
  - But "old" style casts convert everything in everything (compiler believes you)
- 4 new casts added to C++ to allow finer control over casting
  - new syntax `xxx_cast<type>(expr)` is easier to locate for tools/programmer
  - it is harder to type (many believe: a good thing)
  - makes clear what kinds of casts are "meaningful" (useful)

- `comst` / `volatile` conversion
  - `const_cast`
  - allows only to change the "constness" (of pointer and references) in a conversion:

```
void foo(double&);
const double PI = 3.1415;
foo(const_cast<double&>(PI));        // OK
int d = const_cast<int&>(PI);        // error: more than const!
```

- compile–time checked conversion
  - `static_cast`
  - allows to perform any conversion where there is an implicit conversion in the opposite direction (most frequent use of casts in C):

```
int total = 500, days = 9;
double rate = static_cast<double>(total) / days;

enum color {red=0, green, blue}; int i = 2;
color c = static_cast<color>(i);   // c = blue;

foo(static_cast<int>(PI));         // error: changes constness!
```


- run-time checked conversion
  - `dynamic_cast`
  - allows to perform *safe casts* down or across an inheritance hierarchy
  - Failed casts return null pointer (when casting pointers) or exception (with references)

```
base b, *pb;
derived d, *pd;

pb = &d;                                // fine

// ... later
pd = dynamic_cast<derived *>(pb);       // non-null only if pb
                                        // really points to
                                        // derived object

d = dynamic_cast<derived>(b);           // error: no pointer!
```

- unchecked conversion
  - `reinterpret_cast`
  - allows to perform any conversion
  - result is nearly always implementation-defined
  - non-portable!

```
typedef void(*FuncPtr)();
FuncPtr funcPtrArray[10];

int doSomething();

funcPtrArray[0] = reinterpret_cast<FuncPtr>(&doSomething);
```

- If your compiler doesn't support the new style casts yet, you can use the following approximation

```
#define static_cast(TYPE,EXPR)       (TYPE)(EXPR)
#define reinterpret_cast(TYPE,EXPR)  (TYPE)(EXPR)
#define const_cast(TYPE,EXPR)        (TYPE)(EXPR)

// doesn't tell you when the casts fails; use with care
#define dynamic_cast(TYPE,EXPR)      (TYPE)(EXPR)
```


- Current C++ class libraries often use their "home-brewed" dynamic type framework:

```
if (shape_ptr->myType() == RectangleType) {
   Rectangle *rec_ptr = (Rectangle *)shape_ptr;
   // ...
}
```

- Every major library came up with there own facility; therefore, a standardisation took place:

```
if (Rectangle *rec_ptr = dynamic_cast<Rectangle *>(shape_ptr)) {
   // rec_ptr in scope and not null
}
// rec_ptr no longer in scope
```

- There is also a type enquiry operator `typeid` returning a reference to a type description object of type `type_info` (it can be compared and has a name)

```
if (typeid(*shape_ptr) == typeid(Circle)) {
   // it's a Circle pointer
}

const type_info& rt = typeid(*shape_ptr);
cout << "shape_ptr is a pointer to " << rt.name() << endl;
```

- Recall `operator[]` from `safeArray`:

```
#include <assert.h>
#include "safearray.h"

T& safeArray<T>::operator[](int index) {
   assert(index >= 0 && index < Array<T>::size());
   return Array<T>::operator[](index);
}
```

- Printing error message and exit is sometimes too drastic
- error codes / error return values not intuitive and error-prone
  - sometimes error code cannot returned or error return value does not exist (see above)
  - can be ignored by caller
  - must be passed up calling hierarchy (in deeply nested function calls)
  - could use `setjmp` / `longjmp`; fine for C, but don't call destructors of local objects

➠ **Solution: use exception handling!**

- Exception Handling consists of three parts:
  - `try`ing to execute possibly failing code
  - `throw`ing an exception in case of failure
  - `catch`ing the exception and handling it (exception handler)

`try` Block:

```
try {
   // code from which exception may be thrown
}
```

- Region of program from which control can be transferred directly to an exception handler elsewhere in program
- Transferring control from try block to exception handler is called "*throwing an exception*"

`catch` Block:

- Syntax:

```
try {
   // code from which exception may be thrown
} catch ( exception-type-1 &e1 ) {
   // exception handling code for type 1 exceptions
} catch ( exception-type-2 &e2 ) {
   // exception handling code for type 2 exceptions
}
```

- Region of program that contains exception-handling code
- Each thrown exception has a type
- When exception is thrown, control transfers to first catch block in an (dynamically) enclosing block with matching type
- For efficiency, catch exceptions *by reference*
  - avoids call to copy constructor of exception object
  - avoids *slicing* (i.e. automatic conversion to base object) of exception object

`throw` Statement:

- Syntax:

```
throw exception-type;
```

- Causes immediate transfer of control to exception handler (`catch` block) of matching type
- *Copy* of exception object is thrown (automatically)
- Throwing an exception causes abnormal exit from a function
- Normal exit calls destructor for all local objects. For consistency, throwing an exception invokes destructors for all objects instantiated since entering `try` block
- Order of destructor invocation is reverse of order of object instantiation

```
void foo (int i) {
   String s1 = "a string";
   Complex c1(i,i);
   if (some_error) {
      throw SomeException;    // destructor called for c1, then s1
   }
   String s2 = "another string";
}
```

Default `catch` Blocks:

- Syntax:

```
catch (...) {
   // exception handling code for default case
}
```

- Matches *all* exception types
- Use at most one default `catch` block for each `try` block
- Default `catch` block (if any) should be the *last* `catch` block accompanying a given `try` block
- Good programming practice dictates that every `try` block should have a default `catch` block

Re-throwing Exceptions:

- allows some exceptions to be handled locally, others to be handled by high-level general purpose exception handlers
- Re-throw current exception by placing throw statement with no type specifier: `throw;`

Exception Hierarchies

- Exception types can be user-defined (class) types
- Exception hierarchy = hierarchy (build using C++ inheritance mechanism) of user-defined exception types
- Exception base class = "category" of exceptions
- Derived exception class = individual exception type
- catch block for base type catches all exceptions in category
- Example from Standard C++ Library:

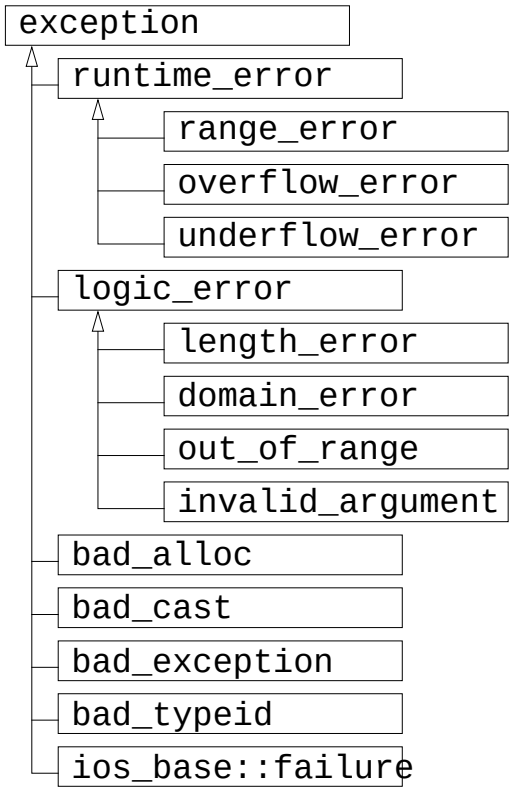
```
class runtime_error : public exception {};
class underflow_error : public runtime_error {};
class range_error : public runtime_error {};
```

- Standard exceptions thrown by the language:

Name	Thrown by
bad_alloc	new
bad_cast	dynamic_cast
bad_typeid	typeid
bad_exception	exception specification
out_of_range	at
	bitset<>::operator[]
invalid_argument	bitset constructor
overflow_error	bitset<>::to_ulong
ios_base::failure	ios_base::clear

➠ Standard exceptions do **NOT** include asynchronous events like UNIX signals (e.g., segmentation violation) or math library errors (e.g., overflow, div by zero)!

- Standard exceptions hierarchy:



Exception Specifications:

- allow to specify list of exceptions, functions might throw

```
void f1() throw(E1, E2);     // f1 can throw E1 or E2 exceptions
void f2() throw();           // f2 doesn't throw exceptions
void f3();                   // exception spec for f3 unknown!
```

  - is checked at runtime
  - `unexpected()` is called if exception specification doesn't hold
  - which by default calls `terminate()` (can be changed with `set_unexpected()`)

Function try blocks:

- allow to catch exceptions in a method body and its member initialization list

```
derived::derived(int i) try : base(i) {
   // body ...
}
catch (...) {
   // exception handler for initializer list + body ...
}
```

- Improved version of class `safeArray`:
  - Exception defined in `safearray.h`:

```
class InvalidIndex {
public:
   InvalidIndex(int i) : idx(i) {}
   int invalid() { return idx; }
private:
   int idx;
};
```

    - ➠ should actually be a subclass of `exception`, `logic_error`, or `out_of_range`
    - ➠ should also be nested class inside `safeArray`

  - New version of `operator[]`:

```
T& safeArray<T>::operator[](int index) {
   if (index < 0 || index >= Array<T>::size())
     throw InvalidIndex(index);
   return Array<T>::operator[](index);
}
```


- Simple example for exception handling:

```
#include <iostream>
#include "safearray.h"

void initialize (safeArray<int>& a, int val) {
   for (int i=0; i<=a.size(); ++i) a[i] = val;
}

int main(int, char**) {
   safeArray<int> a(10);

   try {
      initialize(a, 42);
   } catch (InvalidIndex& idx) {
      std::cerr << "Access to invalid index a["
                << idx.invalid() << "]" << std::endl;
   }
}
```

- ➠ **making programs fool-proof with exceptions is still hard; not much experience for now**
- ➠ For more details and guidelines see Stroustrup, *3rd. Edition* or Meyers, *More Effective C++*

- Compilation-unit local (file local) declarations possible through *unnamed namespaces*:

```
namespace {
   int internal_version;
   void check_filebuf();
}
```

is equivalent to

```
namespace SpecialUniqueName {
   int internal_version;
   void check_filebuf();
}
using namespace SpecialUniqueName;
```

➠ **use unnamed namespaces instead of** `static` **declarations**:

```
   // -- deprecated!
   static int internal_version;
   static void check_filebuf();
```


- A shorter alias for a long namespace name can also be defined, e.g.,

```
namespace Chrono = Chronological_Utilities;
```

This allows you to mix'n'match libraries more easily:

```
namespace lib = Modena;
//namespace lib = RogueWave;
//namespace lib = std;

// never need to change this:
lib::string my_name;
```

- Namespaces are not yet supported by many current compilers. You can partition the global namespace now by using `struct` and using fully qualified names to access them

```
struct MyNamespace {
   static const float version = 1.6;
};

cout << "Version " << MyNamespace::version << endl;
```

For convenience, you can provide typedefs, references, ... as short-cuts in a separate header file, so people without global namespace problems can use them in a "normal" way

Templates have been extended by the standard committee in many ways.
(beyond how they are described in most books)

- Template friend declarations and definitions are permitted in class definitions and class template definitions.
  - Example: class definition that declares a template class `Peer<T>` as a `friend`:

```
class Person {
private:
   float my_money;
   template<class T> friend class Peer;
};
```

  - Example: class template that declares a template class `foo<T>` as a `friend`:

```
template <class T>
class bar {
   T var;
   friend class foo<T>;
};
```


- The new keyword `typename` can be used instead of `class` in template parameter lists

```
template<typename T> class List { /*...*/ };
```

  It is also used for resolving ambiguities:

```
template<class T> class Problem {
public:
   void ack() {
      typename T::A * B;      // without typename, could be
   }                          // interpreted as multiplication
};                            // T::A * B;
```

- Templates and `static` members:

```
template <class T> class X {
   static T s;
};

template <class T> T X<T>::s = 0;
```

  If needed, specialization:

```
template<> double X<double>::s = 1.0;
```

- Template as template arguments

```
template<class T> class List;
template<class T> class Vector;

template<class T, template<class U> class C = List>
class Group {
   C<T> container;
   //...
};

Group<int>          group_int_list;
Group<int, Vector>  group_int_vector;
```

- Explicit instantiation (currently highly unportable)

```
template<typename T> class List {
   bool has(T&) { /* ... */ }
};

template class List<int>;   // request inst. of List for T=int
template bool List<elem>::has(elem&);  // inst. of member 'has'
```


- Partial Specialization

```
template<class U, class V> class relation;  #1 primary template
template<> class relation<int, int>;        #2 specialization
template<class T> class relation<T*, T*>;   #3 partial special.

relation<char*, char*> r1;   # uses 3
relation<int*, char*> r2;    # uses 1 (different pointer types!)
```

Note: function templates cannot be partial specialized, however similar results can sometimes be achieved with template function overloading

- Use of `template` to disambiguate template names (can only be necessary for calling member template functions inside other templates)

```
class X {
public:  template<int Nt> X* alloc();
};

template<class T> void f(T* p) {
   T* p1 = p->alloc<200>();          // ERROR: < means less than
   T* p2 = p->template alloc<200>(); // OK
}
```

- Current compilers typically only support the *inclusion model* for template usage
  - Bodies of
    - member functions of class templates
    - function templates

    need to be "*included*" before they can be used
  - Cannot be "*compiled away*" in separate template implementation file (e.g., `.cpp`)

- New keyword `export` to support the so-called *separation model*:

```
// main.C:
export template<typename T> T twice(T);    // template decl only

int main() {
   int i = 2;
   int j = twice(i);
}

// template_implementation.cpp:
export template<typename T> T twice(T t) { return t*t; }
```


- We know already static class member:
  - *Declaration*: in class definition (in `.h`)

    ```
    class Complex { .... static int num_numbers; ... };
    ```
  - *Definition*: needed elsewhere (in `.C`, ***not*** in `.h`)

    ```
    int Complex::num_numbers = 0;
    ```

- As the initialization occurs outside the class, this wouldn't allow class related constants
  - Initialization is now allowed for integral `static const` data members
  - *Definition* + *Declaration*s now in class definition (in `.h`)

    ```
    class Data {
       static const int max_size = 256;
       // enum {max_size = 256};  // use this if above doesn't work
       char buffer[max_size];
    };
    ```
  - *Definition* elsewhere (in `.C`) no longer needed

- *Abstract* constness of data members (`const` to the user/client), also called: conceptual constness
- *Concrete* constness of a data member (`const` to the implementation), also: bitwise constness

- Example: non-normalized `Rational` class

```
class Rational {
   void reduce() const;      // allowed for constant numbers
   ...                       // because it doesn't change
                             // "abstract" value
private:
   mutable long num;
   mutable long den;
};

void Rational::reduce() const {
   long g = gcd(num, den);
   num /= g;                          // ERROR without mutable
   den /= g;                          // ERROR without mutable
}
```


- Problem:

```
class Vector {
public:
   Vector(int len);   // constructs a Vector len elements long
};

void print(const vector& v);
print(3);    // creates temp Vector(3) and passes it to print!
```

  - Obviously the constructors are needed but they are not always suitable as conversions as well

- Solution: mark constructor "non-converting" with `explicit` (disallows implicit conversions)

```
class Vector {
public:
   explicit Vector(int len);
};

print(3);    // now error!

Vector v1(3), v2 = Vector(3);        // OK
Vector v3 = 3;                       // error!
```

- Variables can be declared inside conditions (and for statements)

```
if (int d = prim(345)) {
   i /= d;
   break;
}
```

```
while (Word& w = readnext()) {
   process(w);
}
```

- You can now overload functions on enumeration types

```
enum Status { OK, Busy, Stopped };
void process(Status s);
void process(int n);        // was error, now OK
```

- There are now separate versions of `new` and `delete` for array allocation for overloading

```
class Foo {
   void* operator new(size_t);       // already available
   void  operator delete(void *);
   void* operator new[](size_t);     // recently added to C++
   void  operator delete[](void *);
};
```

- pointer to `extern "C"` functions is now a different type than pointer to global C++ function

- Scope of declaration inside `for` loop changed
  - ➠ now only valid inside loop body
- Template name lookup and template instantiation procedures changed
- `new` now throws `bad_alloc` exception when out of memory

OLD:

```
X *p=new X;
if (p==NULL) {
   do_something();
}
```

NEW1:

```
#include <new>

X *p=new(nothrow)X;
if (p==NULL) {
  do_something();
}
```

NEW2:

```
#include <new>
try {
  X *p=new X;
} catch(bad_alloc &) {
  do_something();
}
```

- `. . .`

- Good summary:

  Jack W. Reeves
  (B)leading Edge: Moving to Standard C++
  C++ Report, Jul/Aug 1997

- Incompatibilities between Cfront's `iostream` and Standard `iostream`. Biggest changes are:
  - The Standard C++ library puts most library identifiers in the namespace `std`, e.g.,
    `ostream` is now `std::ostream`
    - import identifiers using `namespace` declarations
  - stream classes are now templates taking the character type as parameter
    `typedef basic_istream<char,char_traits<char> > istream;`
  - Base class ios is split into character type dependent and independent portions
    - `ios::flags` now are `ios_base::flags`
  - Can now throw exceptions in addition to setting error flag bits
  - Internationalization using locale
  - Assignment and copying of streams is prohibited
  - File descriptors (through member function `fd()`) are not supported any longer
  - `string` based `stringstream` class replaces `char*` based `strstream`

- `char *p = "a string";`
  - String literals are now `const char*`
- Postfix `operator++` on `bool` operand
  - Don’t use it
- `static` keyword to declare objects local to file scope
  - Use unnamed `namespace`
- access declaration
  - using declarations
- `strstream` class
  - Use `stringstream` class
- Standard C Library headers
  - use new C++ C Library headers

Programming in C++

☆☆☆ Object-Oriented Design ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

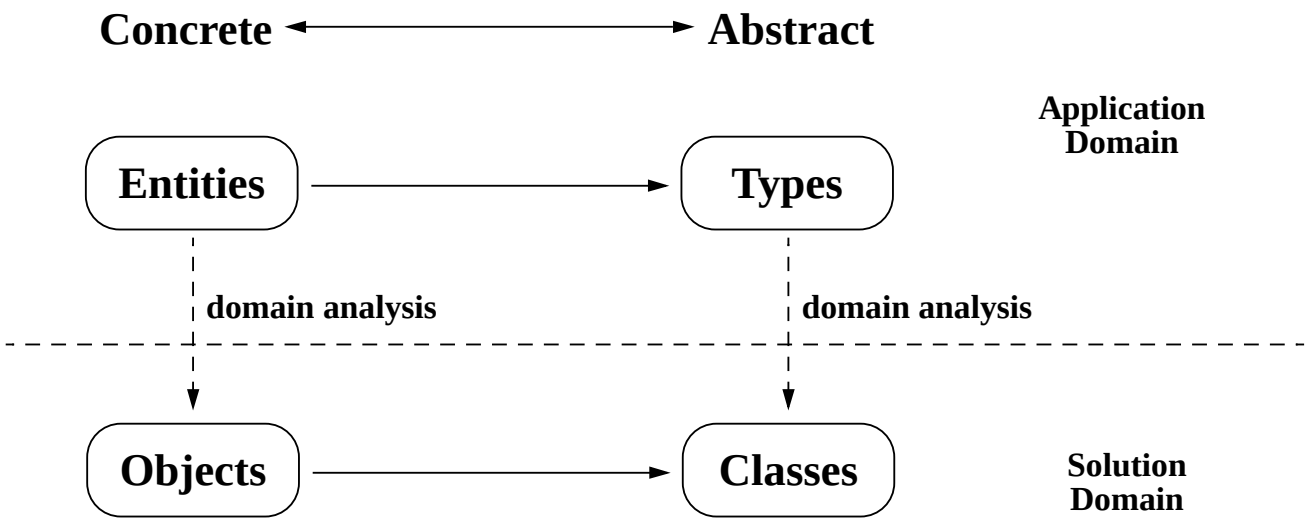
Forschungszentrum Jülich
Germany

- Ranking of software lifecycle activities
  1.) Design
  2.) Design
  3.) Design
  4.) Implementation

- "*The time-consuming thing to learn about C++ is not syntax, but design concepts.*" [Stroustroup]

- "*If the design is right, the implementation is trivial.*
  If you get stuck how to implement something, go back to design."

- The most important single aspect of software development is to be clear about what you are trying to build
- Successful software development is a long-term activity
- The systems we construct tend to be at the limit of complexity that we and our tools can handle
- There are no "cookbook" methods that can replace intelligence, experience, and good taste in design and programming
- Experimentation is essential for all nontrivial software development
- Design and programming are iterative activities
- The different phases of a software project, such as design, programming, and testing, cannot be strictly separated
- Programming and design cannot be considered without also considering the management of these activities



- In addition,
  - type relationships can be modelled with inheritance (public or private)
  - entity relationships can be modelled with object hierarchies

- Identify the entities in your application domain
  - resources, events, (hardware) parts, software interfaces, **concepts**, . . .
- Identify the behaviours of the entities
  - services, tasks, functionality, responsibilities, . . .
- Identify relationships/dependencies between entities
  - *is-a, is-kind-of, is-like, has-a, is-part-of, uses-a, creates-a,* . . .
- Broaden Design (**Guideline**: should be implementable in at least 2 different ways)
  - to be able to reuse later (design + source code!)
  - helps the system evolve / maintain
  - to be sure the implementation can fulfil its requirements
- Create a C++ design structure from the entities
  - define classes (and their interfaces!), objects, inheritance and object hierarchy, . . .
- Implement, Fine-Tune, Test, Maintain

➠ **Note**: this is not a sequential process rather than a back-and-forth or cyclic one

Summary:

➠ Say what you mean

➠ understand what you are saying

- A common base class means common characteristics
- Public inheritance means *is-a* or *is-kind-of*

```
class derived : public base { /* ... */ };
```

  - every object of type `derived` is also an object of type `base`, but not vice-versa!
  - *Liskov substitution principle*:
    can every usage of an object of type `base` be replaced by an object of class `derived`?
  - *additional* functionality and/or data makes good subclass!

  Don't mistake *is-like-a* for *is-a* !

  - find higher abstraction, make parent: e.g. `set` is-like-a `list`, derive from `collection`

- Private inheritance means *is-implemented-in-terms-of*

```
class derived : private base { /* ... */ };
```

  - implementation issue; no design-level conceptional relationship
  - use only if access to protected members needed or to redefine virtual functions, otherwise use layering

- Layering (nested classes) means *has-a, is-part-of*, or *is-implemented-in-terms-of* between classes

```
class Inner { /* ... */ };
class Outer  { Inner I; /* ... */ };
```

- Class member(s) means *has-a, is-part-of*, or *is-implemented-in-terms-of* between objects

```
class foo { /* ... */ };
class bar1 { foo  I; /* ... */ };              // 1-to-1 mapping
class barN { foo *I; /* ... */ };              // 1-to-n mapping
```

- Member function(s) that take parameter of another class means *uses-a*

```
foo::func(const bar& b) { /* ... */ }
```


- *The Open/Closed Principle* (Bertrand Meyer): Software entities (classes, modules, etc) should be open for extension, but closed for modification.
  - New features can be added by *adding new code* rather than by *changing working code*
  - Thus, the working code is not exposed to breakage

- *The Liskov Substitution Principle*: Derived classes must be usable through the base class interface without the need for the user to know the difference.

- *Principle of Dependency Inversion*: Details should depend upon abstractions. Abstractions should not depend upon details.
  - All high level functions and data structures should be utterly independent of low level functions and data structures.

- Dependencies in the design must run in the direction of stability. The dependee must be more stable than the depender. (stability `:=` probable change rate)
  - The more stable a class hierarchy is, the more it must consist of abstract classes. A completely stable hierarchy should consist of nothing but abstract classes.
  - Executable code changes more often than the interfaces

- Use classes to represent concepts
- Keep things as private as possible
  - Once you publicize an aspect of your library (method, class, field), you can never take it out
- Watch out for the "giant object syndrome"
  - Objects represent concepts in your application, not the application itself!
  - Don’t add features "just in case"
- If you must do something ugly, at least localize the ugliness inside a class
- Don’t try technological fixes for sociological problems

- How should objects be created and destroyed?
  (constructors, destructor, class specific `new`/`delete`)
- How does object initialization differ from assignment? (constructors, `operator=()`)
- What does it mean to pass objects of new type by value? (copy constructor)
- What are the constraints on legal values for the new type?
  (error checking in constructors and `operator=()`)
- Does the new type fit into an inheritance graph? (virtuality of functions, ...)
- What kind of conversions are allowed? (constructors, conversion operators)
- What operators and functions make sense for the new type? (public interface)
  - Strive for class interfaces that are complete and minimal
- What standard operators and functions should be explicitly disallowed? (declare private)
- Who should have access to the members of the new type? (access modes, friends)
- How general is the new type? (class template?)

Goal: user-defined classes should be indistinguishable from built-in types!

Object-Oriented Design (both books use C++ for examples)

- Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing, 1994, ISBN 0-8053-5340-2.
- Budd, *Introduction to Object Oriented Programming*, Second Edition, Addison-Wesley, 1996, ISBN 0-201-82419-1.

Aspects of Object-Oriented Design in C++

- Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN 0-201-63361-2.
  - ➠ Provides an overview of the ideas behind patterns and a catalogue of 23 fundamental patterns
- Carroll and Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995, ISBN 0-201-51284-X.
  - ➠ Discusses many practical aspects of library design and implementation

➠ **But remember:**
Design and programming, like bicycle or swiming, cannot be learned by reading books!

Programming in C++

☆☆☆ class `std::string` ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

`std::string` Overview

- *string* `:=` sequence of *characters* `:=` objects of type `charT` (including e.g., `'\0'`)
- Standard `string` class is actually

```
namespace std {
   template < class charT,
              class traits = string_char_traits<charT>,
              class Allocator = allocator>
   class basic_string {
   public:
      static const unsigned npos = -1;
      /* ... */
   };

   typedef basic_string<char> string;
   typedef basic_string<wchar_t> wstring;

}
```

➠ To keep description of `std::string` simple in the following pages

- `string` is used instead of `std::basic_string<char>`
- `traits` and `Allocator` objects are ignored

- *Positions* in a string of *n* characters are specified
  - as an object of type `size_t` (typically `unsigned int`)
  - in the range *0* (first character) to *n-1* (last character)
- *Substrings* are specified by the start position `pos` and the number of characters `len`
  - length of substring is determined by the minimum of `len` and `size()-pos`
  - there is **no** separate `substring` class in the standard (but it is easy to define one)
- The `string` local special constant `npos`
  - can be used to specify the length *"all the remaining characters"*
  - is used by the search algorithms to indicate the result *"not found"*
- Member functions of `string` throw an exception
  - `out_of_range` if a specified `pos` is not in the range *0* to *n-1*
  - `length_error` if a specified `len` would construct a string larger than the maximal possible length

- Object definitions for all examples in the `std::string` chapter:

```
#include <string>
using namespace std;


const string str0 = "***";
const string str1 = "0123456789";

string str2 = str1;
string str3 = "345";

const char *cstr0 = "***";
const char *cstr1 = "abcdefghij";

char cstr2[16];

const char *cstr3 = "345";
```

- Arguments to `string` member functions fall into the following categories:

1.) an object of class `string` or a substring of it

```
(... const string& str, size_t pos=0, size_t len=npos ...)
```

Examples:

```
str1                                    // => "0123456789"
str1, 6                                 // => "6789"
str1, 3, 4                              // => "3456"
str1, 3, 20                             // => "3456789"
```

2.) a C/C++ built-in string (zero-terminated array of `char`) or the first `len` characters of it

```
(... const char* s, size_t len=npos ...)
```

Examples:

```
cstr1                                   // => "abcdefghij"
cstr1, 5                                // => "abcde"
"******"                                // => "******"
"******", 3                             // => "***"
```


3.) a repetition of `len` characters `c`

```
(.... size_t len, char c ...)
```

Examples:

```
5, '*'                                  // => "******"
2, 'a'                                  // => "aa"
```

4.) a (sub)`string` object specified by an *iterator* range

```
(.... InputIterator first, InputIterator last ...)
```

Examples:

```
str1.begin(), str1.end()                // => "0123456789"
str1.begin()+2, str1.end()-2            // => "234567"
str1.rbegin(), str1.rend()              // => "9876543210"
```

- Constructors

```
explicit string();
string(const string& str, size_t pos = 0, size_t len = npos);
string(const char* s, size_t len);
string(const char* s);
string(size_t len, char c);

template<class InputIterator>
string(InputIterator begin, InputIterator end);
```

- Destructor

```
~string();
```

- Examples

```
string s1;                    // => ""
string s2(str1, 3, 3);        // => "345"
string s3(cstr1, 3);          // => "abc"
string s4(3, '*');            // => "***"
string s5(str1.begin(), str1.end());          // => "0123456789"
string s6(str1.rbegin(), str1.rend());        // => "9876543210"
```


- Assignment operator with *value semantics* (i.e., conceptional deep copy)

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
```

- Member function `assign`: modifies and returns `this`

```
string& assign(const string&);
string& assign(const string& str, size_t pos, size_t len);
string& assign(const char* s, size_t len);
string& assign(const char* s);
string& assign(size_t len, char c);

template<class InputIterator>
string& assign(InputIterator first, InputIterator last);
```

- Examples

```
str2.assign(str1, 3, 3)              // => "345"
str2.assign(cstr1, 3)                // => "abc"
str2.assign(3, '*')                  // => "***"
str2.assign(str1.begin(), str1.end())          // => "0123456789"
```

- `string` provides member types and operations similar to those provided by STL containers
  - all STL algorithms can be applied to `string`
  - but only few are useful as `string` often provides more optimized versions directly
  - examples of useful ones are comparison or searches based on user-defined predicates
- Most useful are the usual iterators pointing to the first and one-after-the-last position
  - `string` iterators are *random access iterators*

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```


- Like STL's `vector<T>`, `string`s have a
  - *length* `:=` number of characters stored in string
  - *capacity* `:=` number of characters which could be stored in string without new memory allocation
- Length related member functions

```
size_t size() const;                    // returns length
size_t length() const;                  // same as size()

size_t max_size() const;                // maximal possible length

bool empty() const;                     // size() == 0?
```

- Capacity related member functions

```
size_t capacity() const;                // returns capacity

void resize(size_t len, char c = '\0'); // shrink or enlarge capacity to len
                                        // new characters are initialized with c

void reserve(size_t len = 0);           // inform string that at least len
                                        // characters are needed, change
                                        // capacity as needed
```

std::string — Element Access

- Individual characters of a `string` can be accessed through subscripting.
  It comes in two forms: with and without range check
  - The result of using `operator[](pos)` is *undefined* if `pos >= size()`

    ```
    const_reference operator[](size_t pos) const;
    reference operator[](size_t pos);
    ```

  - Member function `at(pos)` throws `out_of_range` if `pos >= size()`

    ```
    const_reference at(size_t pos) const;
    reference at(size_t pos);
    ```

  Otherwise, both return the character at position `pos`

- To access last character of a string `str` use `str[str.size()-1]`

- Examples:

```
str1[5]                    // => '5'
str1.at(5)                 // => '5'

str1[42]                   // => undefined
str1.at(42)                // => throws out_of_range
```


std::string — Comparisons

- Strings can be compared to (sub)strings or (sub)arrays of characters
- If `pos1` and `len1` are specified only this part of the string is compared
- Returns
  - `0` if the (sub)strings are equal
  - a negative number if the string is lexicographically before the argument character object
  - a positive number otherwise

```
int compare(const string& str) const;
int compare(size_t pos1, size_t len1, const string& str) const;
int compare(size_t pos1, size_t len1, const string& str,
        size_t pos2, size_t len2) const;
int compare(const char* s) const;
int compare(size_t pos1, size_t len1, const char* s,
        size_t len2 = npos) const;
```

- Examples

```
str1.compare(str1)          // => 0
str1.compare(3, 3, str1)    // => >0
```

- Member function `append` allows adding characters described by the arguments to the end (short-cut for `insert` at the end of the string)
- `append` modifies the string itself (`*this`) and returns the modified string

```
string& append(const string& str);
string& append(const string& str, size_t pos, size_t len);
string& append(const char* s, size_t len);
string& append(const char* s);
string& append(size_t len, char c);

template<class InputIterator>
string& append(InputIterator first, InputIterator last);
```

- `operator+=` and `push_back` are provided as a conventional notation for the most common forms of `append`

```
string& operator+=(const string& str);
string& operator+=(const char* s);
string& operator+=(char c);

void push_back(const char);
```


- Examples

```
str2.append(str1)            // => "012345678901234567890"
str2.append(str1, 3, 3)      // => "0123456789345"
str2.append(cstr1, 3)        // => "0123456789abc"
str2.append(cstr1)           // => "0123456789abcdefghij"
str2.append(3, '*')          // => "0123456789***"
str2.append(str1.rbegin(), str1.rend())
                             // => "012345678998765432100"

str2 += str1                 // => "012345678901234567890"
str2 += cstr1                // => "0123456789abcdefghij"
str2 += '*'                  // => "0123456789*"

str2.push_back('*')          // => "0123456789*"
```

- Member function `insert` allows adding characters described by the arguments to the string
- characters can either be inserted
  - *before* the index position `pos`
    - ➠ modifies the string itself (`*this`) and returns the modified string

```
string& insert(size_t pos, const string& str);
string& insert(size_t pos, const string& str,
        size_t pos2, size_t len);
string& insert(size_t pos, const char* s, size_t len);
string& insert(size_t pos, const char* s);
string& insert(size_t pos, size_t len, char c);
```

  - *before* the position described by iterator into the same string `p`

```
iterator insert(iterator p, char c);

void insert(iterator p, size_t len, char c);

template<class InputIterator>
void insert(iterator p,
        InputIterator first, InputIterator last);
```


- Examples

```
str2.insert(3, str0)         // => "012***3456789"
str2.insert(3, str0, 1, 1)   // => "012*3456789"
str2.insert(3, cstr0, 1)     // => "012*3456789"
str2.insert(3, cstr0)        // => "012***3456789"
str2.insert(3, 3, '*')       // => "012***3456789"

str2.insert(str2.begin(), '*')       // => "*0123456789"
str2.insert(str2.begin(), 3, '*')    // => "***0123456789"
str2.insert(str2.begin(), str1.rbegin(), str1.rend())
                                     // => "98765432100123456789"
```

- Member function `erase` deletes a range of characters described by the arguments from the string
- Delete a range of characters described by a start position `pos` and a length `len`

```
string& erase(size_t pos = 0, size_t len = npos);
```

- Delete the single character specified by iterator `position` or
  delete a range of characters described by iterator pair `first` and `last`
  - ➠ Returns an iterator pointing to the element immediately following the element(s) being erased

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void clear();                          // == erase(begin(), end())
```

- Examples

```
str2.erase(3, 3)            // => "0126789"
str2.erase(3)               // => "012"
str2.erase(str2.begin())    // => "123456789"

str2.erase(str2.begin(), str2.end())    // => ""
str2.clear()                            // => ""
str2.erase()                            // => ""
```


- Member function `replace` allows changing a range of characters to other characters described by the arguments
  - ➠ replace `:=` assignment to a substring
- `replace` modifies the string itself (`*this`) and returns the modified string
- The range of characters to `replace` can be described by
  - ❍ a start position `pos1` and the length `len1`

```
string& replace(size_t pos1, size_t len1, const string& str);
string& replace(size_t pos1, size_t len1, const string& str,
        size_t pos2, size_t len2);
string& replace(size_t pos, size_t len1, const char* s,
        size_t len2);
string& replace(size_t pos, size_t len1, const char* s);
string& replace(size_t pos, size_t len1, size_t len2, char c);
```

or

- a pair of iterators `i1` and `i2`
  - `i1` and `i2` must be valid iterators into the string (`*this`)

```
string& replace(iterator i1, iterator i2, const string& str);
string& replace(iterator i1, iterator i2, const char* s,
                size_t len);
string& replace(iterator i1, iterator i2, const char* s);
string& replace(iterator i1, iterator i2, size_t len, char c);
template<class InputIterator>
string& replace(iterator i1, iterator i2,
                InputIterator j1, InputIterator j2);
```


- Examples

```
str2.replace(3, 3, str0)            // => "012***6789"
str2.replace(3, 3, str0, 1, 1)      // => "012*6789"
str2.replace(3, 3, cstr0, 1)        // => "012*6789"
str2.replace(3, 3, cstr0)           // => "012***6789"
str2.replace(3, 3, 3, '*')          // => "012***6789"
str2.replace(str2.begin(), str2.end(), str0)        // => "***"
str2.replace(str2.begin(), str2.end(), cstr0, 1)    // => "*"
str2.replace(str2.begin(), str2.end(), cstr0)       // => "***"
str2.replace(str2.begin(), str2.end(), 3, '*')      // => "***"
str2.replace(str2.begin(), str2.end(), str0.begin(), str0.end())
                                                    // => "***"
```

- Conversion to C/C++ style string
  - returns pointer to `string` owned character array containing copy of string

    ```
    const char* data() const;
    ```
  - the same thing but `'\0'` terminated

    ```
    const char* c_str() const;
    ```
  - copy (parts of) string into user-supplied buffer (**note**: not `'\0'`-terminated!)

    ```
    size_t copy(char* s, size_t len, size_t pos = 0) const;
    ```
- Exchange the contents of two strings

  ```
  void swap(string&);
  ```
- Select substring (`len` characters starting from position `pos`)

  ```
  string substr(size_t pos = 0, size_t len = npos) const;
  ```
- Examples

```
int i = str1.copy(cstr2, 3, 3); cstr2[i] = '\0'    // cstr2="345"
str2.swap(str3)                 // => str2="345" str3="0123456789"
str1.substr(3, 3)               // => "345"
```


- Find *substring* described by arguments in string starting at position `pos`

```
size_t find(const string& str, size_t pos = 0) const;
size_t find(const char* s, size_t pos, size_t len) const;   /*X*/
size_t find(const char* s, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

- Find *substring* described by arguments in string searching *backwards* from `pos`

```
size_t rfind(const string& str, size_t pos = npos) const;
size_t rfind(const char* s, size_t pos, size_t len) const;  /*X*/
size_t rfind(const char* s, size_t pos = npos) const;
size_t rfind(char c, size_t pos = npos) const;
```

- **Note**: in forms marked `/*X*/` `len` characters of `s` are searched from position `pos` in `*this`
- Both return start position of substring in string, or `string::npos` if not found
- Examples

```
str1.find(str3, 0)                  // => 3
str1.find(cstr1, 0)                 // => string::npos
str1.rfind(str3)                    // => 3
str1.rfind('3', string::npos)       // => 3
```

- Find first/last *character* out of character set described by arguments in string forward/backward from position `pos`

```
size_t find_first_of(const string& str, size_t pos = 0) const;
size_t find_first_of(const char* s, size_t pos, size_t len)
          const;
size_t find_first_of(const char* s, size_t pos = 0) const;
size_t find_first_of(char c, size_t pos = 0) const;

size_t find_last_of(const string& str, size_t pos = npos) const;
size_t find_last_of(const char* s, size_t pos, size_t len) const;
size_t find_last_of(const char* s, size_t pos = npos) const;
size_t find_last_of(char c, size_t pos = npos) const;
```

- Both return start position of character in string, or `string::npos` if not found

- Examples

```
str1.find_first_of(str3, 0)          // => 3
str1.find_first_of(cstr3, 0, 3)      // => 3
str1.find_last_of(cstr3, string::npos)  // => 5
str1.find_last_of('3', string::npos)    // => 3
```


- Find first/last *character* **not** in character set described by arguments in string forward/backward from position `pos`

```
size_t find_first_not_of(const string& str, size_t pos = 0)
         const;
size_t find_first_not_of(const char* s, size_t pos, size_t len)
         const;
size_t find_first_not_of(const char* s, size_t pos = 0) const;
size_t find_first_not_of(char c, size_t pos = 0) const;

size_t find_last_not_of(const string& str, size_t pos = npos)
         const;
size_t find_last_not_of(const char* s, size_t pos, size_t len)
         const;
size_t find_last_not_of(const char* s, size_t pos = npos) const;
size_t find_last_not_of(char c, size_t pos = npos) const;
```

- Both return start position of character in string, or `string::npos` if not found

- Examples

```
str1.find_first_not_of(cstr3, 0)            // => 0
str1.find_last_not_of(str3, string::npos)    // => 9
```

- Concatenation is implemented as global function `operator+`
  - to allow non-`string` arguments (character arrays and single `char`'s) on left side of operator
  - to avoid creation of temporary string objects resulting from automatic conversion

```
string operator+(const string& lhs, const string& rhs);
string operator+(const char* lhs,   const string& rhs);
string operator+(char lhs,          const string& rhs);
string operator+(const string& lhs, const char* rhs);
string operator+(const string& lhs, char rhs);
```

- Examples

```
str1 + str1                 // => "01234567890123456789"
cstr1 + str1                // => "abcdefghij0123456789"
'*' + str1                  // => "*0123456789"
str1 + cstr1                // => "0123456789abcdefghij"
str1 + '*'                  // => "0123456789*"
```


- The equality operators `operator==` and `operator!=` are also implemented as global functions for the same reasons

```
bool operator==(const string& lhs, const string& rhs);
bool operator==(const char* lhs,   const string& rhs);
bool operator==(const string& lhs, const char* rhs);

bool operator!=(const string& lhs, const string& rhs);
bool operator!=(const char* lhs,   const string& rhs);
bool operator!=(const string& lhs, const char* rhs);
```

- Examples

```
str1 == str1                // => true
cstr1 == str1               // => false
str1 == cstr1               // => false

str1 != str1                // => false
cstr1 != str1               // => true
str1 != cstr1               // => true
```

- So are the rest of the comparison operators

```
bool operator< (const string& lhs, const string& rhs);
bool operator< (const string& lhs, const char* rhs);
bool operator< (const char* lhs, const string& rhs);

bool operator> (const string& lhs, const string& rhs);
bool operator> (const string& lhs, const char* rhs);
bool operator> (const char* lhs, const string& rhs);

bool operator<=(const string& lhs, const string& rhs);
bool operator<=(const string& lhs, const char* rhs);
bool operator<=(const char* lhs, const string& rhs);

bool operator>=(const string& lhs, const string& rhs);
bool operator>=(const string& lhs, const char* rhs);
bool operator>=(const char* lhs, const string& rhs);
```

- Examples

```
str1 >= str1                // => true
cstr1 > str1                // => true
cstr1 < str1                // => false
```


- `operator>>` reads a *whitespace-terminated word*
  - string is expanded as needed to hold the word
  - initial and terminating whitespace is not entered into the string

```
istream& operator>>(istream& is, string& str);
```

- `operator<<` writes string contents to an output stream

```
ostream& operator<<(ostream& os, const string& str);
```

- `getline` reads a line terminated by `delim` into `str`
  - string `str` is expanded as needed to hold the line
  - delimiter `delim` is not entered into the string

```
istream& getline(istream& is, string& str, char delim = '\n');
```

- Exchange the contents of the strings `lhs` and `rhs`

```
void swap(string& lhs, string& rhs);
```

Programming in C++

☆☆☆ English – German Dictionary ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

abstract base class	abstrakte Basisklasse
abstract/concrete constness	abstrakte(logische)/tatsächliche Unveränderbarkeit
access specifier	Zugriffsspezifikation
aligment	(Speicher)Ausrichtung
arithmetic types	arithmetische Typen
array	Feld
automatic object	automatisches Objekt
assignment	Zuweisung
associativity	Assoziativität
basic types	elementare Typen, grundlegende Typen
base class	Basisklasse
catch block	Ausnahmebehandlungsblock
cast	cast, explizite Typ(en)konversion/Typ(en)umwandlung
character	Textzeichen
class	Klasse
comment	Kommentar
compound statement	(Anweisungs)Block, Anweisungsfolge, zusammengesetzte Anweisung
conditional compilation	bedingte Übersetzung

constructor	Konstruktor
control flow	Steuerfluß
conversion (operator)	(Typ)Umwandlungsoperator
diasy-chaining	aneinanderreihen, verketten, kaskadieren
data hiding	Geheimnisprinzip
data member	Datenelement
declaration	Deklaration
definition	Definition
default parameter	Standardparameter(wert), vorbelegte Parameter Parameterwertvorgabe
derived class	abgeleitete Klasse
derived types	abgeleitete Typen
destructor	Destruktor
do until loop	Durchlaufschleife
dynamic binding	dynamische Bindung
encapsulation	Kapselung
enumeration	Aufzählung
exception handling	Ausnahme(fall)behandlung
expression	Ausdruck

Programming in C++ English – German Dictionary

floating-point	Fliesskomma
for loop	Zählschleife
function body	Funktionsrumpf
function call	Funktionsaufruf
function head	Funktionskopf
global	global
header file	Schnittstellendatei
heap	dynamischer Speicher, Freispeicher
hide	überdecken, verdecken
identifier	Bezeichner
include directive	Dateieinfügeanweisung
indirection	Dereferenzierung
inheritance	Vererbung
inline	inline
integer	ganzzahlig
integral promotion	integrale Promotion
integral types	integrale Typen
iteration	Schleife

jump statement	Sprunganweisung
keyword	Schlüsselwort, reservierter Bezeichner
label	Sprungmarke
labeled statement	benannte Anweisung
literal	Literal (auch "Konstante")
member function	Elementfunktion, Methode
member template	Elementschablone
memory allocation	Speicheranforderung
memory deallocation	Speicherfreigabe
memory management	Speicherverwaltung
multiple Inheritance	Mehrfachvererbung, mehrfache Vererbung
namespace	Namensraum, Namensbereich
nested classes	(ein)geschachtelte Klassen
operator	Operator
operator delete	Löschoperator, Speicherfreigabe
operator new	Erzeugungsoperator, Speicherbelegung
overloading	Überladen
runtime type identification (RTTI)	Typermittlung zur Laufzeit, Laufzeit-Typinformation

placement	Plazierung
pointer	Zeiger
polymorphism	Polymorphie
precedence	Priorität
preprocessor directives	Präprozessordirektiven
private member	privates Element
protected member	geschütztes Element
public member	öffentliches Element
pure virtual	rein virtuell, abstrakt
qualified name	qualifizierter Bezeichner / Name
recursive	rekursiv
reference	Verweis / Referenz
relational operator	Vergleichsoperator
return type	Funktionswerttyp
scope	Gültigkeitsbereich, Bezugsrahmen
signed	vorzeichenbehaftet
stack	Stapel
statement	Anweisung
static binding	statische Bindung

static member	Klassenmethode, Klassenvariable, objektloses Element
static object	statisches Objekt
storage class	Speicherklasse
struct	Struktur, Verbund, Rekord
template	Schablone, generische / parametrisierte Klasse, Musterklasse
throw exception	Ausnahmebehandlungs(code) anstossen, Ausnahme auswerfen
unsigned	vorzeichenlos
union	varianter Rekord
virtual function	virtuelle Funktion
while loop	Abweisungsschleife

Programming in C++

☆☆☆ Index ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W